ccg: dynamic code generation for C and C++

Ian Piumarta `<http://www-sor.inria.fr/~piumarta/>` $Id: ccg.sgml,v 1999/03/21 piumarta Exp $

This document contains brief notes on "ccg", a combination of preprocessor and "runtime assembler" for C and C++ programs. It allows extremely 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. The system is written *entirely* in ANSI C, and requires no compiler-specific extensions (such as inline `asm` statements or the various assembler-related extensions implemented by `gcc`).

Contents

http://www-sor.inria.fr/projects/vvm/ccg/

1 Introduction

`ccg` is an aid to the development and implementation of dynamic code generators. The main components are a preprocessor (the program that is actually called "`ccg`") and a collection of "runtime assemblers" (a set of header files whose names are utterly unimportant).

1.1 Overview

The preprocessor accepts as input a `C` program that contains embedded *dynamic code sections*. (For the sake of brevity we will consider "`C`" to cover both `C` and `C++`.) Each dynamic code section is a sequence of assembly language statements. The output from the preprocessor is an equivalent `C` program in which the dynamic code sections have been replaced with (pure ANSI) `C` code that generates the corresponding binary instructions *at runtime*.

The majority of code generation work is performed by a runtime assembler. Each runtime assembler is implemented as a set of `cpp` macros in a header file specific to the target platform. Since the *entire* assembler is implemented in macros (on which the compiler can perform constant folding), the majority of instructions are assembled completely at (static) compile time. They are reduced to nothing more elaborate than sequences of "`move #constant, address`" in the compiled program.

On the other hand, since the runtime assemblers are also *complete*, and presented to the client program as calls on `cpp` macros in which each operand is passed as an integer argument, the client program has total freedom in specialising all aspects of the generated code: literal constants, registers, branch/jump displacements, the various elements of more complex addressing modes, and so on, can all be *computed* by the program during code generation. Moreover, the client program only pays a performance penality for those operands which are not constant at compile time (and which therefore cannot be "constant folded out of existence" by the compiler).

The targets currently supported are:

- PowerPC
- Sparc
- Pentium

Since the entire `ccg` system is implemented in ANSI `C`, and the output from the preprocessor is also ANSI, it can be used on a wide variety of platforms. Most notably it is suitable for use on platforms where other common "solutions" to dynamic code generation (such as the use of inline `asm` and many related `gcc`-specific extensions) are not available, e.g. MacOS.

1.2 How `ccg` works: a quick appetiser

The programmer includes assembly language fragments in the high-level source program. For example:

```
movl    $42, %eax
```

The preprocessor converts these statements into calls on macros that are implemented in the runtime assembler for the target platform (which is *not necessarily the same* as the host platform). After preprocessing, the above statement is transformed into:

```
MOVLir  (42,_EAX);
```

The runtime assembler provides the definitions of the macro `MOVLir()` and the constant `_EAX` (representing the register "`%eax`"). Since both arguments to the `MOVLir()` macro are constants, the compiler can fold them away entirely and produce "code generation code" for the above statement that will be something similar to this:

```
*(char *)(someAddress)     = 0xB8;      /* insn: move imm32 -> %eax */
*(long *)(someAddress + 1) = 0x2A;      /* constant: 42 */
```

This is the code that is actually compiled into the client program. The final stage of assembly (the placing of binary code into memory) happens at runtime, at the moment when the above statements are executed.

(The insatiably curious can run `ccg` without command-line arguments on the programs in the `ccg/examples` directory to see how the preprocessor modifies the source code. The runtime assemblers are in the `ccg/asm-`*`arch`*`.h` files and should be easily comprehensible. Nevertheless, begin by looking at the PowerPC and Sparc assemblers files before making any attempt to understand the assembler for the Pentium!)

1.3 Raison d'être

`ccg` is obviously not platform-independent. A given program file can generate code that runs on one architecture only, and includes assembly language statements for that architecture. Portability of dynamic code generation is *not* the goal of `ccg`, although the support for portability across compilers and operating sstems for a given processor architecture *is* one of the major goals.

`ccg` was developed to support the very lowest (platform-dependent) layer in a portable, optimising, dynamic code generator. This system performs the majority of the code generation and optimisation work using a platform-independent "intermediate" representation, and then calls a pluggable "back end" (implementing a platform-independent interface) to convert the intermediate form into machine instructions.

We have found that porting the back-end to a new platform is considerably easier with a tool such as `ccg`, and requires only a day or two of work.

2 Using `ccg`

This section introduces the platform-independant parts of the assembly language syntax accepted by `ccg`, and lists the directives that it recognises. It begins by describing the overall layout of a program file that uses `ccg`. This first subsection necessarily uses a few features before they are explained in detail. These detailed explanations of assembly language syntax and preprocessor directives are given in the subsequent subsections.

2.1 Program organisation

Two elements are required in any program that uses the `ccg` preprocessor. These are:

- an initial processor type declaration;

followed by any number of runtime code generation sections consisting of the following:

- a directive indicating the start of the runtime code section;
- a directive controling the address at which code is generated;
- dynamic assembler statements written using a syntax natural for the assembly language of the selected processor type;
- a directive indicating the end of the runtime code section.

First is the directive to select of the processor type (in this case the Intel Pentium):

```
#cpu pentium
```

This directive has two effects:

1. it determines the assembler syntax for opcodes and operands that is accepted by the preprocessor; and
2. it inserts into the output an `#include` directive specific to the selected target processor (to include the header file that implements the appropriate runtime assembler).

Since it causes the inclusion of a header file, it *must* appear at the outermost (global) scope of the program source, *outside* any function definition.

Next come the runtime code generation sections. These must occur *inside* a function defintition, since they are converted into executable `C` statements that perform code generation. There are two such directives:

```
#[
```

begins a runtime code section that requires no resolution of forward references (and which can therefore be generated in a single pass); the other

```
#{
```

begins a section which is generated in two passes, in order to correctly resolve forward references.

The "begin" directive is followed by a pseudo-op specifying the address at which code will be generated. Assuming that we have allocated a suitable region of memory and stored the address in the variable `codeBuf`, we would write:

```
.org    codeBuf
```

We can now write assembly language statements. Each statement is written using the natural syntax for the processor selected previously by the `#cpu` directive. Using the Pentium as an example, we might write two statements that implement a trivial function that returns an integer constant:

```
movl    $42, %eax
ret
```

(Note that the assembly language statements can refer freely to `C` program elements, such as variables or constants that were defined with `#define`.)

The end of the code section is delimited by a directive matching the one that began the code section; i.e:

```
]#
```

for single-pass sections, or

```
}#
```

for two-pass sections.

Note that the begin/end directives are semantically equivalent to `C` braces, and therefore constitute a compound statement:

```
if (someCondition) #[
  ... generate consequent code
]# else #[
  ... generate alternate code
]#
```

We can now call the dynamically generated code, for example by casting the buffer addres into a "`int` function of `void`" and then calling the result as a function:

```
int result= (int(*)(void))codeBuf();    /* execute the generated code */
printf("%d\n", result);
```

The complete program would look like this:

```
#include <stdio.h>

#cpu pentium

int main()
{
  insn codeBuf[1024];
  int result;
  #[
        .org    codeBuf
        movl    $42, %eax
        ret
  ]#
  result= (int(*)(void))codeBuf();
  printf("%d\n", result);
  return 0;
}
```

Space is reserved for the code by declaring an array of `insns`. This type is defined by the `#cpu` directive to be appropriate for the target processor (currently `unsigned int` for 32-bit RISC processors and `unsigned char` for the Pentium).

Note that in the above example the dynamically generated code is placed in "automatic" storage. There are probably not very many other examples of `C` programs that execute functions whose code is in the program's stack!

2.2 Assembly language statements

Dynamic assembly language statements have similar (although simplified) conventions to regular static assemblers. Each statement consists of up to four elements:

- `name:` *(optional)*

  defines a label with the given *name*. The label is a `C` variable (which must already have been declared, either explicitly in `C` or using the `.label` pseudo-op. (The corresponding code in the output file just stores the current assembly address into the variable `name`.)

- `opcode` *(optional)*

  this is transformed into call on a runtime assembler macro to perform the assembly of the instruction named by `opcode`.

- `operands...` *(optional)*

  a comma-separated list of operands which are transformed into the arguments passed to the runtime assembler macro. (Note that on most architectures the types of the operands affect the name of the runtime assembler macro to permit "overloading" of a single opcode with several instructions taking different numbers or types of operands.)

  Note that a dot ("`.`") appearing anywhere in an operand is converted into the address of the opcode associated with the operand.

- `# comment...` *(optional)*

  everything between the "`#`" and the end of the line is converted into a `C`-style comment in the output.

Preprocessor directives are similar in appearance to the directives recognised by `cpp`, the regular `C` preprocessor:

- `#directive [`*argument*`]`

  where the "`#`" *must* appear in the first column for the directive to be recognised.

The directives are described in the next section.

Instead of an instruction (an opcode plus zero or more operands), an assembly language statement can take the form of a "pseudo-op":

- `.pseudoOp [`*argument*`]`

  where the "`#`" *must* appear in the first column for the directive to be recognised.

Most of these are common to all target platforms, although some targets may provide additional platform-dependent pseudo-ops. The pseudo-ops are described in the next section.

The preprocessor checks all opcodes and operands for correct syntax as it processes the source file. It also checks that the operand types are legal for the opcode. If the preprocessor does not generate any error messages then the resulting `.c`[c] file should compile without any warnings or errors related to the runtime assembler. (If it does then you've found a bug in the preprocessor and/or runtime assembler macro definitions.)

The preprocessor is also careful to preserve the correspondance between input lines and output lines. Line numbers in error messages from `cpp` or the `C` compiler can therefore be related directly to the original `ccg` input file.

The following sections describe the preprocessor in detail. We begin by explaining the required elements of a program that uses runtime code generation. After that we will describe all of the preprocessor directives and common pseudo-ops that are available, and finish by giving a simple but complete example using many of the preprocessor's features.

2.3 Preprocessor directives and assembler pseudo-ops

Several directives can only appear before the `#cpu` directive (the "preamble"), and some can appear anywhere. They are presented in these two categories.

2.3.1 Preamble directives

If present, the following directives must appear *before* the `#cpu` directive:

- `#localpc`

  indicates that the program will supply its own definitions of `asm_pc` and `asm_pass`. (These variables are normally declared by the target header file. The `#localpc` directive is primarily useful when there are multiple program files that are logically related and contain dynamic code sections: all but one of these files should include the `#localpc` directive to avoid multiple definitions. It is also useful when the function containing the dynamic code section wants to define `asm_pc` and/or `asm_pass` as a `register` variables to improve performance.)

Zero or more of the above are followed by the required platform selection directive:

- `#cpu` *platform*

  selects *platform* as the target. Only one `#cpu` directive can appear in a given program file. It *must* appear *after* any `#localpc` directive.

2.3.2 General directives

The following directives can appear anywhere:

- `#quiet`

  disables warning messages. (Normally a warning is issued whenever a label is defined inside a single-pass (`#[ ... ]#`) code block.)

- `#comment` *`commentChar`*

  changes the current comment character. The default comment character is a hash ('`#`').

- `#escape` *`escapeChar`*

  changes the current escape character. The default escape character is an exlamation point ('`!`', sometimes also called "bang" or "pling").

- `#[`

  begins a single-pass dynamic code section. Forward references are not correctly resolved in these sections.

- `#{`

  begins a two-pass dynamic code section. Forward references are correctly resolved in these sections.

- `]#`

  ends a single-pass dynamic code section.

- `}#`

  ends a two-pass dynamic code section.

2.3.3 Pseudo-ops

The following pseudo-ops can only appear withing a dynamic code section:

- `.org` ***address***

  specifies the ***address*** at which the next dynamically generation instruction will be stored. At least one `.org` must be executed before any dynamic code is generated.

- `.label` ***name...***

  declares one or more temporary program labels that can be used within the current dynamic code section. Note that labels declared in this manner must obey the placement restrictions of the host language. (In `C` they must appear before any assembly language statements, pseudo-ops or escaped `C` statements within dynamic code sections. In `C++` there are no such restrictions, since declarations can appear anywhere.)

- **`!`** *anything*

  an "escaped" `C` statement. ***anything*** is included "verbatim" in the output program. This is useful for embedding arbitraty `C` statements within dynamic code blocks. Note that the '**`!`**' must appear in the *first* column for it to be recognised, and that it takes precedence over the comment character (in the case where the comment character and/or escape characters have been redefined to the same thing).

2.4 Complete example

We will illustrate the preprocessor and runtime assembler using a simple (but complete) example for the Pentium. The program is shown first, followed by a line-by-line explanation.

```
 1  #cpu     pentium
 2  #comment !
 3  #escape  @
 4
 5  #include <stdio.h>
 6
 7  insn codeBuffer[1024];                  /* buffer for generated code */
 8
 9  int main()
10  {
11    typedef void (*pvfi)(int);            /* Ptr to Void Func of Int */
12    pvfi myFunction= (pvfi)codeBuffer;    /* the generated function */
13    insn *start, *end;                    /* a couple of labels */
14    /* generate some code... */
15    #[
16  @       printf("assembling: pass %d\n", asm_pass);
17          .org    myFunction              ! generate code at this address
18          ! prologue
19  start:  pushl   %ebp
20          movl    %esp, %ebp
21          ! body
22          pushl   8(%ebp)                 ! incoming argument
23          pushl   $(int)"generated %d bytes\n"
24          call    printf
25          ! epilogue
26          leave
27          ret
28  end:
29    ]#
30    /* call the generated code, passing its size as argument */
31    myFunction(end - start);
32    return 0;
33  }
```

The program first selects the Pentium as the target processor (line 1). It then changes the comment and escape characters to '`!`' (in the Sparc style) and '`@`', respectively (lines 2 and 3). Line 13 declares two variables which will be assigned interesting addresses during code generation. The dynamic code section begins on line 15. Line 16 is an escaped `C` statement that prints the current assembly pass, followed by the pseudo-op to set the initial program address on line 17. The two "external" label variables are assigned using the usual label syntax on lines 19 and 28. The dynamic code section is closed on line 29.

(Note that format string passed as an argument to the `printf` function on line 23 is a `C` expression. All arguments (including register numbers) can be given as arbitrary expressions, and can refer freely to the values of `C` variables in effect at the time the code is generated.)

Running the above program generates the following output:

```
assembling: pass 1
assembling: pass 2
generated 18 bytes
```

The first two lines of output are generated during the two passes through the dynamic code section during runtime assembly. The last line of output is generated by the generated code itself (from the call to `printf` on line 24 of the program).

2.5 Computed with register numbers

The runtime assemblers often encode registers with a number not obviously related to the "index" of the register. For example, the registers `%g0`, `%o0`, `%l0` and `%i0` on the Sparc all have different constant values that are not obvious to the client program. The situation is more complex on the Pentium where the registers `%al`, `%ah`, `%ax` and `%eax` all refer to the same physical register but have four different encodings to allow the assembler to choose the appropriate opcode width and operands.

Providing the program with access to the register encodings for a particular plaform could be done by defining macros in the runtime assembler, but this is ugly. Instead, `ccg` provides a "mini" assembler section, delimited with `#(` and `)#`, which can contain *a single register operand* (and nothing else!). The value of this expression is the encoding used by the assembler for the given register.

The following examples are intentionally a little obscure in places, to illustrate the possibilities. On the Sparc:

```
static const int stackPointer  = #( %sp )#;
static const int framePointer  = #( %fp )#;

void genReturnReg(int index)
{
  if ( #(%r(index))# != #(%i0)# ) #[
        mov     %r(index), %i0
  ]#
  #[
        ret
        restore
  ]#
}

void genReturnArg(index)
{
  genReturnReg(#(%i(index))#);
}
```

And on the pentium:

```
int addImmediate(Datum *data)
{
  int destReg= 0;
  switch (data->size) {
    case 1:      destReg= #( %cl  )#; break;
    case 2:      destReg= #( %cx  )#; break;
    case 4:      destReg= #( %ecx )#; break;
    default: abort();
  }
  generateAddInto(data->value, destReg);
  return destReg;
}
```

2.6 Synchronising the instruction and data caches

As with any dynamic code generation system, the data and instruction caches must be flushed before executing the generated code. (This is not necessary in the simple examples in this document because all Pentium machines appear to have unified caches that do not require explicit synchronisation.)

It is the client program's responsibility to flush the caches. All of the runtime assemblers provide the macro

```
iflush(insn *firstAddress, insn *lastAddress)
```

to perform this flushing. (On the Pentium this is a no-op. On the Sparc and PowerPC, failure to correctly flush the caches will result in an illegal instruction trap.) Typical use would be:

```
void aCodeGenerator(void)
#[
        .org    codeAddress
        ... statements ...
! iflush(codeAddress, asm_pc);
]#
```

(`asm_pc` contains address of the byte after the last instruction that was generated).

2.7 Using `ccg` with `make`

The following implicit `Makefile` rules will sensibly control the regeneration of `.c` and `.cc` files from the corresponding `cgg` input files (with `.cg` and `.ccg` extensions, respectively).

```
# path to ccg preprocessor
CCG:=           ccg/ccg

.SUFFIXES:      .cg .ccg

%.c:            %.cg
                $(CCG) -o $@ $<

%.cc:           %.ccg
                $(CCG) -o $@ $<

# include the following if you want to keep intermediate .c[c]
# files (e.g. for source file listing during debugging)
.PRECIOUS:      %.c %.cc
```

The distribution includes the file `ccg.mk` which contains the above definitions and rules. To use it simply make a link to the `ccg` distribution directory in your source directory and then add

```
include ccg/ccg.mk
```

to your existing `Makefile`.

2.8 Useful runtime assembler macros

Several of the macros used internally by the runtime assemblers are probably useful to the client program. They include:

- `_uiP(W,UI)`

  is a boolean predicate returning “true” if the unsigned integer `UI` can be fully represented in a field of `W` bits.

- `_siP(W,SI)`

  is a boolean predicate returning “true” if the signed integer `SI` can be fully represented in a field of `W` bits.

- `_MASK(W)`

  returns an `unsigned long` in which the bottom `W` bits are set, and all other bits cleared.

3 Platform-specific notes

This section defines the syntax accepted by the preprocessor for each of the supported platforms, and briefly mentions any significant differences between the "standard" assembly language conventions for the platform and those supported by the preprocessor and runtime assemblers.

The following subsections assume familiarity with the assembly language of each processor. No attempt is made to explain obscure (but standard) assembly language features.

3.1 PowerPC

The preprocessor and assemblers follow the conventions given in:

- *PowerPC Microprocessor Family: The Programming Environments For 32-Bit Microprocessors*, Motorola, 1997.

The destination operand is always on the left, immediate operands always on the right, and registers are always prefixed with '`r`' to distinguish them from immediate values. The immediate opcodes have an *optional* suffix '`i`':

```
add     r3, r4, r5      # load r3 with the sum r4 + r5
addi    r3, r4, 5       # load r3 with the sum r4 + 5
add     r3, r4, 5       # synonym for "addi r3, r4, 5"
```

Memory addresses are indicated by a base register in parentheses preceded by an immediate offset, or two registers with the '`x`' form of the instruction:

```
lwz     r3, r4(5)       # load r3 from the word after r5
lwzx    r3, r4, r5      # load r3 from the word at r4+r5
lwz     r3, r4(r5)      # synonym for "lwzx r3, r4, r5"
```

Opcodes that side-effect the condition codes have a dot ('`.`') suffix:

```
add     r3, r4, r5      # load r3 with the sum of r4 + r5
add.    r3, r4, r5      # load r3 with the sum of r4 + r5 and set cr0 accordingly
```

3.1.1 Simplified mnemonics

Many of the simplified branch mnemonics, and all of the simplified mnemonics for non-branch instructions, are recognised.

3.1.2 Limitations

The branch and compare instructions support optional condition code register and bit operands. There is no assembler support for the specification of these as symbolic constants. They are however treated as immediate operands, and so it is trivial to define constants for these registers and bit positions when needed. For example:

```
#cpu ppc

#define cr0     0
#define cr1     1
#define cr2     2
#define cr3     3

#define lt      0
#define gt      1
#define eq      2
#define so      3

void aCodeGenerator(void)
#[
        cmpl    cr2, 0, r3, r4
        bt      cr2*4+eq, equalLabel
        bt      cr2*4+lt, lessLabel
]#
```

(This corresponds precisely to the syntax defined by Motorola.) If the condition code register is not specified in compare and branch instructions then it is implicitly `cr0`.

3.2 Sparc

Opcode and operand syntax is as defined in

- SPARC International, *The SPARC Architecture Manual, Version 8*, Prentice-Hall, 1992.

Registers are prefixed with '`%r`'. They can also be specified by their alternative mnemonic forms:

- `%g0 .. %g7`

  correspond to registers `%r0` through `%r7`
- `%o0 .. %o7`

  correspond to registers `%r8` through `%r15`
- `%l0 .. %l7`

  correspond to registers `%r16` through `%r23`
- `%i0 .. %i7`

  correspond to registers `%r24` through `%r31`
- `%sp`

  corresponds to register `%o6` (or `%r14`)
- `%fp`

  corresponds to register `%i6` (or `%r30`)

The two operators for extracting the low and high portions of immediates are supported:

- `%lo(`*imm32*`)`

  is the low 10 bits of *imm32*

- `%hi(`*imm32*`)`

  is the high 22 bits of *imm32*, shifted right by 10 bits without sign extension (suitable for use as the immediate operand for a `sethi` instruction)

All of the "synthetic" instructions listed in SPARC International documentation are supported (including the `set` instruction which expands to one or two instructions depending on the size of its immediate argument).

3.2.1 Memory operands

The suggested Sparc syntax does not place square brackets around memory locations when the location itself is the operand (and not the contents of the location); e.g. for jump instructions:

```
jmpl    %o7+8, %g0
```

Handing this in a clean manner was too painful, and such instructions must therefore use the bracketed memory notation:

```
jmpl    [%o7+8], %g0
```

3.2.2 Computed register numbers

Computed registers require parentheses around the register number, for example:

```
add     %i(argIn), %i(argIn+1), %o(argOut)
ld      [%r(baseReg)+offset], %r(destReg)
ld      [%r(baseReg)+%r(indexReg)], %r(destReg)
jmpl    [%r(isLeafProc ? 31 : 15)+8], %g0       # choose ret or retl
```

3.2.3 Comment character

SPARC International suggest '`!`' as the comment character. In `ccg` it remains the usual '`#`' although it can be changed trivially by including

```
#comment !
```

near the start of the source file, or by specifying "`-c !`" on the `ccg` command line.

3.3 Pentium

The opcode names follow the specification given in:

- *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, Intel Corporation, 1997.

3.3.1 Source and destination operand positions

There is serious disagreement between Intel and the Free Software Foundation (the GNU people) over the position of the source/destination operands. Intel syntax specifies the first operand as the destination (data moves from right to left), for example:

```
movl    %eax, $42       # load %eax with 42
```

On the other hand the GNU assemblers and disassemblers all put the destination in the final operand (data moves from left to right):

```
movl    $42, %eax       # move 42 into %eax
```

We choose to follow the FSF/GNU convention since it is more logical, intuitive, aesthetically pleasing, and possibly even more ethically sound (considering the identity of Intel's most [in]famous client).

3.3.2 Computed register indices

As in the Sparc assembler, "computed" register indices must be placed in parentheses after the '%' prefix. For example:

```
void gen_load(int value, int destReg)
{
  #[
        movl    $value, %(destReg)
  ]#
}

...
  gen_load(42, #( %eax )#);
...
```

3.3.3 Additional pseudo-ops

The Pentium target provides an additional pseudo-op "`.align` *N*", where *N* is an alignment (expressed in bytes) between 1 and 8.

3.3.4 Assembler complexity

Compared with RISC processors, the Pentium has a *very* complex instruction set. The runtime assembler must perform a lot of dynamic analysis to determine the optimal instruction to generate for a given combination of opcode and operands. It is therefore rather complex (consisting of approximately 640 macro definitions, many of which contain hairy conditional expressions to select the correct opcode prefix byte(s), sign-extension and operand width bits, and so on). A given Pentium opcode can generate anything from 1 to 13 bytes of code, depending on the nature of the operands that are passed to it.

Because of this complexity a Pentium dynamic assembler statement ultimately generates many hundreds of characters of `C` code (after the compiler has passed the `.c` file through `cpp`) containing deep conditional expressions. The compiler has to work quite hard to optimise this mess. (The insatiably curious might like to run “`gcc -E`” on a `.c` file produced by `ccg` to see the extent of the problem.) For example, with `gcc` the compile-time overhead is about 50-100 kilobytes of virtual memory for each dynamic assembler statement. Under these conditions it is not difficult to cause the compiler to run out of memory.

The solution is to split the dynamic code sections over several different `C` functions, limiting each function to no more than a hundred or so assembler statements. (The compiler discards some intermediate structures after processing each function.) For very complex code generators the functions containing dynamic code sections can be placed in independent source files to further reduce the memory requirements.

Note that this problem does not exist with the runtime assembers for PowerPC and Sparc, which both have significantly simpler instruction formats requiring significantly less runtime analysis.

4 Customising `ccg`’s behaviour

Several aspects of the runtime assembler and preprocessor can be customised for a particular client program. This is occasionally useful and not difficult for assembler “wizards” to do, but it does require a certain familiarity with the preprocessor and runtime assemblers. If in doubt, seek professional help.

4.1 Customising the runtime assembler

The runtime assemblers share a common framework in which several actions can be changed from their defaults by the client program. The customisable operations are:

- the actions associated with the start and end of dynamic code sections, corresponding to the directives “`#[`”, “`#{`”, “`]#`” and “`}#`”.
- the action associated with the “`.org`” pseudo-op.
- the action associated with the definition of a program label, corresponding to “*`name`* `:`”.

Customisation is achieved by defining one or more macros *before* using the `#cpu` directive. The macros concerned, and their default definitions, are as follows:

- the action associated with “`#[`”

```
#define _ASM_APP_1      { asm_pass= 0; {
```

- the action associated with “`]#`”

```
#define _ASM_NOAPP_1    } asm_hwm= asm_pc; }
```

- the action associated with “`#{`”

```
#define _ASM_APP_2      { for (asm_pass= 1; asm_pass < 3; ++asm_pass) {
```

- the action associated with “`}#`”

```
#define _ASM_NOAPP_2    if (asm_pass==1) asm_hwm= asm_pc; \
                        else if (asm_hwm!=asm_pc) ASMFAIL("phase error"); } }
```

- the action associated with the pseudo-op "`.org O`"

```
#define _ASM_ORG(O)     (asm_pc= (O))
```

- the action associated with label declaration "`.label N`"

```
#define _ASM_LBL(N)     static insn *N= 0
```

- the action associated with label definition "`N:`"

```
#define _ASM_DEF(N)     (N= (((asm_pass==2)&&(asm_pc!=(N))) \
                             ? (insn *)ASMFAIL("phase error") : asm_pc))
```

4.2 Customising the preprocessor

Several preprocessor characteristics can be altered by changing definitions near the beginning of the source file `ccg.c` and then recompiling. The customisable definitions and their default values are as follows:

- the prefix added to the canonical target name in the `#include` directive corresponding to the `#cpu` declaration

```
#define HEADER_PREFIX   "ccg/asm-"
```

- the set of characters that can appear in labels (some clients may want to add the dollar sign ('`$`') to this list)

```
#define LABEL_CHARS     "_A-Za-z0-9"
```

- the "tokens" recognised as the start and end delimiters of one- and two-pass dynamic code sections

```
#define ASM_1_OPEN      "#["
#define ASM_1_CLOSE     "]#"
#define ASM_2_OPEN      "#{"
#define ASM_2_CLOSE     "}#"
```

4.3 Customisation example

One simple but useful customisation is to implement automatic synchronisation of the instruction and data caches at the end of each dynamic assembler section. One way to achieve this is as follows:

```
... header files or other preambulae ...

/*** customised runtime assembler actions: add automatic iflush(lwm,pc) ***/

#define _ASM_NOAPP_1    } iflush(asm_lwm, asm_pc); asm_lwm= asm_pc; }
```

```
#define _ASM_NOAPP_2    if (asm_pass==1) asm_hwm= asm_pc; \
                        else if (asm_hwm!=asm_pc) ASMFAIL("phase error"); \
                        else { iflush(asm_lwm, asm_pc); asm_lwm= asm_pc; } } }

#define _ASM_ORG(ADDR)  if (asm_lwm != 0) iflush(asm_lwm, asm_pc); \
                        (asm_lwm= asm_pc= (ADDR))

#cpu nameOfTargetCPU

/* the following variable must be _defined_ in only ONE program file */

extern insn *asm_lwm;   /* "low water mark" for cache flush */

/*** end of customised runtime assembler actions ***/

... program ...
```

5 Example programs

This section includes a few tiny exmaple programs to demontrate how `ccg` works on the supported architectures. Each subsection covers one example, with a subsubsection for each architecture.

5.1 Instruction counter

This example generates the code required to print out its argument in decimal by calling `printf`. The code is called with the number of instructions generated as the argument.

5.1.1 PowerPC

```
#cpu ppc

#include <stdio.h>

static insn codeBuffer[20];

typedef void (*pvfi)(int);      /* Pointer to Void Function of Int */

int main()
{
  pvfi myFunction= (pvfi)codeBuffer;    /* the generated function */
  insn *start= 0, *end= 0;              /* a couple of labels */
  /* generate some code... */
  #[
        .org    myFunction              # generate code at this address
        # prologue: fake -- just move LR into a call-saved register
start:  mflr    r28
        # body
!{
```

```
!       int fmt= (int)"generated %d instructions\n"; /* cache for 2 refs */
        mr      r4, r3          # first argument -> second printf argument
        lis     r3, _HI(fmt)
        ori     r3, r3, _LO(fmt)
        bl      printf
!}
        # epilogue: fake -- restore LR and return
        mtlr    r28
        blr
end:
  ]#
  /* call the generated code, passing its size as argument */
  iflush(codeBuffer, asm_pc);
  myFunction(end - start);
  return 0;
}
```

5.1.2 Sparc

```
#cpu sparc

#include <stdio.h>

static insn codeBuffer[20];

typedef void (*pvfi)(int);      /* Pointer to Void Function of Int */

int main()
{
  pvfi myFunction= (pvfi)codeBuffer;    /* the generated function */
  insn *start, *end;                    /* a couple of labels */
  /* generate some code... */
  #[
        .org    myFunction              # generate code at this address
        # prologue: no temps
start:  save    %sp, -96, %sp
        # body
        mov     %i0, %o1
        set     (int)"generated %d instructions\n", %o0
        call    printf
        nop
        # epilogue
        restore
        ret
        nop
end:
  ]#
  /* sync the caches */
  iflush(start, end);
  /* call the generated code, passing its size as argument */
  myFunction(end - start);
  return 0;
}
```

5.1.3 Pentium

```
#cpu pentium

#include <stdio.h>

static insn codeBuffer[32];

typedef void (*pvfi)(int);      /* Pointer to Void Function of Int */

int main()
{
  pvfi myFunction= (pvfi)codeBuffer;    /* the generated function */
  void *start, *end;                    /* a couple of labels */
  /* generate some code... */
  #[
        .org    myFunction              # generate code at this address
        # prologue
start:  pushl   %ebp
        movl    %esp, %ebp
        # body
        pushl   8(%ebp)         # first argument
        pushl   $(int)"generated %d bytes\n"
        call    printf
        # epilogue
        leave
        ret
end:
  ]#
  /* call the generated code, passing its size as argument */
  myFunction(end - start);
  return 0;
}
```

5.2 “Reverse polish” notation compiler

This example implements a small “compiler”. The compiler converts a string containing an expression in “reverse polish notation” into a dynamically generated function that applies the expression to its argument. Only the compiler function differs for the various architectures, and all share a common definition of `main` as shown here:

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

The platform-dependent code generation is performed by the function `rpnCompile()`, as follows...

5.2.1 PowerPC

```
#cpu ppc

#include <stdio.h>
#include <stdlib.h>

typedef int (*pifi)(int);

pifi rpnCompile(char *expr)
{
  static insn *codePtr= 0;
  pifi fn;
  int top= 3;
  if (codePtr == 0) {
    codePtr= (insn *)malloc(1024);
    printf("code at %p\n", codePtr);
  }
  #[ .org codePtr ]#
  while (*expr) {
    char buf[32];
    int n;
    if (sscanf(expr, "%[0-9]%n", buf, &n)) #[
!      expr+= n - 1;
!       ++top;
        lis     r(top), _HI(atoi(buf))
        ori     r(top), r(top), _LO(atoi(buf))
    ]#
    else if (*expr == '+') #[
!       --top;
        add     r(top), r(top), r(top+1)
    ]#
    else if (*expr == '-') #[
!       --top;
        sub     r(top), r(top), r(top+1)
    ]#
    else if (*expr == '*') #[
!       --top;
        mullw   r(top), r(top), r(top+1)
    ]#
    else if (*expr == '/') #[
!       --top;
        divw    r(top), r(top), r(top+1)
    ]#
    else {
      fprintf(stderr, "cannot compile: %s\n", expr);
      abort();
    }
  ++expr;
  }
  #[    blr     ]#      /* return */
  iflush(codePtr, asm_pc);
  fn= (pifi)codePtr;
  codePtr= asm_pc;
  return fn;
}
```

5.2.2 Sparc

```
#cpu sparc

#comment !      /* traditional sparc comment char */
#escape  @      /* not necessary, but avoids ambiguity and confusion */

#include <stdio.h>
#include <stdlib.h>

typedef int (*pifi)(int);

pifi rpnCompile(char *expr)
{
  static insn *codePtr= 0;
  pifi fn;
  int top= _Ro(0);
  if (codePtr == 0)
    codePtr= (insn *)malloc(1024);
  #[
        .org    codePtr         ! leaf procedure: empty prologue
  ]#

  while (*expr) {
    char buf[32];
    int n;
    if (sscanf(expr, "%[0-9]%n", buf, &n)) #[
@       expr+= n - 1;
@       ++top;
        set     (atoi(buf)), %r(top)
    ]#
    else if (*expr == '+') #[
@       --top;
        add     %r(top), %r(top+1), %r(top)
    ]#
    else if (*expr == '-') #[
@       --top;
        sub     %r(top), %r(top+1), %r(top)
    ]#
    else if (*expr == '*') #[
@       --top;
        smul    %r(top), %r(top+1), %r(top)
    ]#
    else if (*expr == '/') #[
@       --top;
        sdiv    %r(top), %r(top+1), %r(top)
    ]#
    else {
      fprintf(stderr, "cannot compile: %s\n", expr);
      abort();
    }
  ++expr;
  }
  #[    ! epilogue
        retl
        nop                     ! delay slot
  ]#
  iflush(codePtr, asm_pc);
  fn= (pifi)codePtr;
  codePtr= asm_pc;
  return fn;
}
```

5.2.3 Pentium

```
#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
        # prologue: save and reload frame pointer
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
  ]#
  /* top of stack is always in %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;
  }
  #[    # epilogue
        leave
        ret
  ]#
  fn= (pifi)codePtr;
  codePtr= asm_pc;
  return fn;
}
```

5.3 Dynamic bytecode translation

This is a simple programming language that parses a source file and then compiles the program to either an interpreted bytecode set or to native code using a `ccg`-based dynamic code generator. The example sources are too large for inclusion in this document, and can be found in the directory *ccg/exampes/lang* `<ccg/examples/lang>`.