Portable Target Codes for Compilers

Martin Richards
mr@cl.cam.ac.uk
http://www.cl.cam.ac.uk/users/mr/

University Computer Laboratory
New Museum Site
Pembroke Street
Cambridge, CB2 3QG
Contents of Talk

• Introduction
  – Java Bytecodes
  – BCPL Cintcode
  – Tao Systems Ltd

• Interpreter Design
  – Efficiency Issues
  – Benchmarks tests
  – A Demonstration

• Problems with Modern Architectures

• Obvious Solution
  – Details
  – Effectiveness

• A Few General Observations
class vector {
    int arr[];
    int sum() {
        int la[] = arr;
        int S = 0;
        for (int i=la.length; --i>=0)
            S += la[i];
        return S;
    }
}

Java Byte Code

aload_0  Load this
getfield #10  Load this.arr
astore_1  Store in la
iconst_0
istore_2  Store 0 in S
aload_1  Load la
arraylength  Get its length
istore_3  Store in i

A:  iinc 3 -1  Subtract 1 from i
    iload_3  Load i
    iflt B  Exit loop if < 0
    iload_2  Load S
    aload_1  Load la
    iload_3  Load i
    iaload  Load la[i]
    iadd  Add is S
    istore_2  Store in S
    goto A  Do it again

B:  iload_2  Load S
    ireturn  Return it
SECTION "fact"

GET "libhdr"

LET f(n) = n=0 -> 1, n*f(n-1)

LET start() = VALOF
  { FOR i = 1 TO 8 DO
    writef("f(%i2) = %i6*n", i, f(i))
    RESULTIS 0
  }

... // Entry to: f(n)
28:  L1:
28:  JNE0 L3 J if n ≠ 0
30:  L1 A := 1
31:  RTN Return from f
32:  L3:
32:  LM1 A := -1
33:  AP3 A := A + n
34:  LF L1 B := A; A := f
36:  K4 A := f(n-1)
37:  LP3 B := A; A := n
38:  MUL A := B * A
39:  RTN Return result
...
Cintcode in Binary

Big-Ende Cintcode

... 3E031174 JNE0 +3 L1 RTN
     0FC30CF9 LM1 AP3 LF -7
     0483347B K4 LP3 MUL RTN
     ...

Little-Ende Cintcode

... 7B11033E RTN L1 +3 JNE0
     F90CC30F -7 LF AP3 LM1
     7B348304 RTN MUL LP3 K4
     ...
... 

fetch: ...

switch(B[pc++])
{
    case 0: ...

    case f_mul: a = b*a;
                goto fetch;

    case f_lp3: b = a;
                 a = p[3];
                 goto fetch;

    case 255: ...
}

Martin Richards 8 Seminar 13/11/96
Points to Note

- 256 function codes
- For efficiency keep interpretive overhead small compare to action routine
- Keep the entire interpreter small enough to fit in the on chip cache of the processor
- Most C compilers do a poor job with this code
  - does not contain small simple loops
  - the inner loop contains a computed jump
  - bad for pipelining
  - bad for instruction prefetching
  - bad for jump prediction
fetch:

    movb (%esi),%al
    incl %esi
    jmp *jtbl(,%eax,4)

...

jtbl:

    .long  rl0, rl1, rl2, rl3
    ...
    .long  rl252, rl253, rl254, rl255
    ...


Assembler for the PC

rl52: # mul
mul %ecx, %eax
imul %ebx
movl %eax, %ebx  # a := b * a
movl 36(%esp), %edx  # restore G
movzbl (%esi), %eax
incl %esi
jmp *jtbl(,%eax,4)
...

rl131: # lp3
movl %ebx, %ecx  # b := a
movl 4*3(%ebp), %ebx  # a := p[3]
movb (%esi), %al
incl %esi
jmp *jtbl(,%eax,4)
...
rl52: # mul frq=136949

    mov eax,ecx
    imul ebx
    mov ebx,eax  # a := b * a
    mov edx,[esp+36]  # restore G
    movzx eax,BYTE PTR[esi]
    inc esi
    jmp DWORD PTR[jtbl+4*eax]
...

rl131: # lp3 frq=1059706

    mov ecx,ebx  # b := a
    mov ebx,[ebp+4*3]  # a := p[3]
    mov al,[esi]
    inc esi
    jmp DWORD PTR[jtbl+4*eax]
...
Interpretive Code Design

- Is a byte stream code a good idea?
- For compactness, try to make each byte code equally likely (not easy).
- What operations are most frequent? (Combine common pairs and triples)
- Need statistics from benchmark programs.
Benchmarks

**Bench**

- Smallish compute intensive
- Modelling common operations in an operating system kernel

**BCPL self compilation**

- Larger more realistic application including I/O
- Well understood program
- Executes 22,475,632 Cintcode instructions
- Uses about 200K bytes of Cintcode memory
## Execution Statistics

### Self Compilation Test

<table>
<thead>
<tr>
<th>Count</th>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,059,706</td>
<td>LP3</td>
<td>b := a; a := p[3]</td>
</tr>
<tr>
<td>527,561</td>
<td>LP4</td>
<td>b := a; a := p[4]</td>
</tr>
<tr>
<td>1,406,834</td>
<td>LG n</td>
<td>b := a; a := g[n]</td>
</tr>
<tr>
<td>546,386</td>
<td>JLE l</td>
<td>if b ≤ a goto l</td>
</tr>
<tr>
<td>136,949</td>
<td>MUL</td>
<td>a := b * a</td>
</tr>
<tr>
<td>1,333,284</td>
<td>RTN</td>
<td>procedure return</td>
</tr>
</tbody>
</table>

22,475,632       | Total executions |
Statistics Summary

**Self Compilation Test**

- Load local: 3,809,782
- Store local: 802,744
- Load global: 5,081,621
- Store global: 802,744
- Load positive integer: 4,117,524
- Unconditional jumps: 455,240
- Conditional jumps: 2,152,955
- Jumps on 0: 496,907
- Procedure calls: 1,333,286
- Procedure returns: 1,333,284
- Subscripted load: 1,365,222
- Subscripted store: 598,275
More Statistics

Operand type count

No operand 11,972,904
1 byte integer 6,897,634
2 byte integer 435,405
4 byte integer 0
Direct relative byte 2,853,783
Indirect relative byte 174,870
Forward relative refs 2,469,382
Backward relative refs 559,271

Other Statistics

- Relative address distances
- Local variable offsets
- Distribution of small integer operands
Interpretation of the Statistics

- Statistics should be read with a pinch of salt
  - S3 is executed 27494 times
  - while S2 is only executed 4383 times

- Statistics should be read intelligently and smoothed

Martin Richards
Seminar 13/11/96
**Strategy**

- One byte instructions for common operations
- Multibyte instructions for less common instructions
- Graceful degradation

**Load Integer Instructions**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>LM1</td>
<td>(b := a; a := -1)</td>
</tr>
<tr>
<td>L0</td>
<td>(b := a; a := 0)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>L10</td>
<td>(b := a; a := 10)</td>
</tr>
<tr>
<td>L (n)</td>
<td>(b := a; a := n)</td>
</tr>
<tr>
<td>LH (hh)</td>
<td>(b := a; a := hh)</td>
</tr>
<tr>
<td>LMH (hh)</td>
<td>(b := a; a := -hh)</td>
</tr>
<tr>
<td>LW (www)</td>
<td>(b := a; a := www)</td>
</tr>
</tbody>
</table>
# Cintcode Instructions

<table>
<thead>
<tr>
<th>0</th>
<th>32</th>
<th>64</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>K</td>
<td>LLP</td>
<td>L</td>
<td>LP</td>
<td>SP</td>
<td>AP</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>KH</td>
<td>LLPH</td>
<td>LH</td>
<td>LPH</td>
<td>SPH</td>
<td>APH</td>
</tr>
<tr>
<td>2</td>
<td>BRK</td>
<td>KW</td>
<td>LLPW</td>
<td>LW</td>
<td>LPW</td>
<td>SPW</td>
<td>APW</td>
</tr>
<tr>
<td>3</td>
<td>K3</td>
<td>K3G</td>
<td>K3G1</td>
<td>K3GH</td>
<td>LP3</td>
<td>SP3</td>
<td>AP3</td>
</tr>
<tr>
<td>4</td>
<td>K4</td>
<td>K4G</td>
<td>K4G1</td>
<td>K4GH</td>
<td>LP4</td>
<td>SP4</td>
<td>AP4</td>
</tr>
<tr>
<td>5</td>
<td>K5</td>
<td>K5G</td>
<td>K5G1</td>
<td>K5GH</td>
<td>LP5</td>
<td>SP5</td>
<td>AP5</td>
</tr>
<tr>
<td>6</td>
<td>K6</td>
<td>K6G</td>
<td>K6G1</td>
<td>K6GH</td>
<td>LP6</td>
<td>SP6</td>
<td>AP6</td>
</tr>
<tr>
<td>7</td>
<td>K7</td>
<td>K7G</td>
<td>K7G1</td>
<td>K7GH</td>
<td>LP7</td>
<td>SP7</td>
<td>AP7</td>
</tr>
<tr>
<td>8</td>
<td>K8</td>
<td>K8G</td>
<td>K8G1</td>
<td>K8GH</td>
<td>LP8</td>
<td>SP8</td>
<td>AP8</td>
</tr>
<tr>
<td>9</td>
<td>K9</td>
<td>K9G</td>
<td>K9G1</td>
<td>K9GH</td>
<td>LP9</td>
<td>SP9</td>
<td>AP9</td>
</tr>
<tr>
<td>10</td>
<td>K10</td>
<td>K10G</td>
<td>K10G1</td>
<td>K10GH</td>
<td>LP10</td>
<td>SP10</td>
<td>AP10</td>
</tr>
<tr>
<td>11</td>
<td>K11</td>
<td>K11G</td>
<td>K11G1</td>
<td>K11GH</td>
<td>LP11</td>
<td>SP11</td>
<td>AP11</td>
</tr>
<tr>
<td>12</td>
<td>LF</td>
<td>SOG</td>
<td>SOG1</td>
<td>SOGH</td>
<td>LP12</td>
<td>SP12</td>
<td>AP12</td>
</tr>
<tr>
<td>13</td>
<td>LF$</td>
<td>LOG</td>
<td>LOG1</td>
<td>LOGH</td>
<td>LP13</td>
<td>SP13</td>
<td>XPBYT</td>
</tr>
<tr>
<td>14</td>
<td>LM</td>
<td>L1G</td>
<td>L1G1</td>
<td>L1GH</td>
<td>LP14</td>
<td>SP14</td>
<td>LMH</td>
</tr>
<tr>
<td>15</td>
<td>LM1</td>
<td>L2G</td>
<td>L2G1</td>
<td>L2GH</td>
<td>LP15</td>
<td>SP15</td>
<td>BTC</td>
</tr>
<tr>
<td>16</td>
<td>L0</td>
<td>LG</td>
<td>LG1</td>
<td>LGH</td>
<td>LP16</td>
<td>SP16</td>
<td>NOP</td>
</tr>
<tr>
<td>17</td>
<td>L1</td>
<td>SG</td>
<td>SG1</td>
<td>SGH</td>
<td>SYS</td>
<td>S1</td>
<td>A1</td>
</tr>
<tr>
<td>18</td>
<td>L2</td>
<td>LLG</td>
<td>LLG1</td>
<td>LLGH</td>
<td>SWB</td>
<td>S2</td>
<td>A2</td>
</tr>
<tr>
<td>19</td>
<td>L3</td>
<td>AG</td>
<td>AG1</td>
<td>AGH</td>
<td>SWL</td>
<td>S3</td>
<td>A3</td>
</tr>
<tr>
<td>20</td>
<td>L4</td>
<td>MUL</td>
<td>ADD</td>
<td>RV</td>
<td>ST</td>
<td>S4</td>
<td>A4</td>
</tr>
<tr>
<td>21</td>
<td>L5</td>
<td>DIV</td>
<td>SUB</td>
<td>RV1</td>
<td>ST1</td>
<td>XCH</td>
<td>A5</td>
</tr>
<tr>
<td>22</td>
<td>L6</td>
<td>REM</td>
<td>LSH</td>
<td>RV2</td>
<td>ST2</td>
<td>GBYT</td>
<td>RVP3</td>
</tr>
<tr>
<td>23</td>
<td>L7</td>
<td>XOR</td>
<td>RHS</td>
<td>RV3</td>
<td>ST3</td>
<td>PBYT</td>
<td>RVP4</td>
</tr>
<tr>
<td>24</td>
<td>L8</td>
<td>SL</td>
<td>AND</td>
<td>RV4</td>
<td>STP3</td>
<td>ATC</td>
<td>RVP5</td>
</tr>
<tr>
<td>25</td>
<td>L9</td>
<td>SL$</td>
<td>OR</td>
<td>RV5</td>
<td>STP4</td>
<td>ATB</td>
<td>RVP6</td>
</tr>
<tr>
<td>26</td>
<td>L10</td>
<td>LL</td>
<td>LLL</td>
<td>RV6</td>
<td>STP5</td>
<td>J</td>
<td>RVP7</td>
</tr>
<tr>
<td>27</td>
<td>FHOP</td>
<td>LL$</td>
<td>LLL$</td>
<td>RTN</td>
<td>GOTO</td>
<td>J$</td>
<td>STOP3</td>
</tr>
<tr>
<td>28</td>
<td>JEQ</td>
<td>JNE</td>
<td>JLS</td>
<td>JGR</td>
<td>JLE</td>
<td>JGE</td>
<td>STOP4</td>
</tr>
<tr>
<td>29</td>
<td>JEQ$</td>
<td>JNE$</td>
<td>JLS$</td>
<td>JGR$</td>
<td>JLE$</td>
<td>JGE$</td>
<td>STP13</td>
</tr>
<tr>
<td>30</td>
<td>JEQ0</td>
<td>JNEO</td>
<td>JLSO</td>
<td>JGRO</td>
<td>JLEO</td>
<td>JGEO</td>
<td>STP14</td>
</tr>
<tr>
<td>31</td>
<td>JEQ0$</td>
<td>JNEO$</td>
<td>JLSO$</td>
<td>JGRO$</td>
<td>JLEO$</td>
<td>JGEO$</td>
<td>-</td>
</tr>
</tbody>
</table>
To demonstrate

- Speed
- Compactness
- Machine Independence
- Ease of statistics gathering
- Machine independent low level debugging
There are problems

- Assembled binary byte stream machine independent code is not ideal for many modern machines, particularly those that are extremely fast, eg:
  - DEC Alpha
  - Sun’s Ultra Sparc

- On these machine it is difficult to write an efficient byte stream interpreter, because
  - byte access is relatively slow
  - multi-byte immediate operands are expensive
  - big/little ended problems
  - instruction dispatch is difficult to code efficiently
Recall

...  

fetch: ...

switch(B[pc++])
{
  case 0: ...
       ...
  case f_mul:  a = b*a;
               goto fetch;
       ...
  case f_lp3:  b = a;
               a = p[3];
               goto fetch;
       ...
  case 255: ...
}

Martin Richards  24  Seminar 13/11/96
fetch:
# s6 = address of cintcode instruction
# whose bytes are F, A, ...
# ra = address of jtbl

  ldq   t12,0(s6)  # t12= word containing F
  extbl t12,s6,t0 # t0= F
  addq  s6,1,s6   # pc++
  s4addq t0,ra,a0 # a0 = jtbl + 4 * F
  jmp   zero,(a0),L131

jtbl:  # The jump table
  br    L0; br    L1; br    L2; br    L3
  ...
  br    L252; br    L253; br    L254; br    L255
Sources of inefficiency

- Code very sequential
  - Instructions often use operands computed by the previous instruction
- Can take little advantage of simultaneous instruction execution
- Several memory refs in dispatch operation
- Nothing useful to do in delay slots
- The computed jump ruins:
  - The processor pipeline
  - Prefetching
  - Jump prediction
- Multi-byte immediate operands are expensive
- Even single byte access is expensive
Pipelined Alpha Code

Pipelining the interpreter helps

fetch:
# s6 = address of cintcode instruction
# whose bytes are F, A, ...
# t0 = F
# ra = address of jtbl
ldq_u t12,1(s6) # t12= word containing A
addq s6,1,s6 # pc++
s4addq t0,ra,a0 # a0 = jtbl + 4 * F
mov t0,v0 # v0 = F (the op code)
extbl t12,s6,t0 # t0 = A

# s6 = address of byte A
# v0 = F
# t0 = A
# t12= the 64 bit word containing A
jmp zero,(a0),L131
Accessing an immediate operand

16 bit operands

L97:  # lh    frq=75539
    ldq_u t10,1(s6)
    mov s0,s1    # b := a
    extwl t12,s6,s0
    extwh t10,s6,t10
    ldq_u t12,2(s6)    # prefetch
    addq s6,2,s6    # a := H[pc]; pc += 2
    or t10,s0,s0
    extbl t12,s6,t0
    br fetch
Obvious solution

- Use different interpretive codes of different architectures
  - Conventional Cintcode on the 386/486/Pentium
  - Instructions packed into 64 bit words on the DEC Alpha

- The result of compilation should be loadable into either of these forms (or any other).

- The result of compilation should be an internal assembly language
  - Generated by machine, and
  - Read by machine, so:
  - No need to be human readable
SECTION "fact"

GET "libhdr"

LET f(n) = n=0 -> 1, n*f(n-1)

LET start() = VALOF
{ FOR i = 1 TO 8 DO
   printf("f(\%i2) = \%i6*n", i, f(i))
   RESULTIS 0
}
Code for fact

- The opcodes are those of Cintcode plus a few directives eg LAB, STRING, etc

- All encoded as a stream of integers:

```
F257  F256  K7   C102  C97  C99  C116  C32
C32   C32   F281  K7   C102  C32  C32  C32
C32   C32   C32   F278  L1   F62  L3   F17
F123  F278  L3   F15   F195  F12  L1   F4
F131  F52   F123  F281  K7   C115  C116  C97
C114  C116  C32  C32   F278  L4   F17  F163
F278  L6   F131  F12  L1   F9   F169  F131
F168  F280  M1   F36    G70  F17  F195  F163
F24   F156  L6   F16   F123  F261  M1   K13
C102  C40   C37  C105  C50  C41  C32  C61
C32   C37   C105  C54  C10  F260  K1   G1
L4    G70    F258
```
MODSTART
SECTION K7 C102 C97 C99 C116 C32 C32 C32

//Entry to: f
ENTRY K7 C102 C32 C32 C32 C32 C32 C32
LAB L1
JNEO L3
L1
RTN
LAB L3
LM1
AP3
LF L1
K4
LP3
MUL
RTN
...

Martin Richards
32
Seminar 13/11/96
Second Attempt – SIAL

Code for fact in SASM

SIAL is like CIAL but

- With fewer opcodes and more operands
- Most load operations do not push a to b

MODSTART
SECTION K4 C102 C97 C99 C116

//Entry to: f
ENTRY K1 C102
LAB L1
JNE0 L3
L K1
RTN
LAB L3
LM K1
AP P3
ATB
LF L1
K P4
ATBLP P3
MUL
RTN
...
Observations

- Directives present
- Symbolic labels
- Does not specify how the interpretive instructions are to be represented
- Freedom for the loader to encode the instructions in a form appropriate for the target machine
- The loader and interpreter must cooperate with each other.
Compaction of SIAL

- Split SIAL into different streams:
  - opcodes
  - local variable offsets
  - symbolic labels
  - global variable number
  - string constant characters
  - etc

- Compact each stream by a method appropriate for that stream
Compaction of Opcode Stream

- Use Huffman style variable length encoding bases on static frequency counts.
- But first preprocess the stream to provide compact representations of some repetitive patterns such as
  \[ \text{LAB LAB LAB } \ldots \]
  and
  \[ \text{LP SP LP SP } \ldots \]
Compaction of Label Stream

• Symbolic labels are numbered.
• In practice, they are close to being a monotonic increasing sequence
• Take first differences, plus a few other tricks
• Then encode Huffman style
Compaction of Character Stream

- The stream is typically too short to take much advantage of context so Lempel-Zif or ZIP style compaction
  - Uses too much context
  - Decoder too large
- However single character context is helpful
- Consecutive letters and often in the same case
Compaction of Globals Stream

- Although the Global Vector is peculiar to BCPL, the stream of global numbers is similar to:
  - references to static variables
  - references to variables in FORTRAN Blank COMMON
  - references to method functions in an Object oriented languages

- Compaction can use techniques similar to those used in cache stores
  - a global once referenced is likely to be referenced again
SIAL Compaction Results

BCPL Compiler size

Cintcode: 26184 bytes
Compacted SIAL: 18007 bytes

Raw SIAL compacted by:

compress: 35570 bytes
gzip: 27213 bytes
DJW’s bred: 23144 bytes

Sorted SIAL compacted by:

compress: 36047 bytes
gzip: 22912 bytes
DJW’s bred: 19243 bytes
1. Copy Compressed SIAL to memory
2. Allocate vector for label values
3. Repeatedly scan SIAL until label values are known
4. Allocate vector for assembled SIAL
5. Final pass to assemble code into this vector
Observations

- Number of passes is very dependent on:
  - the treatment of variable length relative addresses
  - cunningness of the loader in label value prediction
- Compacted SIAL can be loaded into Cintcode form in just one pass
Code for a 64 bit machine

- Choose 64 bit granularity for labels
- Choose 64 bit granularity for procedure call return addresses
- Use Huffman style encoding of instruction opcodes
  - No need to use 8 bit opcodes any more
- Constrain immediate operands to lie in current instruction word
- Pad the right hand end with 2 or 3 zero bits
- Instructions like RTN and J, which must be the last in a 64 bit word, can be quite long provided they typically contain several leading zeros
Corresponding Interpreter

fetch:
    switch( instrs & mask)
    { case 0: instrs = *pc++;
        goto fetch;
        ...
        case f_:...: action code
            instrs <<= shift;
            goto fetch;
        ...
    }

Summary

- Compacted SIAL is indeed compact
- It is machine independent
- No big-/little- ended problem
- Can be interpreted more efficiently than Cintcode
Postscript

- The small program that generated human readable SIAL can easily be modified to generate native assembly language.

- For Gnu assembly under Linux this took 2 days.

- The resulting code ran Bench about 10 times faster and the self compilation test about 4 time faster.
Fact.b in assembler

# MODSTART
# SECTION K4 C102 C97 C99 C116

# Entry to: f
# ENTRY  K1 C102
# LAB     L1

LA1:
    movl   %ebp,0(%edx)
movl   %edx,%ebp
popl   %edx
movl   %edx,4(%ebp)
movl   %eax,8(%ebp)
movl   %ebx,12(%ebp)
# JNE0   L3
orl    %ebx,%ebx
jne    LA3
# L      K1
movl   $1,%ebx
# RTN
movl   4(%ebp),%eax
movl   0(%ebp),%ebp
jmp    *%eax
Fact.b in assembler (cont.)

# LAB    L3
LA3:
# LM     K1
    movl $-1,%ebx
# AP     P3
    addl 12(%ebp),%ebx
# ATB
    movl %ebx,%ecx
# LF     L1
    leal LA1,%ebx
# K      P4
    movl %ebx,%eax
    movl %ecx,%ebx
    leal 16(%ebp),%edx
    call *%eax
# ATBLP  P3
    movl %ebx,%ecx
    movl 12(%ebp),%ebx
# MUL
    movl %ecx,%eax
    imul %ebx
    movl %eax,%ebx
# RTN
    movl 4(%ebp),%eax
    movl 0(%ebp),%ebp
    jmp *%eax
...