QuaC: Binary Optimization for Fast Runtime Code Generation in C (EXTENDED ABSTRACT)

Curtis Yarvin *Adam Sah*

Abstract

Runtime code generation (RTCG) has considerable theoretical potential but has so far seen little use in practice. Adequate tools are lacking. We present QuaC, an RTCG system that lets C programmers specialize their functions at runtime with a simple, portable user interface. QuaC works by applying compiler optimization techniques to machine code in memory. It is fast and highly retargetable.

1 Introduction

In theory, runtime code generation is an extremely powerful technique. Code runs faster when specialized to the current data; and often at runtime data stays constant long enough to be worth the investment of specialized code. Routines specialized to runtime constants will be smaller and faster.

This is not a new idea, and in the past many systems have explored the field. Before compilers or an emphasis on portability became widespread, most software was written in assembly language, and programmers found it natural and common to use self-modifying code. With the advent of high-level languages, self-modifying code and other forms of RTCG largely disappeared, because HLLs provide no obvious mechanism to support it; but experiments continued in dedicated systems [[PLR85], [May87]] and compilers [[PS84], [CU91]], which could afford to be machine-dependent.

One of the most interesting and spectacular uses of RTCG in recent years, and the one which sparked our interest in the field, is Massalin's Synthesis operating system [[MP89]]. Massalin noted that much kernel data stays constant for long periods of time; his system generates specialized code for operations like process table traversal and open file access, and achieves speedups of up to 50x over comparable Unix systems.

But the techniques in Synthesis, like previous attempts at systematic RTCG, have not been widely adopted. We see several reasons for this:

- Portability. Fast RTCG systems must interact directly with assembly or machine code; as such, their implementation is unportable. Synthesis, for example, is tied closely to the MC68k architecture.
- Interface. Modern software systems are written in high-level languages. We know of no HLLs designed for user-directed runtime specialization. RTCG interface designers must modify their users' HLL, export a machine-specific interface, or create a third language to define specializable code. None of these alternatives is desirable, but most designers choose the second. Synthesis, like many systems, provides assembly-language *templates*, fragments of code containing *holes* to be filled in at runtime. This is efficient, but it makes the entire system unportable and forces its programmer to work closely with assembly language.
- Performance. Some systems forego any attempt at an interface for specialization, and generate full HLL code by hand at runtime. A good example is Christopher's Asgard system [Chr94], which needs to repeatedly evaluate complex algebraic expressions defined at runtime; it converts them into C, sends them to the compiler, and dynamically loads the resulting object module. This process produces good code, but causes performance problems. The generated code must be highly persistent and often-executed to amortize the high cost of compilation. And for most problems it is difficult to generate C code from scratch as a string.

Thus it is unsurprising that previous attempts at popularizing RTCG have not succeeded. The technique is theoretically promising, but practical concerns have outweighed its value.

Our goal in designing QuaC was to overcome some of these problems. We wanted our RTCG system to:

- Provide a simple, C-based interface.
- Allow efficient, portable implementation.
- Require no modification of the C compiler.

We have had to compromise on some of these goals, but overall we feel we have achieved them.

2 The QuaC Interface

QuaC's interface resembles Synthesis' template system. Instead of assembly blocks, however, QuaC templates are C functions. This obviates the need for a special template compiler. Any function in the program can be used as a template.

To specialize, we *curry* a function, defining some of its arguments to be constant. If the program contains the function

```
int read(int fd, char buf, int len);
```

and one invokes QuaC with

```
fd = 2;
errorRead = synth read(fd, *, *);
```

`errorRead` will be of the type

```
int (*errorRead)(char *buf, int len);
```

and will perform the operation `read(2, buf, len)`.

`errorRead()` will be specialized for the case `fd=2`, and will run faster than `read()`; for example, it might write directly to a tty buffer without file table indirection.

Another way to define data for specialization is `fix`. The program uses `fix` to tell the QuaC runtime system that the contents of an area of memory are now constant:

```
fix object length;
```

`fix` lets the runtime optimizer replace memory loads with immediate constants, providing a further source of known data for optimization. For example, an expression tree evaluator might `fix` a tree in memory and then `synth` the interpreter function to it.

3 Binary Optimization

QuaC would be easy to implement with compiler support, as a language extension to C. We would add the `fix` and `synth` keywords to the parser, and modify the output generator to include information for specialization (such as a copy of the intermediate representation for template functions).

Such an implementation would be straightforward. [KEH93] describes an RTCG system that works this way. With the template's intermediate representation stored in the executable, the runtime system needs be little more than the compiler's optimizer and code generator.

We chose to avoid this approach, for several reasons:

- It is little better than just running the compiler. We would be including most of a full compiler in the runtime system, and invoking most of the full compiler's overhead.
- By operating at the machine-code level, we gain flexibility. We can specialize functions from system libraries, functions written in different languages, even functions hand-coded in assembler.
- Most important, we do not want to modify the compiler. Few commercial applications have source code to their C compiler, or want to switch to a free compiler. Compiler writers are unlikely to add support until they see users; users are unlikely to consider the system until they see working tools.

3.1 Optimization of Machine Code

Therefore we chose to implement QuaC as a library. We read the template functions directly from text memory as machine code, and run traditional optimization algorithms such as constant propagation on that.

Machine code, however, lacks some of the information available to a compile-time optimizer. For dataflow analysis, we need to restore three critical structures:

- exactly what variables each instruction affects
- the control flow graph
- procedure call parameters

This is a significant problem. We solve most of it; our solutions are not perfect, but by and large they work. The resulting binary optimizer is not appreciably inferior to a compiler-based optimizer.

3.1.1 Hints

QuaC is not an automatic optimizer; all specialization is user-directed. Therefore we will feel comfortable asking the user for hints.

We impose two restrictions on the use of hints: that the optimizer must operate correctly in their absence, and that all hints must be generatable by a source preprocessor outside the compiler. Our hint mechanism is flag functions whose call the optimizer detects and removes.

Hints do not solve all our problems, and they are not our only solution; but they will help.

3.2 Restoring Instruction Effects

A compiler-based optimizer's intermediate instructions are similar to machine instructions, but operate on logical variables. A binary optimizer sees operations on registers and memory.

Register-to-register instructions are as amenable to dataflow analysis as logical instructions, but memory operations are not. An indirection on an unknown pointer might be a reference to any object in memory. This will disrupt stack dataflow analysis in functions with automatic arrays, and functions which pass addresses of stack variables as arguments.

We have two solutions to this problem. The first is suggesting that template functions not make such use of the stack. It is rarely impossible to code around indirect stack references. The second is range analysis, which will eliminate many of the problems with automatic arrays.

3.3 Restoring the Control Flow Graph

If all the function's branches are direct, restoring the CFG is easy. Indirect branches, which will occur wherever the compiler translates a case statement into a jump table, are more difficult to handle. We have four strategies:

- Reversion. If we simply copy the template's branch instruction, it will branch through the original jump table and take us back into the template function. This ensures correctness, but after a reverting branch we are back in unspecialized code, and we cannot perform optimizations like register reallocation before the branch.
- Range analysis. We can retain reversion as the general case, but most (and probably all) indirect branches are jump tables derived from case statements. To use a jump table, the compiler must be sure that the index variable is within the bounds of the table; thus it must insert range checks. A full range analysis will find the checks and tell us the bounds of the jump table, which we can then parse into the CFG. The only time this will fail is when the compiler's analysis beats ours; it might, for example, assure itself of the index variable's range through interprocedural analysis, which we do not do. We solve this problem with a hint:

  ```
  #define Switch(x)      switch(Qfunk(x))
  ```

  where Qfunk is some unanalyzable external function whose call the QuaC optimizer recognizes and deletes.
- 'Skank-and-clip.' When the compiler's analysis beats ours, a range analysis hint will introduce unnecessary range checks. The cost of this is miniscule and we consider it acceptable, but a perfectly-efficient solution would be better. We can achieve this by guessing the boundaries of the jump table; if we guess an table entry which is not in fact valid, our only loss will be unnecessary parsing, and if we guess that a valid jump index is invalid, we will fill in that entry in our new jump table with a reference to a function of ours, which when invoked saves the processor state and regenerates the function with the new CFG path. This is slow but may be valuable for frequently-executed routines, because it converges on optimal code. For extremely large or offset jump tables we need to place the jump table in an anonymous section of memory [YBA93] and regenerate on page faults; this may require OS support.

3.4 Restoring Function Call Information

QuaC's analysis is intraprocedural. We need to be able to recognize function calls and work around them. To do this we require that the code we analyze follow an Algol-style function model, with a frame accessible only within the activation; and we have to know the frame access protocol, eg, stack pointer. We know of no C compilers for which this poses a problem.

It is also useful, though not strictly necessary, to know what local registers or stack values may be used in the called function. For this we need to know the calling convention and the number of arguments passed; for the latter we take a hint.

3.5 Adaptable Binaries

Future compilers may provide program structure information such as that from Wahbe and Lucco's ABS [WLG94]. In that case, hints and range analysis are unnecessary.

4 Choosing Optimizations

If there are no unknown stack indirections we have the same information as a compiler-based optimizer and can perform the same transformations. In practice it is likely that the user wants fast synthesis, and we restrict ourselves to a few basic optimizations: constant propagation, dead code elimination with def-use chains; optionally value numbering and register reallocation; directed by hints, loop unrolling and inlining.

5 Portability

QuaC is not trivially portable; but it is highly retargetable, more so than most retargetable compilers.

The dataflow analyzer views the processor as an abstract state machine with some number of registers and a flat memory; code is a stream of abstract instructions. In iterative analysis it takes each instruction and determines its sources and effects.

The machine-dependent module needs no semantic understanding of most instructions; all it must know is how to parse out the opcode, and source and destination operands. To map the source values onto the results it jumps to a canon of the instruction and then copies the effects out of the canonical destinations.

Optimization demands more knowledge of the instruction set; we need rules for improving instructions based on knowledge of constant data. This can be coded ad hoc or done with a rule engine as in `lcc` [FH91].

This approach maps well onto all RISC and CISC architectures we know of, including those with delay slots.

6 Implementation

We are in the early stages of implementation on QuaC. Our prototype system does dataflow analysis, all our standard optimizations, and none of the optional ones. It runs on the DEC Alpha architecture and took three months to implement, by two programmers on quarter-time.

7 Performance

We would have liked to modify a large software system, ideally an operating system, to use QuaC; but we lacked the time or resources. In smaller systems, the best use of RTCG is in interpreters. With a loop unrolling hint, QuaC can take a block of straight-line source and an interpreting function, and generate near compiler-quality code.

For a test system we built a very simple integer expression calculator. We read in an expression tree, lay it out postorder in a buffer, and call `fix` on the buffer; then we `synth` the expression evaluator on the buffer and its length.

We have not had time to performance-test this practical example. We have performed microbenchmarks of code generation speed, but the system which we microbenchmarked was straight from debugging phase and had not yet been tuned. The final draft will contain full performance results.

All our algorithms are linear. On a fifteen-instruction, three-argument toy function with one loop, regenerating for two constant words took 750μs on a 133 Mhz Alpha. We consider this excessive. It is largely due to the untuned nature of the instruction resolver, which spent 20μs per instruction, and the reassembler, which took 90μs; neither of these is intrinsically expensive, and our implementations of both contain extensive debugging code.

8 Related Work

Runtime code generation is an old technique; its uses are too numerous to cite. The most similar RTCG system we know of is Keppel's [KEH93], which uses intermediate-code templates and a compiler backend.

Binary code analysis was pioneered in profiling systems, such as `pixie` [Dig]; but the analysis techniques needed for profiling differ from those for optimization.

Currying and partial evaluation are old techniques from the programming language community; they are usually applied in higher-level languages than C.

9 Conclusion

QuaC is hardly a finished system. It is a prototype, and has yet to stand the test of porting. We believe the concepts behind it are useful, and hope that RTCG systems will be more common in the future.

References

[Chr94] Wayne Christopher. *Multiple-Representation Editing of Physically-Based 3D Animation.* PhD thesis, University of California, Berkeley, January 1994.

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

[Dig] Digital Equipment Corporation. *Ultrix v4.2* `pixie` *Manual Page*.

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

[KEH93] David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating Runtime-Compiled Value-Specific Optimizations. Technical Report 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[May87] Cathy May. A Fast S/370 Simulator. *ACM SIGPLAN Notices*, 22(6):1–13, 1987.

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

[PLR85] Rob Pike, Bart N. Locanthi, and J. F. Reiser. Hardware/Software Tradeoffs for Bitmap Graphics on the Blit. *Software—Practice and Experience*, 16(2):131–151, February 1985.

[PS84] Lori L. Pollock and Mary Lou Soffa. Incremental Compilation of Locally Optimized Code. In *Proceedings of the ACM SIGPLAN '84 Conference on Programming Language Design and Implementation*, pages 152–164, 1984.

[WLG94] Robert Wahbe, Steven Lucco, and Susan L. Graham. Adaptable Binary Programs, 1994. Submitted to PLDI '94.

[YBA93] Curtis Yarvin, Richard Bukowski, and Thomas Anderson. Anonymous RPC: Low-Latency Protection in a 64-Bit Address Space. In *Proceedings of the 1993 Summer USENIX Conference*, 1993.