Runtime Code Generation

Markus Malmqvist
mmalmqvi@cc.hut.fi

Tero Hasu
thasu@niksula.hut.fi

May 6, 1996

Abstract

Runtime code generation (RTCG) means dynamically adding code to the instruction stream of an executing program. It has been in use for a long time and is a broad subject. In this paper we survey some basic techniques and show how they have been applied in practise.

1 Introduction

Runtime code generation (RTCG) has been in use since the earliest programmable-store computers (in the 1940s). 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.

Changing hardware and software technologies made the use of RTCG less profitable and thus reduced the demand for it. Memories grew, reducing the I/O component of memory access times, which made it possible to statically generate many specialized alternatives of a general procedure. Portability became a major issue and was achieved using high-level languages, most of which lacked ways of expressing the dynamic generation of new code. It was gradually forgotten that A's data is B's code, and that nothing fundamentally prevented A and B from being the same.

Recently, however, on-going changes in hardware technology, software technology and workloads have brought about circumstances which make the use of runtime code generation more profitable.

In Section 2 we describe some of the essential terms of this area, and in Section 3 we speculate on the usefulness of RTCG. In Section 4 we cover some of the related theory. In Section 5 we present some cases in which RTCG has actually been used. Section 6 concludes.

2 Terminology

Runtime code generation is a general name for structured techniques that change a program's instruction space while the program is running. There is a variety of other common, overlapping names.

Dynamic Compilation is a little narrower a term than runtime code generation, as it is typically seen to encompass only generation of code that does not modify itself.

Dynamic Linking adds code to the instruction space of the program. The code is typically generated using a code generator that is outside of the program. The linked code may have been generated on demand for the application.

Runtime Compilation is a term which is often used to describe loop parallelization that is done at runtime and which performs no code generation. Certain loop-carried data dependencies cannot be discovered by a static compiler, and runtime compilation may be beneficial in these cases.

Instruction-Space Modification is a general term, and is used to describe changing bits in an instruction space, regardless of whether the change comes from inside the application or from outside.

Self-Modifying Code is a term whose meaning is hard to pin down, as there are many ways to interpret the word "self". There may be an instruction which modifies:

- itself
- an instruction in the same basic block
- an instruction in the same (source code) module
- an instruction in the same address space
- an instruction in the same protection domain

If "self" is interpreted to mean the instruction space, just adding instructions to the instruction space without reusing any region of memory could also be considered as self-modification.

Deferred Compilation is a lightweight approach to run-time code generation, in which compile-time specialization is employed to reduce the cost of optimizing and generating code at run time [14].

3 Pros and Cons of RTCG

3.1 Reasons to Use RTCG

Times are changing so that RTCG is becoming more useful. RTCG is an increasingly interesting option, because new significant static optimizations are hard to find. In recent years CPU speed has increased quickly while memory access speed has improved only little. This diminishes relative RTCG penalty. Due to growing cache miss penalties, code size is again becoming an important issue. RTCG can adapt to the cache size of the target architecture for example when unrolling loops. RTCG often reduces the size of the code. This can make a code fragment small enough to fit in the cache. Datasets processed by computers become larger and larger. This increases the probability that the cost of runtime compilation is less than the gain acquired from repeated use of optimized code.

3.2 Why RTCG Can Produce More Efficient Code

Optimizing with RTCG can be fruitful, because all information concerning the operation of a program can be used. Only part of this information is available for traditional static optimizers. With RTCG it is therefore possible to create code that beats any statically created code [10].

There are ways to gain similar speedups with static compilation. Code can be made fully *cased*, which means that there is a specialized code fragment for every possible input. Due to excessive space demands this is rarely sensible and often impossible. Another method is common *casing*. Statistic analysis is used to spot the most common input values, for which specialized code is written. However, reliable identification of common cases can be a tough problem. On the other hand, RTCG can produce specialized code for an input when needed. This costs the time spent dynamically compiling the code, which is in many cases acceptable.

Typical optimizations used with RTCG could be loop unrolling, dead code elimination, constant folding and constant inlining. At runtime, cache size can be used when deciding whether to unroll a loop or not. Some dead code can be found only at runtime. Runtime constants can be used in constant folding and they can be inlined to instruction immediates. Thus all these optimizations can be done better at runtime.

Specialized code acquired with RTCG has also typically less decisions and therefore also less branches than the statically optimized code, so instruction prefetch can work more efficiently.

3.3 Problems in Using RTCG

Machine independence is hard to reach, and debugging dynamically changing code can be cumbersome. The performance gain can vary greatly even on similar architectures. Cache architectures differ, so visibility of changes in instruction space vary between hardware implementations. It is also important to know when to stop while partially evaluating a program (see Section 4.1). Otherwise the program might be evaluated fully and nothing is gained. Partial evaluation should consume some well chosen input and produce better code with little work. Maybe the most pressing issue is the lack of guidelines which could be used to deduce whether it is profitable to use RTCG in a certain case or not. Therefore the most difficult, at least partly open issues are both machine-dependent and machine-independent profitability analysis, automatic discovery of places there RTCG could be used for optimization and manual direction of RTCG in languages by programmers. The last one is of course a less tempting alternative to automatic discovery.

Automatic RTCG may under- or overspecialize code [13]. On the other hand, it is not pleasant for the programmer to manually decide in which parts of the program to apply RTCG. Programmer-directed RTCG is typically also less general. Better implementations of automatic discovery of promising targets for RTCG would therefore be the best solution.

3.4 Some Solutions to Problems

Machine-independence could be achieved by using a generic compiler and dynamic linking, but the startup cost would be excessive. There are competitive retargetable code generators that can efficiently produce code of good quality from intermediate representation. The intermediate representation or IR is produced by an application-specific compiler that can be either written by hand or generated automatically via partial evaluation (see Section 4.1). This way machine independence is guaranteed. Differences in cache architectures can be hidden behind the compiler interface and debugging can be managed by making RTCG well-structured enough that the debugger can change representation dynamically.

3.5 A Cost Model for RTCG

One way to model time cost of RTCG is to use a simple linear model. It could be something like: $\text{cost} = \text{startup} + m \times N$ [9]. Startup means the time spent for dynamic compilation, m is time used when running the optimized code fragment once, and N tells how many times the code fragment is ran. Statical cost could be: cost $= ms \times N$, where ms is time needed for one invocation of statically compiled code fragment. Of course the cost of RTCG should be lower than the cost achieved by statical means. Because statically compiled code has no startup cost, but ms is bigger than m, RTCG will become more profitable as N becomes larger. The value of N at which the costs are same, is called the breakeven point.

While optimization level increases, startup cost and breakeven point are increased but m is decreased. This implies that for large datasets it is worthwhile to use complex optimizations. Of course, if an optimization takes a long time and still provides only modest speedup, an enormous data set is needed before this kind of optimization is sensible.

3.6 An Example

`Bitblt` is a routine that has been optimized with RTCG by many. It is a graphical bit-transfer operation which merges a source rectangle with a destination rectangle, using some logic operation such as `and`, `or`, `xor`. It is general, handling different alignments, sizes and overlapping rectangles.

The outer loop executes once per horizontal line in source rectangle. The inner loop reads the source one machine word at a time. Due to generality, the inner loop is complicated and depends on numerous parameters. Many of them change rarely. *Fully-cased* `bitblt` can be extremely long, even 1MB. *Common-casing* is hard to implement, because change in one parameter can affect many optimizations.

With RTCG, a custom inner loop is generated on demand, one for every call to `bitblt`. Some optimizations, that are not possible statically, can be made also. Included below is a prototypical bitblt inner loop [9]. `Op` is constant over entire call while `ls` and `rs` are constants over the inner loop.

```
j = dst.left;
mask = lmask;
a = src[src.left];
for (i = src.left+1; i < src.right; i++) {
  b = src[i];
  m = (a << ls) | (b >> rs);
  switch (op) {
    case AND:
      dst[j]=dst[j]^((~m&dst[j])&mask);
      break;
    case NOT:
      dst[j]=dst[j]^mask;
      break;
```

```
    case OR:
      dst[j]=dst[j]|(m&mask);
      break;
  ... 13 more cases ...
  }
  j = j + 1;
  ... end-of-line tests
}
```

4 Techniques in Using RTCG

4.1 Partial Evaluation

Partial evaluation means evaluation of a program with respect to some part of the program's input. As the result, a residual program is obtained. This residual program executes the computations which depend on the input not used in partial evaluation. The partial evaluator is traditionally often called `mix`. The equation `[p](d1, d2) = [[mix] (p, d1)] d2` [13] means that the result of evaluation of program p with arguments d1 and d2 must be the same as the result obtained from first applying partial evaluation to argument d1 and then giving the argument d2 to the residual program.

The most interesting aspect is that `mix` can be applied to itself. The result is a program that will generate the residual program when ran. Upon further self-application a stand-alone code generator is obtained. This could be used to optimize any program in respect to any input. However, RTCG is often quite specialized and might therefore not directly rely on theory of partial evaluation. Pure partial evaluation can also be quite slow. Therefore code generators are usually made by hand. This hand-written code generator can then be used to specialize itself to a certain procedure to obtain a special runtime code generator for the procedure.

4.2 Runtime Compilers

Because runtime compilers are application-specific, they can be much faster than generic compilers. It may well be that entire compilation phases can be left out while other phases are optimized for the application. As a result, startup cost decreases. It is important that only those optimizations are executed, which pay back the time spent. Speedups gained with RTCG are data-dependent, but variations are typically small in practise.

While machine-independence is usually strongly recommended, sometimes it may be suitable to specialize RTCG not only for a certain application but also for a certain machine for maximum speed. In these cases, a *template compiler* can be used. Templates are machine code sequences with holes. The template compiler fills these holes with runtime values and concatenates templates accordingly. This results to very fast one-pass compilation that can often rival even optimized, hand-crafted assembly. However, the use of template compilers is a very laborious process. Detailed optimizations must be done by hand. In the past, RTCG has mostly been used this way.

A more recommendable way is to use a compiler which produces code in machine-independent intermediate representation (IR). An IR compiler is still application-specific. A retargetable code generator can be used to transform the IR into native machine language of the target machine. Intermediate representation can be easily optimized automatically. Also, the programmer is free from writing machine code. Templates can be used to speed up IR compilation, because the code produced has a very similar structure in every invocation. Sometimes even code produced via machine-independent RTCG can be better than hand-crafted assembly.

Of course, an IR compiler is slower than a template compiler due to its more generic nature. A template compiler can make at compile time some optimizations which an IR compiler must make at runtime. An IR compiler can manipulate larger blocks than a template compiler while optimizing, but this can produce only a minor speedup. The retargetable code generator can be quite simple, because most useful optimizations are made in IR compiler after promoting the runtime constants to immediates.

Machine-dependent RTCG doesn't necessarily use templates. This can make RTCG more general. RTCG can generate machine code that will in turn produce optimized code at runtime. RTCG made this way can also be faster, because emitting instructions is faster than handling templates. However, one pass code generation can be hard to do well.

4.3 Code Analysis

One analysis method is to divide the computation made by a program into stages [13]. Values computed in earlier stages change more slowly than those computed in later stages. Stages are analyzed to determine which of them should be optimized. If a subexpression contains only values that belong to the current stage or an earlier stage, the result will belong to the current stage. In practise, this process must be approximated due to recursion. This makes RTCG less optimized and therefore making the analysis more accurate is a task of high priority.

While early computations are compiled normally, the instructions concerning the late computations, which depend only on values defined in a earlier stage, are emitted at runtime. Only emitted instructions will belong to final runtime-optimized program. There can be many little code generators in the code that generates runtime-optimized code, each producing code for a small part of the source program. Actually, if more than two stages are recognized, the little code generators will produce other code generators and so on. This is desirable, because programs usually contain many stages. With cataloguing the little code generators, generation of duplicate code can be avoided.

In general, when optimizing RTCG it is crucial to investigate which optimizations will probably work so much more efficiently at runtime that they will pay back the time invested.

5 Existing Systems

We have selected some existing applications for presentation in this chapter, and attempt to describe them in some detail.

5.1 `dcg`

`dcg` [6] is a system that allows clients to specify dynamically generated code in a machine-independent manner. Code generation costs approximately 350 instructions per generated instruction.

`dcg` client programs are, in essence, small compiler front-ends, because they specify dynamically generated code using the machine-independent intermediate representation (IR) of the `lcc` compiler [7]. The IR, while smaller and simpler than `gcc`'s, for example, specifies a rich enough set of operators that all C language constructs can be expressed.

`dcg` consists of a small library of interface routines which simplify the creation of an `lcc` IR. Once the client has, using these routines, created an IR specification for a single procedure (which is the unit of code generation for `dcg`), the specification can be passed on to `dcg` for compilation. `dcg` compiles the procedure and returns a pointer to the executable code. The client invokes that code as an indirect call to a C procedure.

`dcg` does binary code selection using `burg` [8], which uses `Bottom-Up Rewrite System` (`BURS`) technology to optimally translate an IR tree into machine instructions [17].

Retargeting `dcg` for a new target machine is done in three parts. First, a mapping from IR patterns to machine instructions is created. Second, machine instructions are mapped to binary templates. Third, auxiliary code for observing calling conventions, data layout restrictions, register allocation, etc. is defined. Small languages for expressing the mappings are provided.

For more information about the library routines, etc., see [6]. Figures 1 and 2, also taken from [6], show a piece of code that builds a function using `dcg`, and the code generated by `dcg`, respectively.

```
typedef int (*FPtr)(int);

FPtr example() {
    Symbol arg[2];      /* argument vec sent to gen */
    int ncalls = 0;     /* number of calls made by plus1 */

    arg[0] = sargi();   /* allocate symbol for 'x' */
    /* associate with a virtual register (if possible) */
    dcg_param_alloc(arg, ncalls);

    /* create and register IR tree for "return x + 1;" with dcg */
    regtree(reti(addi(indiri(addrfp(arg[0])), cnsti(scnsti(1)))));

    /* generate code on heap */
    return (FPtr)dcg_gen(sfunc("plus1"), arg, ncalls);
}
```

Figure 1: *Routine to build function "int plus1(int x) { return x + 1; }" dynamically*

```
addiu $sp, -152         # allocate AR
add $25, $4, 1          # ADDI ($4 holds argument 1)
move $2, $25            # RETI ($2 holds return value)
addiu $sp, 152
j $31
```

Figure 2: *The R3000 code emitted to compute "return x + 1;"*

5.2 'C

From the work with `dcg` grew out 'C (Tick C) [5], a superset of ANSI C, which offers several improvements over `dcg`. Perhaps the most significant improvement is the interface for code specification, which is high-level instead of being based on an intermediate representation of a compiler. 'C, like `dcg`, allows machine-independent specification of dynamically generated code.

In ‘C dynamic code is specified at runtime, and these specifications can then either be composed to build larger specifications, or instantiated to produce executable code. To provide support for specifying dynamic code, ‘C adds two type constructors and three unary operators to ANSI C. The two new type constructors, `cspec` and `vspec`, are both postfix-declared types (similar to pointers). The three new unary operators, `'`, `@`, and `$`, have the same precedence as the standard unary prefix operators.

The `'` operator specifies that its operand (an expression or a compound statement) is to be dynamically generated. Nested backquote expressions are forbidden, so it is not possible to specify dynamic code which specifies dynamic code. The use of some of the C constructs, such as `break`, `continue`, `case`, or `goto`, is restricted. In particular, the mentioned statements cannot be used to transfer control outside the containing backquote expression.

The type of a dynamic code specification is `cspec` (code specification). The evaluation type of the code specification is the type of the dynamic value of the code. For example, the type of the expression `'4` is `int cspec`. Applying `'` to a compound statement yields a result of type `void cspec`. The statical typing of dynamic code specifications allows the compiler to type-check code composition statically.

Variables with a `vspec` (variable specification) type are dynamically generated lvalues. `vspec`s allow the construction of functions that take a runtime-determined number of arguments.

The `@` operator allows dynamic code specifications to be combined into larger specifications. Its legal operands are `cspec`s and `vspec`s, and it must be applied inside a backquote expression.

The `$` operator causes its operand to be evaluated at specification time. It may only be applied to an expression within dynamic code, and the expression must not have `cspec` or `vspec` type. The resulting value is incorporated as a runtime constant into the containing `cspec`.

The functions provided by the ‘C standard library include:

`compile` Compiles `cspec`s.

`param` Creates a parameter for the function under construction.

`local` Used to reserve space in a function’s activation record.

Here is an example, a piece of ‘C code which compiles and calls a function that prints "Hello world".

```
void cspec hello = '{ printf("Hello"); };
```

```
void cspec world = '{ printf(" world"); };

/* Concatenate the two statements. */
void cspec helloworld = '{ @hello; @world; };

/* Compile and call the result. The TC_V indicates that the return
   type is void. */
compile(helloworld, TC_V)();
```

'C also supports partial evaluation. This doesn't actually change the power of the language, but it may be useful. Because code generation of all templates happens at compile time, partial signatures are used to indicate which arguments can be specialized in a function; this is done to avoid code explosion. A partial signature is a function prototype prefixed with the `partial` keyword. Type specifier `bound` is used to indicate bound arguments. Partial evaluation of a function is performed using the unary prefix operator `eval`. Keyword `unbound` is used in place of arguments which are not being provided during partial evaluation.

While the 'C compiler prototype was based on `dcg`, a more recent implementation, called `tcc`, exists [16]. It uses templates and is based on `VCODE`, which is a dynamic code generation system [4]. In that implementation, backquoted code is compiled into C code that invokes `VCODE` macros to emit code directly, without processing any intermediate representation at run time.

5.3 Fabius

Like the 'C compiler mentioned earlier, `Fabius` also is a compiler which can be used to produce code which will generate code at run time [11]. The major difference between the two compilers is that `Fabius` automatically compiles ordinary programs, written in a subset of ML, into code that does RTCG. It discovers and exploits opportunities for dynamic optimization by itself.

One of the most important goals in developing `Fabius` has been the reduction of the run-time cost of code generation to minimum. The approach to compilation that `Fabius` uses is called deferred compilation. `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. The creators of `Fabius` report that the average cost of RTCG is as low as about six instructions executed per generated instruction. The partial evaluation techniques and heuristics used include memoization of specialization, unfolding of state conditionals, and the residualization of static values in dynamic contexts.

When `Fabius` begins compilation, it must first 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 isn't easy. When a chosen function is applied to an early argument, a code generator is invoked to create a specialized function that is parameterized by the remaining "late" arguments. The function is first annotated with early and late annotations. It is then compiled into a register-transfer language, which also contains the annotations. The result, in turn, is translated into annotated MIPS assembly code. Now, the annotations indicate whether the instructions should simply be executed (early), or emitted into a dynamic code segment (late).

`Fabius` adds initialization code to the compiled program to allocate a dynamic code segment; one register, called the code pointer, is dedicated to keeping track of the position where the next instruction to be emitted is placed.

`Fabius` uses run-time instruction selection, an optimization that may in some cases be very beneficial. For more information about `Fabius`, see [11].

5.4 C

A similar approach for run-time specialization has also been applied to the C programming language [3]. Because of unrestricted aliasing and mutable data structures, C is perhaps not as well suited for this approach as ML. Program analyses have to be overly conservative, and opportunities for optimization are thus limited. ‘C, for example, doesn't have such a problem, because of the programmer's responsibility to specify manually where and how to do specialization.

A C language implementation has been done, using `gcc`, and is integrated in a partial evaluation system that specializes programs at compile time as well as run time. At compile time the system produces templates and transforms them so that they can be processed by a standard compiler. At run time only minor operations, such as selecting and copying templates, need to be performed. This leads to good performance.

5.5 SPIN

5.5.1 Introduction

General operating systems often work very poorly with certain types of applications, because they must be capable to run all applications. Usually, they are not modular enough to allow comfortable specialization and extension. If they are, performance might be seriously compromised when using extensions. Specialization might also hurt other applications.

Protected cross-domain communication is very expensive on many systems.

This can be overcome by allowing dynamic linking of code into the kernel. If protection is not addressed properly, runtime core extension can be very risky, because it affects all applications. Therefore extensions should be made application specific.

`SPIN` offers extensibility with both safety and good performance [1]. Extensions are dynamically linked into the kernel's address space, which allows fast communication between kernel and extensions. The linker is also in the core. Extensions can be loaded at any time. Extensions and kernel code is written with Modula-3 which guarantees module isolation and therefore security.

5.5.2 Implementation

An extension provides a collection of handlers and maps them to events. Both events and handlers are procedures, event procedures are exported from some interface and handler procedures of an event procedure are those which have the same type as the event procedure. The right to raise an event is equivalent to the right to call the event procedure which in turn is equivalent to the right to reference the type name of the event procedure. Event procedure contains default handler for the event. If there are no other handlers for the event, event raising is equivalent to calling the event procedure. Otherwise the event is captured by the dispatcher which dynamically optimizes the calling of handlers.

`SPIN` provides a set of fine-grained services for memory and processor resources management. These are used by the incore extensions. Because the extensions can use core services efficiently, very low-level services can be provided without hurting performance. In fact, there is no reason to create higher level abstractions via concatenation of low-level services, they can be conveniently constructed by simply calling the low-level services.

5.6 `Shade`

5.6.1 Introduction

A tracing tool should be independent from language and compiler. It should be able to trace also those applications that employ signals, exceptions and dynamically-linked libraries. Tool should be so fast that realistic workloads can be studied. It should also be extensible so that it can be used to collect any retrievable information. Simulation of non-existent machines should also be possible.

Of course, these are partly conflicting features. For example added flexibility typically hurts performance. `Shade` uses several methods to overcome this fact [2]. Traced executeable code is dynamically cross-compiled into executeable code for the host machine. Compiled code is cached to make dynamic compilation to pay off. Simulation and tracing is integrated so that trace information can be saved directly during simulations. `Shade` can adapt to different tracing requirements collecting only the requested information. This enhances performance. A client can supply special purpose code to enable collection of data not supported directly by `Shade`.

5.6.2 Implementation

A small main loop maps each simulated target instruction to a corresponding fragment of host code, called a translation. Each target instruction is dynamically cross-compiled into a translation which will simulate the instruction and save any requested trace information. Target applications memory and register references are mapped to host memory and registers. One translation often simulates several target instructions. Some special instructions such as branch and traps force `Shade` to move to next translation. Compiled translations are cached and translations can be chained so that returning to main loop between translations is not usually necessary.

6 Conclusion

It seems likely that the advances in hardware and software will continue the trends outlined in this article. This will strengthen RTCG as an option for more public use. It will be a long time before RTCG is commonly used, however.

The Web is highly distributed and therefore methods to cross-compile machine-independent source code are needed. RTCG could be used to avoid unnecessary code generation via dividing monolithic applications. When compiling source code to machine-independent code, staging analysis or similar methods could be used to prepare the application for the use of RTCG. This way only part of the code must be compiled into native code before starting the application. The rest of the code could be compiled on demand at runtime. This would enhance response time.

7 Acknowledgements

[9] proved most helpful when constructing this paper, thanks to its authors.

References

[1] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Backer, C. Chambers, and S. Eggers Extensibility, safety and performance in the SPIN operating system. University of Washington, Department of Computer Science and Engineering, 1995.

[2] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. 1994.

[3] C. Consel and F. Noel. A general approach to run-time specialization and its application to C. *In POPL '96 Symposium on Principles of Programming Languages*, pp. 145-156, January 1996.

[4] D. R. Engler. VCODE: a very fast, retargetable, and extensible dynamic code generation substrate. *Technical Memorandum MIT/LCS/TM534, MIT*, July 1995.

[5] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. *In Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 131-144, January 1996.

[6] D. R. Engler and T. A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. *Sixth International Conference on Architecture Support for Programming Languages and Operating Systems*, pp. 263-272, October 1994.

[7] C. W. Fraser and D. R. Hanson. A code generation interface for ANSI C. *Software - Practice and Experience*, **21**:9, pp. 963-988, September 1991.

[8] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG - fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, **27**:4, pp. 68-76, April 1991.

[9] D. Keppel, S. J. Eggers, and R. R. Henry. A case for runtime code generation. *Technical Report UWCSE 91-11-04, University of Washington, Department of Computer Science and Engineering*, November 1991.

[10] D. Keppel, S. J. Eggers, and R. R. Henry. Evaluating runtime-compiled value-specific optimizations. *Technical Report UWCSE 93-11-02, University of Washington, Department of Computer Science and Engineering*, November 1993.

[11] P. Lee and M. Leone. Optimizing ML with run-time code generation. *In Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, May 1996. To appear. A preliminary version is available as Technical Report CMU-CS-95-205, School of Computer Science, Carnegie Mellon University, December 1995.

[12] M. Leone and P. Lee. A declarative approach to run-time code generation. School of Computer Science, Carnegie Mellon University.

[13] M. Leone and P. Lee. Deferred compilation: The automation of run-time code generation. *Technical Report CMU-CS-93-225, School of Computer Science, Carnegie Mellon University*, December 1993.

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

[15] D. Pardo. Runtime Code Generation (RTCG). `http://www.cs.washington.edu/homes/pardo/rtcg.d/index.html`.

[16] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A template-based compiler for 'C. *In proceedings of the First Workshop on Compiler Support for System Software*, February 1996.

[17] T. A. Proebsting. Simple and efficient BURS table generation. *In Proveedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, June 1992.