Syn68k
ARDI's dynamically compiling 68LC040 emulator

Mat Hostetter <mat@ardi.com>

October 27, 1995

1 Overview

This document is meant to give a concise technical summary of how syn68k works.

"Syn68k," ARDI's 68LC040 emulator, is both highly portable and fast. The portable core of syn68k, which works by dynamically compiling 680x0 code into an efficient interpreted form, was designed to run on all major CPU's. On supported architectures, syn68k can also translate 680x0 code into native code that the host processor can run directly.

2 Syngen

ARDI's "syngen" system analyzes a lisp-like file describing the bit patterns and semantics of the 680x0 instruction set and produces lookup tables and C code for the runtime system to use. This process takes place only when syn68k is built, so we can afford extensive analysis here. The code and tables generated by syngen depend somewhat on the characteristics of the host processor; for example, on a little endian machine it is advantageous to byte swap some extracted 680x0 operands at compile time instead of at runtime.

The 680x0 description file can describe multiple ways to emulate any particular 680x0 opcode. The runtime system looks at what CC bits are live after the instruction and chooses the fastest variant it can legally use. In the following example, we have two CC variants of lsrw; one computes no CC bits, and the other computes all of them:

```
(defopcode lsrw_ea
  (list 68000 amode_alterable_memory () (list "1110001011mmmmmm"))
  (list "-----" "-----" dont_expand
(assign $1.muw (>> $1.muw 1)))
  (list "CN0XZ" "-----" dont_expand
```

```
(list
 (assign ccx (assign ccc (& $1.muw 1)))
 (ASSIGN_NNZ_WORD (assign $1.muw (>> $1.muw 1))))))
```

The 680x0 description file can also specify which 680x0 operands should be "expanded" to become implicitly known by the corresponding synthetic opcode. For example, fully expanding out "addl dx,dy" would result in 64 synthetic opcodes, one for each combination of data register operands. This results in smaller and faster synthetic opcodes at the expense of increasing the total number of synthetic opcodes. To conserve space, we only expand out common 680x0 opcodes. On host architectures where we can compile to native code, we don't waste space by "expanding out" common synthetic opcodes.

3 Test suites

ARDI has a large set of test suites that try thousands upon thousands of variations of 680x0 opcodes and compare the results to those generated by a real 68040. These test suites have proven to be an invaluable debugging tool, both as new features are added and as we have ported syn68k to other architectures (notably 80x86, 680x0, i860 and Alpha). Our native code support is so recent that our test suites do not yet adequately test all of the situations that arise when generating native code, but we plan to extend them in the near future.

4 Interpreted code

Our interpreted code consists of contiguous sequences of "synthetic opcodes" and their operands. Syngen can generate ANSI C, but when compiled with gcc it uses C language extensions that make synthetic opcodes pointers to the C code responsible for interpreting that opcode. This "threaded interpreting" entirely eliminates switch dispatch and loop overhead.

To illustrate the above points, here is the assembly language generated for the synthetic opcode that would handle a fully expanded "addl d0,d1" when no CC bit values are required. This is what gcc's 80x86 output looks like (edited for readability) after we run our interpreter through ARDI's Pentium-specific instruction scheduling Perl script:

```
movl    _d0,%eax        ; fetch d0
movl    (%esi),%edi     ; fetch next synthetic opcode
addl    %eax,_d1        ; do the add
addl    $4,%esi         ; increment synthetic PC
jmp     *%edi           ; jump to next synthetic opcode handler
```

We must emphasize that the preceding example is not native code generated by our emulator, but merely a snippet of what gcc generates for our interpreter.

This gives you some idea of the efficiency of the portable component of our emulator.

5 Native code

Syn68k supports optional architecture-specific native code extensions. On systems where they are present, the runtime system tries to generate native code whenever possible. In those rare cases when it cannot, it reverts to our interpreted code. Since syn68k supports both native and synthetic code, the runtime system automatically inserts gateways between the two whenever there is a transition. This approach allows us to gradually phase in native code handlers for most 680x0 instructions while leaving tricky and unimportant rare cases alone.

Although our native code compilation engine is not architecture-specific, to date we have only implemented an 80x86 back end. The 80x86 architecture has achieved such important status in the industry that it makes sense for us to describe how we generate native code for it, even though many of these techniques would not be necessary on RISC architectures.

We are glad that we implemented the most difficult back end first. We believe that, were we to have started with a RISC back end, we would have quite possibly architected a system where retrofitting the exotic mechanisms necessary for efficient 80x86 support was difficult.

Three major problems make translating 680x0 code to 80x86 code difficult:

1. The 80x86 has only 8 registers, while the 680x0 has 16.
2. The 80x86 is little endian, while the 680x0 is big endian.
3. The 80x86 does not have general-purpose postincrement and predecrement operators, which are used frequently in 680x0 code.

On the other hand, several factors make the job easier:

1. The 80x86 has all of the CISC addressing modes commonly used in 680x0 code.
2. The 80x86 has CC bits that map directly to their 680x0 counterparts (except for the 680x0's X bit).
3. The 80x86 supports 8-, 16- and 32-bit operations, (although it can only support 8 bit operations on four of its registers).
4. The 80x86 and 680x0 have analogous conditional branch instructions.
5. The 80x86 allows unaligned memory accesses without substantial overhead.

The toughest problem is the lack of registers. On 32-register RISC architectures it's easy to allocate one RISC register for each 680x0 register, but on the 80x86 a different approach is needed. The obvious solution is to perform full-blown inter-block register allocation, but we fear that using traditional compiler techniques would be unacceptably slow.

For now, we have adopted a simple constraint: between basic blocks, all registers and live CC bits must reside in their canonical home in memory. Within a block, anything goes. So what liberties does syn68k take within a block?

The 80x86 register set is treated as a cache for recently used 680x0 registers, and the 80x86 CC bits are used as a cache for the 680x0 CC bits. At any particular point within a block, each 680x0 register is either sitting in its memory home or is cached in an 80x86 register, and each live 680x0 CC bit is either cached in its 80x86 equivalent or stored in its memory home. Cached registers may be in canonical form, may be byte swapped, may have only their low two bytes swapped, or may be offset by a known constant from their actual value.

Each 680x0 instruction can require that 680x0 registers be cached in particular ways; the compilation engine generates the minimal code needed to satisfy those constraints and then calls a sequence of routines to generate the native code. As each 680x0 instruction is processed, each 680x0 register's cache status is updated. Dirty registers are canonicalized and spilled back to memory at the end of each block (or when we run out of 80x86 registers and we need to make room).

We allow 680x0 registers to be cached with varying byte orders and offsets so that we can perform the optimizations of lazy byte swapping and lazy constant offsetting. If the 680x0 program loads a register from memory and then ends up writing it out later, we avoid unnecessary byte swaps by not canonicalizing the value immediately. Lazy constant offsetting mitigates the overhead of postincrement and predecrement side effects. For example, this 680x0 code:

```
pea     0x1
pea     0x2
pea     0x3
pea     0x4
...
```

becomes this 80x86 code:

```
movl    _a7,%edi
movl    $0x01000000,-4(%edi)    ; "push" big-endian constant
movl    $0x02000000,-8(%edi)
movl    $0x03000000,-12(%edi)
movl    $0x04000000,-16(%edi)
... <more uses of a7 may follow, and they'll use %edi>
subl    $16,%edi
```

```
movl    $edi,_a7
...
```

As mentioned above, we use the 80x86 condition code bits as a cache for the real 680x0 CC bits. Although live cached CC bits are occasionally spilled back to memory because some 80x86 instruction is about to clobber them, this trick almost always works. Using 80x86 CC bits, we can frequently get away with extremely concise code sequences; for example, a 680x0 compare and conditional branch becomes an 80x86 compare and conditional branch.

6 Self-modifying code

Like most dynamically compiling emulators, syn68k doesn't detect self-modifying code; the overhead is too high. Fortunately, self-modifying programs don't work on the real 68040 either. We rely on the program making explicit system calls to flush the caches whenever 680x0 code may have been modified or created. Some programs (like HyperCard) flush the caches very often, which can cause real performance headaches if code is continuously recompiled. We have solved this problem by checksumming 680x0 blocks as they are compiled and only decompiling blocks which fail their checksums. This optimization alone sped up some HyperCard stacks by a factor of three or so.

7 Responsiveness

Responsiveness is a concern for any dynamic compiler. Fortunately, syn68k is largely driven by automatically generated lookup tables so compilation speed is good. Like other dynamic compilers, syn68k only bothers to compile 680x0 code when it encounters it for the first time.

When syn68k encounters new code, it compiles other 680x0 code that it can reach from there but does not compile through jsr's. Only when a jsr is actually executed does syn68k compile the target routine. Once that target routine is compiled, syn68k modifies the jsr handler to point directly to the target routine so that the jsr will be extremely fast the second time it is executed. We've found that lazily compiling through jsr's does a good job of avoiding compilation lag that might annoy the user.

Syn68k does not attempt to generate native code for a basic block until that block (or a nearby one) has been executed 50 times. This saves memory and some compilation time, although we haven't noticed any particular sluggishness when compiling to native code.

8 Other optimizations

Syn68k maintains an internal "jsr stack" to speed up the common case of jsr/rts. We realize that the rts address might have been fiddled with, so the rts handler verifies at runtime that the rts address matches the tag on top of the jsr stack. If it matches, syn68k does a fast jump. If it doesn't, syn68k looks up the code corresponding to the rts address in a hash table.

9 Neat hacks

The low-level code generation routines for the 80x86 back end are machine generated from assembly language templates. Thousands of operand permutations for 80x86 instructions of interest are run through the system's assembler and analyzed to derive the rules the assembler uses to create binaries. Those rules are encapsulated into C code and compiled into syn68k so we can generate binaries on the fly. Here is a sample template:

```
{ "i386_leal_indoff", "", "", "", "", "-",
    "leal %0(%1),%2",
    { "offset", "base", "dst" },
    { { SIZE_32, CONSTANT, IN }, { SIZE_32, REGISTER, IN },
  { SIZE_32, REGISTER, OUT } } },
```

This approach has saved us countless hours of debugging and allows our system to automatically perform the same optimizations as the host system's assembler.

We've annotated our 80x86 descriptions with information about Pentium pairability so that future versions of syn68k can schedule the native code we generate (we already schedule our main interpreter when we build syn68k).

10 Future optimizations

We are working on a simple inter-block register allocation algorithm.

By relocating most 680x0 register to 80x86 register moves to the beginning of each block, we can improve Pentium pairability and reduce 80486 and Pentium address generation pipeline stalls.

Now that we compile to native code, A-line trap overhead is becoming significant. We may soon compile A-line traps to native code that directly calls the appropriate ROMlib routine with the appropriate arguments (checking at runtime to make sure that neither the trap nor the A-line vector has been patched out, of course).

11 Code examples

Here are two sample 680x0 code sequences from real applications, and the 80x86 code that syn68k generates for them. We chose these code sequences specifically to showcase several of the techniques we use, so you shouldn't use them as a substitute for benchmarks. Not all 680x0 code translates as well as these examples do, but these examples are far from exotic.

- Example 1 (Solarian):

```
680x0 code:

addqb   #1,a4@(1)
movel   #0,d0
moveb   a4@,d0
swap    d0
clrw    d0
swap    d0
asll    #2,d0
lea     a5@(-13462),a0
addal   d0,a0
moveal  a0@,a0
movel   #0,d0
moveb   a4@(1),d0
cmpw    a0@,d0
bcs     0x3fffee2

80x86 code:

movl    _a4,%edi                 ; addqb #1,a4@(1)
addb    $0x1,0x1(%edi)
xorl    %ebx,%ebx                ; movel #0,d0
movb    (%edi),%bl               ; moveb a4@,d0
rorl    $0x10,%ebx               ; swap d0
xorw    %bx,%bx                  ; clrw d0
rorl    $0x10,%ebx               ; swap d0
shll    $0x2,%ebx                ; asll #2,d0
movl    _a5,%esi                 ; lea a5@(-13462),a0
leal    0xffffcb6a(%esi),%edx
addl    %ebx,%edx                ; addal d0,a0
movl    (%edx),%edx              ; moveal a0@,a0
xorl    %ebx,%ebx                ; movel #0,d0
movb    0x1(%edi),%bl            ; moveb a4@(1),d0
bswap   %edx                     ; cmpw a0@,d0
```

```
movw    (%edx),%cx
rorw    $0x8,%cx
cmpw    %cx,%bx
movl    %edx,_a0                ; <spill dirty 68k
movl    %ebx,_d0                ;  registers back to memory>
jb      0x6fae0c                ; bcs 0x3fffee2
jmp     0x6faf0c                ; <go to "fall through" code>
```

- Example 2 (PageMaker):

```
680x0 code:

movel   #0,d2
moveb   d0,d2
lslw    #8,d0
orw     d0,d2
movel   d2,d0
swap    d2
orl     d2,d0
movel   a0,d2
lsrb    #1,d2
bcc     0x3fffed4

80x86 code:

xorl    %ebx,%ebx               ; movel #0,d2
movl    _d0,%edx                ; moveb d0,d2
movb    %dl,%bl
shlw    $0x8,%dx                ; lslw #8,d0
orw     %dx,%bx                 ; orw d0,d2
movl    %ebx,%edx               ; movel d2,d0
rorl    $0x10,%ebx              ; swap d2
orl     %ebx,%edx               ; orl d2,d0
movl    _a0,%ecx                ; movel a0,d2
movl    %ecx,%ebx
shrb    %bl                     ; lsrb #1,d2
movl    %ebx,_d2                ; <spill dirty 68k
movl    %edx,_d0                ;  registers back to memory>
jae     0x3b734c                ; bcc 0x3fffed4
jmp     0x43d48c                ; <go to "fall through" 68k code>
```

12 Benchmarks

These performance numbers were computed with Speedometer 3.23. We've removed the floating point tests from the list since they do not measure syn68k's speed. Syn68k contains no special provisions for Speedometer's benchmarks and we believe that these numbers are indicative of syn68k's performance for many other CPU-intensive programs.

Test/CPU	Quadra 610	Pentium 90Mhz	486DX4 75Mhz	486DX/2 66Mhz
CPU	16.018	28.833	15.727	13.840
Dhrystones	19.586	21.886	12.084	9.424
Tower	18.909	27.130	12.235	11.556
Quicksort	17.759	27.105	15.606	13.919
Bubble sort	18.409	31.154	19.286	16.875
Queens	19.083	38.167	19.083	18.320
Puzzle	22.083	44.167	23.661	21.032
Permutations	21.019	28.564	11.604	12.242
Int. Matrix	24.200	26.469	19.369	16.608
Sieve	23.362	60.290	33.982	30.145
Average	20.490	33.881	18.582	16.680

Preliminary analysis suggests that we average a roughly 3:1 instruction count increase when translating to 80x86 code. We have not yet taken rigorous measurements, but the 3:1 figure is lent some credence by the fact that our 75MHz 486DX4 gets nearly the same performance as our Quadra 610. We believe that inter-block register allocation will noticeably improve this ratio.