A Case for Runtime Code Generation

David Keppel, Susan J. Eggers and Robert R. Henry

Technical Report 91-11-04

Department of Computer Science and Engineering, FR-35
University of Washington, Seattle, WA 98195 USA

A Case for Runtime Code Generation

David Keppel, Susan J. Eggers and Robert R. Henry

Abstract

We define *runtime code generation* (RTCG) as dynamically adding code to the instruction stream of an executing program. It has been in use since the 1940s, but fell from favor because changing hardware and software technologies made it less profitabile and its *ad-hoc* implementation made it nonportable. In this paper we argue for it's return. We base our arguments on ongoing changes in hardware technology, software technology, and workloads. Each of these changes brings about circumstances in which dynamically optimized code can execute fast enough to pay for the runtime overhead of creating it. We support our analysis with concrete examples. We compare static-code and RTCG implementations of several applications and show that RTCG leads to performance improvements.

Keywords: computer architecture; code generation; compiler optimization; partial evaluation; performance

1 Introduction

We define *runtime code generation* (RTCG) as dynamically adding code to the instruction stream of an executing program. For example, a user may interactively enter an expression that is to be evaluated once for each item in a data set, such as a regular expression to match lines in a file. An RTCG implementation dynamically compiles the expression into a special-case function and then calls the function once per datum. By comparison, a *static-code* implementation translates the expression to some intermediate form and then interprets it with a statically-compiled interpreter.

In some systems the view of RTCG is strongly influenced by the definition of "code". For example, the bytecodes for a virtual machine may be executed by a program, by microcode, or directly by hardware. We restrict ourselves to instructions that are executed directly by hardware: runtime generation of native machine code.

Runtime code generation has been in use since the earliest programmable-store computers. In those days, memory was tight and clever *ad-hoc* self-modifying code sequences were often smaller and thus faster. Large programs were run by overlaying memory. Later, instruction update was sometimes codified. For example, procedure linkage on the CDC 6600 was implemented by patching the return instruction on procedure entry.

Changing technology reduced the demand for RTCG. Memories grew, reducing the I/O component of memory access times. Text sharing made reentrant code important. The proliferation of architectures and implementations meant that portability became a major issue. Portability was achieved using high-level languages, and most lacked ways of expressing the dynamic generation of new code. Finally, monolithic compilers encouraged a dichotomy between compile time and run time, and compiler writers, architects and general users gradually forgot that A's data is B's code, and that nothing fundamentally prevented A and B from being the same.

Why do we bring it up again? One reason is that despite technology advances and bad press, RTCG is still being used in a large number of research and production systems. It is functionally required for incremental compilers [PS84], dynamic linkers [HO91, sys90], and debuggers [Kes90]. Other systems use RTCG to improve performance: high-performance virtual machines [DS84, May87], interactive systems that demand good response time on repeated actions or inner loops [PLR85, MP89], and database query optimization [CAK+81]. Finally, at least one processor uses runtime generation of microcode [DM87], and runtime programming of gate arrays has been proposed [JFnt].

A second reason for reexamining RTCG is that hardware implementations are changing in ways that may

The first two authors may be reached at Department of Computer Science and Engineering, FR-35 University of Washington Seattle, Washington 98195. The third author may be reached at Tera Computer Company 400 N. 34th Street, Suite 300, Seattle, Washington 98103.

This work was supported by NSF PYI Award #MIP-9058-439 and Sun Microsystems, Inc.

again favor it. When memories got larger, code size became less important. However, small fast memories have been reintroduced as caches. Current trends are for smaller first-level caches and larger penalties for misses [Jou90, MBB+91]. In addition, implementation-dependent compiler optimizations, such as loop unrolling, can affect cache performance substantially; RTCG lets a program dynamically adapt to particular implementations within a family of architectures.

Third, recent advances in software methods may solve some portability and programability problems. Machine-dependencies can be encapsulated in a retargetable code generator, while instruction space coherency and protection can be hidden using a portable interface [Kep91]. In addition, partial evaluation [JSS89] may be useful in automatically deriving runtime compilers that are optimized for the particular application.

Two original and major problems still remain. First, many systems use RTCG in *ad-hoc* ways, so their generality and portability suffer. Second, although the large number of existing systems suggest that RTCG can be profitable, there are few comparisons of RTCG and statically-compiled alternatives. There are no models or metrics to decide when designers should manually apply RTCG, nor are there algorithms to do so automatically. Only intuition and experience guide its use.

Despite these problems, RTCG merits serious investigation. There is still a functional need for it in some systems, changing hardware may favor its use, and changing software may make it easier to use. These factors compel us to undertake empirical studies, develop better models, and search for ways to apply it automatically and portably.

In this paper, we discuss factors that led to the dismissal of RTCG from the programmer's toolkit (§2) and recent advances in hardware and software that may now make it more useful (§3). We then present a cost model that compares static and runtime compiling and consider tradeoffs between dynamic compile costs and code quality (§4). We examine several applications that use RTCG profitably, and discuss reasons for the improvement (§5). Finally, we discuss some of the open RTCG issues (§6).

2 The Fall From Favor

Runtime code generation was used in early systems, but lost popularity as practices changed and as other concerns became more important.

In older systems, memory space was always tight. RTCG was used in a variety of ways to build programs that fit in small memories. For example, self-modifying sequences were sometimes smaller. Parts of the program could be overlaid and moved to and from disk. Fast executable code could be generated on demand from a compact representation. As first-level memories became larger, program size became relatively unimportant to program performance. It was possible to expend memory, statically generating many specialized alternatives of a general procedure. Thus, RTCG became less profitable.

Portability became an issue with the proliferation of architectures and implementations, and the cost of writing and maintaining software for each platform. Many portability problems were solved by switching to high-level languages, but most high-level languages lack constructs for specifying RTCG. Therefore, RTCG was often done with a *template compiler* that "patches together" machine-code fragments at runtime. Such RTCG implementations had good performance, but had to be written partly in assembly language. As an alternative, running programs could generate source for a high level language, feed that source to a traditional "heavyweight" compiler, then dynamically link the generated object code with the running program. However, invoking the "heavy" compiler was so expensive that it was far less likely RTCG would be profitable.

Portability between implementations of one architecture was also a problem. A program might run correctly on one implementation and fail on another. For example, a machine without an instruction cache or prefetch will "see" instruction space changes immediately, while an implementation with caches may not see changes until the old instruction is knocked out of the cache.

Debugging RTCG was difficult because many uses were unstructured. Unstructured RTCG makes it hard to predict how and when the code changes, making it difficult to reason about the program. Further, it is not obvious how to debug code that is generated dynamically. Thus, static code was cheaper since it was easier to reason about.

Finally, few comparative studies demonstrated RTCG's performance benefits, so it was difficult to determine which programs were appropriate for an RTCG implementation. For example, it is reported that database queries are much faster if the query is dynamically optimized and compiled [CAK+81]; yet there are no published quantitative analyses of when, how much, and why it is faster.

3 Changes In Technology

Recent changes in hardware and software make RTCG more likely to be useful. Some changes, such as caches, recreate the initial motivations for self-modifying code, and others, such as portable code generator interfaces, solve previous problems.

3.1 Hardware Technology

Large memories made program performance independent of code size. However, caching, especially the current trend of smaller first-level caches and growing miss penalties [Jou90, MBB+91], has reintroduced the size cost.

Implementations of the same architecture may have caches whose size and miss penalties vary by more than an order of magnitude. Statically-compiled binaries must run well on all members of an architectural family. So, for example, loops can be unrolled statically only for the smallest cache size in the target family, but can be unrolled dynamically for any cache configuration.

A specialized routine produced by RTCG may be smaller than its statically-compiled general-purpose counterpart and may fit into a cache where the larger one will not. For example, a dynamically compiled `bitblt` routine may be one-quarter the size of the static-code version [PLR85]. In-cache execution can increase the instruction issue rate and reduce out-of-cache traffic. Dynamically-generated code may also have fewer branches than the static code, thus improving prefetching performance. Finally, dynamic specialization may eliminate dynamically dead code and place dynamically-adjacent code adjacent in the cache, reducing cache conflicts.

Variables that are constant over a dynamically-compiled routine's life (*runtime constants*) may be used as immediate literals in the dynamically-compiled code, just as normal constants are literals in static code. This has several benefits. Immediates in the instruction stream may be prefetched into the cache. The cost for accessing the immediates may be less than an explicit load. Finally, values may be read simultaneously from instruction and data caches, increasing the effective memory bandwidth.

Another hardware trend is that CPUs are getting faster relative to the memory hierarchy. Thus, it may be profitable to expend CPU time to alleviate other bottlenecks. For example, it might sometimes be better to generate code at runtime than to page in statically-compiled code [DS84, SC91]

3.2 Software Technology

Several trends in software technology may also contribute to RTCG's utility. First, as optimizers have improved, interest has turned from data-independent optimizations to data-dependent optimizations. For example, profiles are used to improve branch prediction and drive code placement. However, a profile represents an execution that might not be typical of a given execution [Wal91, McF91]. By comparison, data-dependent optimizations done at runtime use the program's real data values and real behavior. Dynamic compilation would also allow the choice of data placement algorithms for a particular layout to be made at runtime [SC91].

Second, retargetable compilers should allow RTCG problems to be stated in a machine-independent way without sacrificing either the quality of the machine-dependent code or the speed of generating it. Traditionally, implementors could either use a fast but non-portable template compiler, or use a portable compiler and suffer long dynamic compile times. Current automatic code generator generators can build code generators that rival hand-written code generators in both speed and code quality, and should be able to produce object code directly from a portable intermediate representation, executing as few as a few hundred instructions per instruction generated [FH91], a rate that is comparable to hand-written code generators. In some circumstances, dynamic compiler performance can be improved even further using a template compiler (to be discussed in §4.3). The extra specialization makes code generation even faster, though perhaps at the expense of code quality. One of our experiments (§5.3) shows that templates can be derived automatically from a machine-independent specification, and that code generation can still be done in a few tens of instructions executed per instruction generated.

Third, using a code generator hidden behind a good interface will hide the details of instruction space changes. That will then allow instruction caching issues to be encapsulated in a portable interface, solving instruction cache portability problems [Kep91].

Finally, partial evaluation may be able to derive fast special-purpose compilers automatically from their general-purpose counterparts plus the source code for the routine to be compiled. Although partial evaluation itself is a well-established technique used in compiler transformations [Har77], our initial experiments with Similix [JSS89] lead us to believe that partially evaluating a program as large and complex as a conventional compiler or code generator is not yet possible. However, RTCG will clearly benefit from advances in the practice of partial evaluation.

3.3 Workload Changes

A third change is in workload composition. RTCG's (typically) large startup times mean that for small datasets, it is often more expensive. However, for larger datasets, the cost of compilation can be amortized over more invocations of the code, so RTCG can have asymptotically better performance.

RTCG can also be used to flatten indirection needed by some new workloads. For example, traditional cache simulators had a small number of parameters describing the configuration of a single cache, such as cache and block size and associativity. Newer multi-cache

simulators have over a hundred configuration parameters describing, in addition to the previous characteristics, each cache's latency, bandwidth, and relative miss costs. The cache parameters vary from run to run, but are fixed over any given run. Thus, a runtime compiler can optimize the simulator for each run, compiling the cache parameters into the code as constants [PHH88].

4 Performance Opportunities

Runtime code generation can lead to better performance simply because a compiler can generate better code when it has more information, and all information is available at runtime. For example, if a variable becomes a runtime constant, then it can be propagated, triggering constant folding and dead code elimination; if an array is known at runtime to be small, then iterations may be fully unrolled. The crux of the argument is that, although it costs something to run the compiler at runtime, RTCG can sometimes produce code that is enough faster to pay back the dynamic compile costs. (We assume that most code is noncritical and is compiled statically; it is, therefore, reentrant and sharable. Only the small part of the code that is a bottleneck is compiled dynamically.)

A static compiler can generate code using one of three strategies. It can generate *generic* code that works for all input values but which may be inefficient. Alternatively, it can generate *fully-cased* code with a prologue that dispatches to a different optimized fragment for each input value; however, there may be a huge code size explosion. Third, the compiler can generate *common-cased* code, with a generic routine plus a specialized fragment for each common case; unfortunately, it may be hard to identify the common cases.

In comparison, RTCG can generate code that is specific to the particular data values of a particular invocation. Thus, every case can get code that is as specialized as the fully-cased code, but the static compiler does not need to generate code for every possible alternative.

We could take this too far. Note that all data is available at runtime, so the dynamic compiler could consume all input data and produce the final result. This is equivalent to simply running the statically generated code. Our goal is to find a balance point where the runtime compiler consumes some but not all input data, does little work, produces substantially better code, and then runs the improved code on the remaining data.

4.1 Bitblt: An Example

We now examine `bitblt`, a bit-transfer operation used frequently in the core of raster graphics systems [PLR85]. The example is used to show how runtime data can be used to improve performance and motivates the cost model presented in §4.2.

```
j = dst.left
mask = lmask
a = src[src.left]
for i = src.left+1 to src.right do
  b = src[i]
  m = (a << ls) | (b >> rs)
  case (op) of
  AND: dst[j]=dst[j]^((~m&dst[j])&mask)
  NOT: dst[j]=dst[j]^mask
  OR:  dst[j]=dst[j]|(m&mask)
  ... 13 more cases ...
  end case
  j = j + 1
  ... end-of-line tests ...
end for
```

Figure 1: Prototypical `bitblt` Inner Loop

`bitblt` merges a source rectangle with a destination rectangle, using operations such as `and`, `or`, `xor`, etc. It is general, being able to merge rectangles of all sizes and alignments, using any merge operator. That same generality makes it difficult to implement efficiently.

The outer loop of `bitblt` executes once per horizontal line in the source rectangle. The inner loop reads the source rectangle one machine word at a time and combines it with a word of the destination according to a specific operator. The inner loop is complicated, because the source and destination may start at arbitrary bit boundaries, need not be aligned with each other and may overlap. Consequently, each source word must be aligned with its destination word, and some of the leftmost and rightmost bits of each destination line may be unaffected.

Consider the prototypical inner loop from `bitblt`, shown in a C-like language in Figure 4.1. The parameter `op` is constant over the entire call. Thus, for any one invocation of `bitblt`, the `case` statement selects the same alternative each time through the loop. Interchanging the `for` loop and the `case` statement eliminates the `case` on each iteration of the inner loop, but at the expense of replicating the inner loop sixteen times, once for each alternative in the `case`. Similarly, variables `ls` and `rs` are constant over the inner loop, enabling optimizations of shifts and masks. The code can be further optimized for overlapping rectangles. Although each of the optimizations can be done statically, each requires replicating the code. Fully-casing may lead to an implementation as large as 1MB [PLR85]. Common-casing is hard, since a one-pixel change in the rectangles' positions or size can affect half a dozen op-

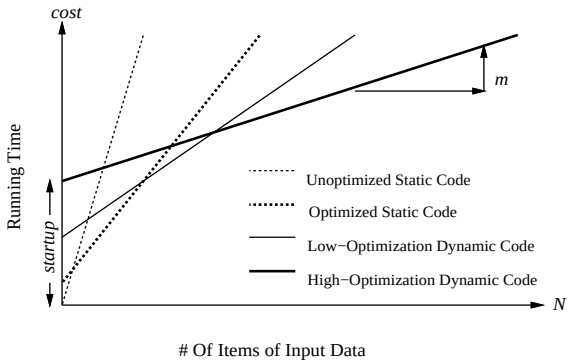

Figure 2: Startup and Asymptotic Costs

timizations.

A RTCG implementation dynamically compiles a custom inner loop on entry to `bitblt`. For any given invocation, the dynamic compiler performs all the optimizations listed above, and, in addition, can perform optimizations that the static compiler cannot. For example, the loop may be unrolled for the specific line length, which is profitable for short inner loops.

For large rectangles, the optimizations more than pay for the compile time. A straightforward RTCG version of `bitblt` is about 10 times faster than a straightforward static-code implementation [Mir87]. An optimized RTCG `bitblt` runs twice as fast as machine-dependent static code and four times as fast as portable C code that has been partially but not fully cased [Loc87]. The dynamic compiler is substantially smaller than the statically optimized `bitblt`, 2.8KB instead of 8.0KB [Loc87].

`bitblt` illustrates how RTCG can improve code quality and also demonstrates that the benefit is data-dependent: for small rectangles, the compile time would always be larger than any savings from dynamic compilation. Thus, small rectangles are handled by statically-generated code [PLR85, Loc87].

4.2 Cost Model

We now generalize several performance factors exposed in the `bitblt` example. We model execution time using a simple linear model, $cost = startup + m \times N$, where the total execution time ($cost$) is a function of some startup cost ($startup$), the marginal cost per data item (m) and the number of data items (N). Figure 4.2 shows a simple model of the execution cost of several implementations of a hypothetical code fragment.

When running static code, such as a loop, over some dataset, the total cost is the cost to enter and exit the loop ($startup_s$), plus a cost per data item (m_s). An unrolled static-code loop has a lower marginal cost per data item. However, it has extra code to handle a number of iterations that is not a multiple of the unrolling factor, and thus a higher startup cost.

Similarly, running dynamic code has two costs: $startup_d$ is the cost to dynamically compile the code, plus the cost to enter and exit the code; m_d is the cost per data item. Invoking the compiler at runtime increases the startup time compared to static code. However, the RTCG optimizer knows more and so may generate better code (smaller m_d). The *breakeven point* occurs when the costs of the static and dynamic code are the same for a given N.

There is also a tradeoff between compile time and overall execution time. A runtime compiler with a better optimizer may take longer to run but produce better code. The result has worse performance on small data sets but better performance on large data sets.

Finally, we define the *granularity* of code generation as the number of instructions executed between changes to the instruction space. Self-modifying code is fine-grained. We expect that most contemporary uses of RTCG will be coarse-grained, so that the costs of code generation and instruction space manipulation can be amortized over many instructions.

4.3 Runtime Compilers

Compilation speed is crucial to RTCG's performance. Runtime compilers can be faster than conventional compilers because the runtime compiler does not need to compile arbitrary programs, but only certain constructs. Thus, some compiler phases may be discarded, while others may be specialized for the particular program. Both optimizations reduce $startup_d$.

For compiling a fixed source program,[1] all steps in the front end of the compiler, as well as some in the back end, can be run statically. The remaining phases can sometimes be collapsed or specialized in a program-dependent way. For example, if a routine R never makes function calls, then a specialized compiler for R can be ignorant of function calling concerns.

Many systems do the specialization manually, using statically generated *templates*, which are application-specific machine-code sequences that contain "holes". A dynamic compiler is written by hand for the particular application. The compiler is retargeted by rewriting the machine-code templates. The runtime compiler concatenates templates and fills in the holes with values computed at runtime. The latter approach is similar to generating code using machine-dependent virtual machine instructions automatically derived from a training suite [Hen87], but in most existing RTCG systems

[1] When the source code is not known statically, RTCG can still be profitable, since the user input will be translated to some intermediate representation anyway [Tho68], and code generation may be but a small part of the total cost [CAK+81].

the templates are derived manually. With a template compiler, nearly all the compilation is static, including some parts of code generation, assembly, and linking. Thus, runtime compilation can be done in just a few tens of instructions per instruction generated.

5 Measured Performance

We now examine three applications that use RTCG to improve performance: a virtual machine interpreter, a data decompression algorithm and a record comparator. A common feature of these programs is that they use indirection to allow runtime flexibility, but the target of the indirection changes rarely and only in well-defined places. Consequently, the RTCG versions were faster than their statically-compiled counterparts.

5.1 Virtual Machines

A virtual machine is a program that simulates the actions of some real or imagined machine. A virtual machine interpreter has a primitive for each instruction in the virtual machine's instruction set. Simulating each virtual machine instruction with a static-code interpreter requires four steps: dispatch (to the primitive), load operands, execute, and store operands.

An RTCG implementation collects a sequence of virtual machine instructions and compiles them to native machine code. Dispatching is done during compilation and the machine code is cached, so both compiling and dispatching costs can be avoided on reexecution. Also, the dynamic compiler can optimize across several virtual machine instructions.

The Smalltalk-80 byte-coded virtual machine has both static-code and RTCG implementations. Fast static-code implementations [Mir87] interpret the byte codes using a threaded code interpreter. An RTCG implementation that does simple optimizations [DS84] is 40% faster than fast static-code implementations [Mir91]. Better RTCG optimization would cause longer compile times but yield code asymptotically 5-10 times faster than the static code [CU91].

Another virtual machine, `g88`, is a statically-compiled 88000 simulator that translates instructions to threaded code [Bed89]. It runs on the 68000 and takes, on average, about twenty 68000 machine instructions to simulate one 88000 instruction. Simulating only user-mode code would drop the ratio to ten to one [Bed91]. `shade` is a SPARC simulator and instruction-level profiler that runs on a SPARC and uses RTCG [CK91]. On the `gcc` and `doduc` benchmarks from the SPEC suite, `shade` spends about 10% and 0.1% of its total runtime doing compilation, respectively. The ratio of real instructions to simulated instructions ranged from only 3.3 to 6.2, and these ratios could be further reduced by removing `shade`'s profiler capabilities.[2]

5.2 Fast Decompression

`Pardoz` is a compression algorithm that trades off compression ratios and compression rates to boost decompression rates. It achieves its fast decompression by using byte-width rather than variable bit-width compression codes. An escape code indicates that the next code appears literally in the output. All other compressed codes are used as indicies into a table of variable length strings that are simply appended to the output. As a baseline, the C implementation of `pardoz` decompresses at roughly four times the rate of `uncompress`, a standard implementation of Lempel-Ziv decompression [Wel84].

The stream of compressed data codes can be viewed as a program for an interpreter with one primitive for each string length and one for the escape. We implemented a threaded-code interpreter for `pardoz` in assembly language, with unrolled loops and hand-scheduled loads and stores. On both SPARC and MIPS-based systems, it was twice as fast as the C implementation.

The RTCG version dynamically compiles the inner loop of the decompressor, using a block of instructions for each code. Output strings are embedded as immediate constants, and blocks of instructions are aligned to speed dispatch. On a Sun 4/490, the RTCG version of `pardoz` runs 40% faster than the fastest static code version. On a DECstation 5000, however, RTCG provides only a 2% performance improvement. We suspect the SPARC is faster because it makes good use of indexed addressing and annulled branches, and because RTCG eliminates relatively slower 4/490 loads from a unified cache.

This example illustrates that RTCG can improve performance, but with very different improvements, even on similar architectures. That in turn underscores the need for a better understanding of which architectural factors are important for good RTCG performance.

5.3 Record Comparison

The last application sorts a collection of fixed-format records. The sort routine takes a set of records and a *sort vector* describing the order in which to compare fields.

A static implementation compares fields in the order specified by the sort vector. Individual field comparisons are fast, so dispatching to the next field to

[2] Simulating a SPARC on a SPARC could give a performance advantage over cross-machine simulation. For example, floating-point operations could be executed using hardware, where they might be simulated in cross-machine simulation. However, g88 makes the same optimizations to get its performance [Bed89].

test is a large overhead. It is impractical to fully-case the code, since a record with F fields can have $F!$ cases. Common-casing can exploit record and sort vector statistics, but is only as good as the statistics. The RTCG alternative is to delay final compilation until the sort vector is known. The RTCG code has no dispatching and removes the indirection of the sort vector. RTCG needs only space for the runtime compiler, plus one instance of the sort routine.

We wrote two RTCG implementations of record sorting. The first builds sort predicates using a template compiler, built by "tricking" a version of the GNU C compiler into producing fragments of position-independent code. The sort vector is dynamically compiled by concatenating the resulting templates. The second implementation translates the sort vector to C code that is compiled using a standard compiler. The resulting object code is then dynamically linked into the sort program.

We ran the static code and two dynamic code versions over a collection of randomly generated records executing on VAX 3500, Motorola 68020, Intel 80386 and MIPS R2000 platforms. Not all implementations worked on all platforms. The results are summarized in Figure 5.3. The breakeven point when using the template compiler is from 100 to 500 sorted records; with the heavyweight compiler, from 20,000-50,000 records. Overall speedups ranged from 10% to 80%. On the MIPS-based platform, the dynamically-linked RTCG version of the program was 30% faster than static code compiled with the same degree of optimization. However, the highest level of optimization (`-j`) cannot be used with dynamically-linked code. Still, the RTCG version at a lower optimization was more than 10% faster than the static code with the best optimization.

This experiment demonstrates that it is possible to do RTCG using existing retargetable compilers, and that (in at least some cases) the retargetable compilers build fast dynamic compilers. In addition, it shows that having more definite information available at runtime can be more beneficial than having a better optimizer.

6 Open Issues

We have argued that runtime code generation is a technology that deserves further study. To advocate it's widespread use would be premature, because it is unclear *a-priori* when it will be profitable, and because useful technologies such as fast retargetable compilers or partial evaluation have yet to be applied. In this section we outline some of the open research issues.

Machine-Dependent Profitability Analysis: Architectural and implementation design choices affect the costs associated with RTCG and thus the point at which it becomes profitable. Timings for many operations are well-reported, but timings for some, such as cache flushing, need to be reported more carefully.

Architectural designs should not unnecessarily penalize RTCG. To pick on a single example, the SPARC Applications Binary Interface (ABI) requires programs to update the instruction space using the `iflush` instruction. Each `iflush` updates only a few bytes and may cause a trap. Performance is generally good on small regions, but there may be thousands or millions of traps when flushing large regions. No architectural change is needed to improve the interface. For example, a better interface would use a system call, which would cause a single trap, independent of the region size.

We described an experiment in §5.2 in which there was substantial performance improvement on the SPARC architecture but a negligible amount on the MIPS. While we can attribute this to specific factors for this specific decompression algorithm (addressing modes, cache organization and access times), general models could determine how these and other architectural factors affect RTCG's profitability.

Machine-Independent Profitability Analysis: The profitability of RTCG depends on the application. For example, our studies show that the performance of NFA regular expression pattern matching is improved using RTCG, but DFA regular expression pattern matching is not; and that the performance of the textbook Boyer-Moore string matching is improved with RTCG, but that the *fast* version is not. We need machine-independent profitability models.

Code Generator Performance: Compiler speed and code quality also affect RTCG's profitability (§4.2). Thus, a complete profitability analysis also requires detailed models for describing code generator performance, both running time and code quality.

Automatic Discovery: If RTCG is used as an optimization technique, it would be best if we could discover opportunities in existing source code automatically. To do so requires a mechanism for discovering candidates, the profitability analysis described above, and a technique for transforming source language constructs into RTCG counterparts.

Specification: In the absence of automatic techniques for discovering RTCG candidates, RTCG hints could be specified directly by the programmer. Some languages can describe the creation of "new code", such as LISP with `eval`, but lack any way of saying whether a static or dynamic code implementation is preferred. Other languages need to be augmented to include constructs for describing the dynamic creation of code.

Debugging: Debugging is beyond the scope of this paper, but we note that some RTCG debugging problems can be removed if the use of RTCG is well-structured and the debugger views it as dynamic change of representation. For example, breakpoints may be set in

Machine	VAX	68000	80386	R2000
Startup: Template Compiler	6.4	5.6	4.38	–
Startup: Generic Compiler	750	–	–	1700
Speedup: Template Compiler	1.4	1.8	1.7	–
Speedup: Generic Compiler	1.4	–	–	1.1/1.3
Template Breakeven (N)	500	500	100	–
Generic Breakeven (N)	20,000	–	–	50,000

Figure 3: Startup Dilation, Asymptotic Speedup, and Breakeven Point for RTCG Record Comparison

nonexistant code by introducing the breakpoint just after the dynamic compiler is run. Alternatively, debugging can use an equivalent, if slower, static-code implementation, Then, the breakpoint can be re-phrased in terms of static code.

7 Summary

Several important factors led to dynamic code techniques being dropped from favor: expanding memories, portability and debugging problems, and a lack of proven performance. However, hardware and software is changing in ways that may improve RTCG's utility: smaller memories are being reintroduced as caches, there is an increasing demand for data-dependent optimizations, there are better abstractions for code generation and modification, partial evaluation may be able to extract specialized compilers automatically, and faster computers make it possible to run more data-intensive applications. Finally, although RTCG's use is deprecated, it continues to be used. This leads us to believe that it *is* sometimes profitable and that what is needed is better models and methodology for determining *when* it is profitable.

In this paper we have analyzed RTCG as an optimization technique. A runtime compiler can generate better code because it has more specific information about the particular execution. In some cases, the dynamically-generated code can be fast enough to pay for the cost of runtime compilation. To support our analysis, we compared static-code and dynamic-code versions of several applications: virtual machines, a data decompression algorithm, and record comparison. Each ran faster, but the performance benefit depended on factors such as machine architecture, compiler technology and workload.

Although modern architectures have "dropped" RTCG, current designs do not need to be changed to support it. The next step is to apply the software technologies and perform further empirical studies. From this experience we can hope to devise models and metrics and determine more concretely and reliably when RTCG is applicable.

8 Acknowledgements

Thanks to Craig Chambers and David Patterson for their help.

References

[Bed89] Robert Bedichek. Some Efficient Architecture Simulation Techniques. *Winter '90 USENIX Conference*, pages 53–63, 26 October, 1989.

[Bed91] Robert Bedichek. Personal communication, October 1991.

[CAK+81] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. Alorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. Griffiths Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost. Support for Repetitive Transactions and Ad Hoc Queries in System R. *ACM Transactions on Databse Systems*, 6(1):70–94, March 1981.

[CK91] Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. Technical Report (internal), Sun Microsystems, March 1991.

[CU91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. *OOPSLA '91 Proceedings; SIGPLAN Notices*, 26(11):1–15, November 1991.

[DM87] David R. Ditzel and Hubert R. McLellan. Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero. *Proceedings of the 14th Annual International Symposium on Computer Architecture; Computer Architecture News*, 15(2):2–9, June 1987.

[DS84] Peter Deutsch and Alan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. *11th Annual Symposium on Principles of Programming Languages (POPL 11)*, pages 297–302, January 1984.

[FH91] Chris W. Fraser and Robert R. Henry. Hard-Coding Bottom-Up Code Generation Tables to Save Time and Space. *Software – Practice and Experience*, 21(1):1–12, January 1991.

[Har77] Anders Haraldsson. A Program Manipulation System Based on Partial Evaluation. Technical Report Linköping Studies in Science and Technology Dissertations No. 14, Department of Mathematics, Linköping University, S-581 83 Linköping, Sweden, 1977.

[Hen87] Robert R. Henry. Code Generation by Table Lookup. Technical Report 87-07-07, University of Washington Computer Science, 1987.

[HO91] W. Wilson Ho and Rondald A Olsson. An Approach to Genuine Dynamic Linking. *Software-Practice and Experience*, 21(4):375–390, April 1991.

[JFnt] Charles Johnson and David L. Fox. The Silicon Palimpset – A Programming Model for Electrically Reconfigurable Processors. *SIGForth (to appear)*, 7 March 1991 preprint.

[Jou90] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Sondergaard. Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation. *Lisp and Symbolic Computation*, 2(9-50):10, 1989.

[Kep91] David Keppel. A Portable Interface for On-The-Fly Instruction Space Modification. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 86–95, April 1991.

[Kes90] Peter Kessler. Fast Breakpoints: Design and Implementation. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation; SIGPLAN Notices*, 25(6):78–84, June 1990.

[Loc87] Bart N. Locanthi. Fast BitBlt With asm() and CPP. *European Unix Users Group Conference Proceedings (EUUG)*, September 1987.

[May87] Cathy May. A Fast S/370 Simulator. *Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques; SIGPLAN Notices*, 22(6):1–13, June 1987.

[MBB+91] T. N. Mudge, R. B. Brown, W. P. Birmingham, J. A. Dykstra, A. I. Kayssi, R. J. Lomax, O. A. Olukotun, K. A. Sakallah, and R. Milano. The Design of a Micro-Supercomputer. *IEEE Computer*, pages 57–64, January 1991.

[McF91] Scott McFarling. Procedure Merging with Instruction Caches. *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI); SIGPLAN Notices*, 26(6):71–79, June 1991.

[Mir87] Eliot Miranda. BrouHaHa - A Portable Smalltalk Interpreter. *OOPSLA 87 Proceedings; SIGPLAN Notices*, 22(12):354–364, December 1987.

[Mir91] Eliot Miranda. Portable Fast Direct Threaded Code. Technical Report USENET comp.compilers article, ID #3035@redstar.cs.qmw.ac.uk, Computer Science Dept, QMW, University of London, UK, 28 March 1991.

[MP89] Henry Massalin and Calton Pu. Threads and Input/Output in the Synthesis Kernel. *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[PHH88] Steven Przybylski, Mark Horowitz, and John Hennessy. Performance Tradeoffs in Cache Design. *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 290–298, May 1988.

[PLR85] Rob Pike, Bart N. Locanthi, and John F. Reiser. Hardware/Software Trade-offs for Bitmap Graphics on the Blit. *Software - Practice and Experience*, 15(2):131–151, February 1985.

[PS84] Lori L. Pollock and Mary Lou Soffa. Incremental Compilation of Locally Optimized Code. *Conference Record of the 12th Annual Symposium on Principles of Programming Languages (POPL 12)*, pages 152–164, 1984.

[SC91] Harold S. Stone and John Cocke. Computer Architecture in the 1990s. *IEEE Computer*, pages 30–38, September 1991.

[sys90] *System V Application Binary Interface*. Prentice-Hall, 1990.

[Tho68] Ken Thompson. Regular Expression Search Algorithm. *Communications of the Association for Computing Machinery (CACM)*, 11(6):419–422, June 1968.

[Wal91] David W. Wall. Predicting Program Behavior Using Real or Estimated Profiles. *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation; SIGPLAN Notices*, 26(6), June 1991.

[Wel84] Terry A. Welch. A Technique for High Performance Data Compression. *IEEE Computer*, 17(6):8–19, June 1984.