ccg: a tool for writing dynamic code generators

Ian Piumarta
Laboratoire d'informatique de l'université de Paris VI (LIP6)

mailto:ian.piumarta@inria.fr
http://www-sor.inria.fr/~piumarta

OOPSLA'99 workshop on simplicity, performance and portability in virtual machine design

---

Abstract

"ccg" is a tool for writing dynamic code generators in C and C++. It allows efficient dynamic code generation for PowerPC, Sparc and Pentium to be embedded in arbitrary C programs. Dynamically generated code is specified using the standard assembler syntax of the target platform, with extensions to allow C expressions to appear in operands. The program can therefore "specialise" all apsects of the generated code at runtime: literal operands, register selection, jump/branch destinations, elements of complex addressing modes, and so on.

Introduction

Modern virtual machines often rely on dynamic native code generation to achieve good performance. Writing a dynamic code generator is a complex task, and solutions are often ad–hoc due to the lack of proper support tools. ccg is a dynamic code generator generator that relieves the VM implementor of many of the unpleasant tasks associated with dynamic code generation, and which can be used to create code generators of varying complexity: from simple template expansion to more complex optimising code generation. ccg supports the complete specialisation of native code fragments based on values computed at runtime. Special care has been taken to ensure that the code generators that it produces have very low overhead, typically two or three instructions executed for each instruction generated dynamically at runtime.

Using ccg to build dynamic code generators

ccg has two main components: a target–independent source–to–source *preprocessor* and a target–dependent *runtime assembler*.

The preprocessor

The preprocessor takes a C (or C++) source file as input and copies the majority of it unchanged to the output. The preprocessor recognises several **directives** that look

similar to cpp directives. Among these is a directive to select the target processor of the dynamic code generator, for example:

```
#cpu pentium
```

This is directive has two effects. First, it selects the assembler syntax and instruction set that the preprocessor recognises within dynamic code sections (see below). This allows code for each target to use the "native" syntactic conventions of its assembly language, and permits the preprocessor to reject illegal instructions and addressing modes before running the C compiler (thereby avoiding confusing complaints from the compiler about undefined functions resulting from illegal assembler statements). Second, it is replaced in the output file by a `#include` directive to read the cpp macro definitions for the corresponding runtime assembler.

The preprocessor also detects **dynamic code sections** within the source file. These are sections of assembly language for the selected target CPU, delimited by `#[` and `]#`. For example:

```
#[ movl $42, %eax ]#
```

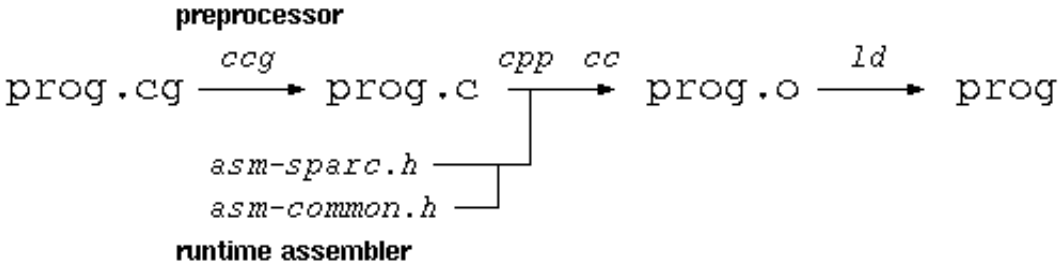
These are replaced in the output file by calls to the runtime assembler macros that will generate dynamic code corresponding to the assembly language statements in the dynamic code section. For example, the above dynamic code section is replaced by the following macro call in the output file:

```
{ _MOVLir(42, _EAX); }
```

The preprocessor is very careful to maintain the correspondance between input and output line numbers, so that any error messages from cpp or cc can easily be related to the original input file.

It is essential to understand the difference between dynamic code sections and "asm" statements. Dynamic code sections are *not* replaced by the assembler statements that they contain (as would be the case with an asm statement), but rather by C code that *generates* the assembly instructions in the dynamic code section at *runtime*.

The following figure shows the steps and files involved in compiling a dynamic code generator, in this case for the Sparc processor:



The runtime assembler

As mentioned earlier, the runtime assembler is composed mainly of a set of cpp macro definitions that implement an assembler for the selected target architecture. The runtime assembler also defines a couple of external variables of which the most important is asm_pc, the address at which the next dynamically–generated instruction will be written. The preprocessor recognises several assembly language "pseudo–ops", including

```
.org anExpression
```

which is replaced by an assignment of *anExpression* to asm_pc.

Continuing with the above example, the macro call

```
_MOVLir(42, _EAX);
```

is expanded by cpp (through several levels of intermediate macros) into the C code to generate the corresponding instruction:

```
((*(u_int8_t*)asm_pc)++)=((u_int8_t)((0xb8|(0x40&0x7))&0xff)),
((*((u_int32_t*)asm_pc)++)=((u_int32_t)((42))));
```

which in turn is easily optimised by any decent Pentium C compiler into something similar to:

```
movb    $0xb8, (%esi)
incl    %esi
movl    $42, (%esi)
leal    4(%esi), %esi
```

Operands can be specialised at runtime with computed values. For example, both the literal source and register destination in the above dynamic instruction could be specified as expressions based on values contained in C variables. The generated code would still contain "fixed" source and destination operands, but these would be recomputed each time the dynamic code section is executed to generate new code. Such specialisation is possible for all the elements found in operands: literals, register "numbers", branch destinations, indices and offsets in complex addressing modes, and so

on. (We should note at this point that there exists a second pair of delimiters for dynamic code, `#{ ... }#`, which differ in that two passes are made over the dynamic code section to perform range–checking on forward branch displacements.) The above kind of "implicit specialisation" is not possible for opcodes, although the same effect can easily be achieved with a C `if-else` or `switch` statements containing alternative dynamic code sections.

Runtime assemblers are implemented *entirely* in `cpp` macros. The most important consequence of this is that any decent C compiler can perform constant–folding and other optimisations at static compile time to produce optimal code for the runtime code generator. For example, the "expanded" code shown above is a somewhat simplified version of reality, since the runtime assembler macros include full error checking (illegal register combinations, literal and branch displacement sizes, and so on) implemented as conditional expressions. In almost all legal cases the C compiler "constant–folds" and "dead–code–eliminates" these checks out of existence since the condition is reduced at static compile time to a constant "true" or "false", leaving only the "store and increment" instructions behind in the final compiled code.

The runtime cost is on average three instructions executed and one and a half memory writes for each instruction generated dynamically. This compares favourably with simple concatenation of pre–assembled "templates" using `memcpy` (the simplest possible alternative for dynamic code generation), and the 6 instructions executed for each generated instruction reported for `ccg`'s closest living relative, `vcode`.

Example: an "RPN compiler" based on "templates"

To illustrate the use of ccg we consider a compiler that creates callable C functions at runtime to evaluate simple numeric functions applied to an argument. The code generator accepts as input a string containing an RPN expression, and returns the address of a dynamically–generated function. The function accepts an integer argument, applies the expression to it, and returns the result. It could be used, for example, to generate a function converting farenheit degrees into centigrade degrees:

```
#cpu pentium

typedef int (*pifi)(int);int main()
{
  pifi f2c= rpnCompile("32-5*9/");
  int i;
  for (i= 32; i <= 212; i+= 10)
    printf("  %3d F = %3d C\n", i, f2c(i));
  return 0;
}
```

The function `rpnCompile` is the runtime code generator. It must perform the following tasks:

1. allocate space for the dynamically–generated function;
2. generate a C prologue appropriate for a function taking one integer argument;

3. “parse” the RPN expression and generate machine instructions to apply that expression to the runtime argument;
4. generate a C epilogue to return the integer result; and
5. return the address of the generated code, of type `pifi` (pointer to `int` function of `int`)

Each of these tasks is shown below. The example is written for the Pentium processor, with which most people should be familiar.

Allocating space for the compiled expression

Most processors (including the Pentium) allow executable code to appear in data space. A buffer for the dynamically–generated code can therefore be allocated simply by calling `malloc`:

```
pifi rpnCompile(char *expr)
{
  static insn *codePtr= 0;
  pifi fn;
  if (codePtr == 0)
    codePtr= (insn *)malloc(1024);
```

This stores a pointer to a buffer holding up to 1024 bytes of dynamically–generated code in the variable `codePtr`.

Generating the C prologue

The prologue must save the frame pointer `ebp`, and then copy the stack pointer `esp` into it to create the frame in which the dynamically–generated function will execute.

```
  #[
        .org    codePtr
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
  ]#
```

The last line in the prologue moves the single int argument from the stack into the working register `eax`. Each operation in the expression will take one argument from the stack and the other from `eax`, leaving the result in `eax`.

Parsing the expression and expanding the appropriate template.

The “parser” is a simple loop that iterates over the string containing the input expression, terminating when it reaches the `'\0'` at the end of the string.

```
  while (*expr) {
```

We must deal with two entities in the input expression: numeric constants and arithmetic operations.

Numeric constants can be conveniently detected using `sscanf` and converted into an `int` using `atoi`. The compiler increments the input expression pointer `expr` past the numeric constant, and generates code to save the current intermediate result in `eax` onto the stack and reload the register with the constant:

```
    int n;
    if (sscanf(expr, "%[0-9]%n", buf, &n)) #[
!       expr+= n - 1;
        pushl   %eax
        movl    $(atoi(buf)), %eax
    ]#
```

Note the call to `atoi` which appears as an *immediate* operand to the `movl` instruction! The call to `atoi` happens during dynamic code generation, and the constant result becomes the literal operand to the generated `movl` instruction which is executed *later*, when the program calls the dynamically–generated function.

If the input element is not a constant then it must be an **arithmetic operation**. These are detected using a `switch` with one `case` per operation. The generated code must pop the "left" operand of the stack and apply it to the "right" operand in the intermediate register `%eax`. The result is left in the register:

```
    else switch (*expr) {
      case '+': #[
          popl    %ecx
          addl    %ecx, %eax
        ]#
        break;
      case '-': #[
          movl    %eax, %ecx
          popl    %eax
          subl    %ecx, %eax
        ]#
        break;
      case '*': #[
          popl    %ecx
          imull   %ecx, %eax
        ]#
        break;
      case '/': #[
          movl    %eax, %ecx
          popl    %eax
          cltd
          idivl   %ecx, %eax
        ]#
        break;
      default:
        fprintf(stderr, "rpnCompile: unknown operator: %s\n", expr);
        abort();
```

```
    }
```

The loop iterating over the input expression can now be terminated:

```
    ++expr;
    }
```

Generating the C epilogue and returning the address of the generated code

The epilogue is trivial. The result is already in the place required by the Pentium ABI (the register %eax), and the instruction `leave` restores the caller's frame before returning:

```
    #[
          leave
          ret
    ]#
    fn= (pifi)codePtr;
    codePtr= asm_pc;
    return fn;
  }
```

At the end of code generation `codePtr` contains the address of the dynamically–generated function and the `ccg`–defined variable `asm_pc` contains the address where the next dynamically–generated instruction will be placed. `rpnCompile` therefore sets `codePtr` to the value of `asm_pc` and returns the original value of `codePtr` as its result. The caller can store this address and then call it to apply the dynamically–compiled expression to an argument in the same manner as calling a statically–compiled C function.

The code produced by calling `rpnCompile("32-5*9/")` is as follows:

```
0x8049d8a:      push   %ebp
0x8049d8b:      mov    %esp,%ebp
0x8049d8d:      mov    0x8(%ebp),%eax
0x8049d90:      push   %eax
0x8049d91:      mov    $0x20,%eax
0x8049d96:      mov    %eax,%ecx
0x8049d98:      pop    %eax
0x8049d99:      sub    %ecx,%eax
0x8049d9b:      push   %eax
0x8049d9c:      mov    $0x5,%eax
0x8049da1:      pop    %ecx
0x8049da2:      imul   %ecx,%eax
0x8049da5:      push   %eax
0x8049da6:      mov    $0x9,%eax
0x8049dab:      mov    %eax,%ecx
0x8049dad:      pop    %eax
0x8049dae:      cltd
0x8049daf:      idiv   %ecx,%eax
```

```
0x8049db1:      leave
0x8049db2:      ret
```

Using ccg for VM implementation

Although extremely simple, the `rpnCompile` example demontrates the essentials of `ccg`. In a real VM the input to the `compile` function would be a sequence of bytecodes, and the output the address of the compiled native code. The `compile` function itself would be a lot more complex, possibly using dataflow analysis to generate optimised dynamic code. For example, the code generator for the stack–based language provided as an example in the `ccg` package is roughly 1000 lines long, and uses a "simulation stack" to generate optimised dynamic code. For the `f2c` function used in the above example it produces

```
0x805352c:      push   %ebp
0x805352d:      mov    %esp,%ebp
0x805352f:      push   %ebx
0x8053530:      mov    0x8(%ebp),%eax
0x8053533:      sub    $0x20,%eax
0x8053536:      imul   $0x5,%eax,%eax
0x8053539:      mov    $0x9,%ebx
0x805353e:      cltd
0x805353f:      idiv   %ebx,%eax
0x8053541:      pop    %ebx
0x8053542:      leave
0x805354b:      ret
```

which is perfectly acceptable. Using `ccg` this optimising dynamic code generator required two days of effort to implement. Without `ccg` the task would have been a lot more difficult.

Availability

The `ccg` preprocessor and runtime assemblers for PowerPC, Sparc and Pentium are available online at http://www–sor.inria.fr/projects/vvm. Full documentation and several examples are provided with the package, including the above RPN compiler and a small stack–based interpreted language that uses an optimising dynamic code generator to achieve the same performance as statically–compiled, optimised C on numerical benchmarks.

Appendix: full text of RPN compiler example

```
/* –*– c –*– */

#cpu pentium

#include <stdio.h>
```

```
#include <stdlib.h>


typedef int (*pifi)(int);


pifi rpnCompile(char *expr)
{
  static insn *codePtr= 0;
  pifi fn;
  if (codePtr == 0)
    codePtr= (insn *)malloc(1024);
  #[
        .org    codePtr
        .align  4
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
  ]#
  while (*expr) {
    char buf[32];
    int n;
    if (sscanf(expr, "%[0−9]%n", buf, &n)) #[
!       expr+= n − 1;
        pushl   %eax
        movl    $(atoi(buf)), %eax
    ]#
    else if (*expr == '+') #[
        popl    %ecx
        addl    %ecx, %eax
    ]#
    else if (*expr == '−') #[
        movl    %eax, %ecx
        popl    %eax
        subl    %ecx, %eax
    ]#
    else if (*expr == '*') #[
        popl    %ecx
        imull   %ecx, %eax
    ]#
    else if (*expr == '/') #[
        movl    %eax, %ecx
        popl    %eax
        cltd
        idivl   %ecx, %eax
    ]#
    else {
      fprintf(stderr, "cannot compile: %s\n", expr);
      abort();
    }
  ++expr;
  }
  #[
        leave
        ret
  ]#
  fn= (pifi)codePtr;
  codePtr= asm_pc;
  return fn;
}
```

```
int main()
{
  pifi c2f= rpnCompile("9*5/32+");
  pifi f2c= rpnCompile("32−5*9/");
  int i;
  for (i= 0; i <= 100; i+= 10)
    printf("  %3d C = %3d F\n", i, c2f(i));
  for (i= 32; i <= 212; i+= 10)
    printf("  %3d F = %3d C\n", i, f2c(i));
  return 0;
}
```