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

Java Fragment

```
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
```

A BCPL Program

```
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
}
```

Its Cintcode Compilation

```
...
//          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-Ender Cintcode

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

Little-Ender Cintcode

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

Interpreter in C

```
        ...
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:       ...
      }
```

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

Assembler for the PC

```
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                     frq=136949
       movl %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                     frq=1059706
       movl %ebx,%ecx            # b := a
       movl 4*3(%ebp),%ebx       # a := p[3]
       movb (%esi),%al
       incl %esi
       jmp *jtbl(,%eax,4)
       ...
```

TASM Version

```
rl152: # 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.

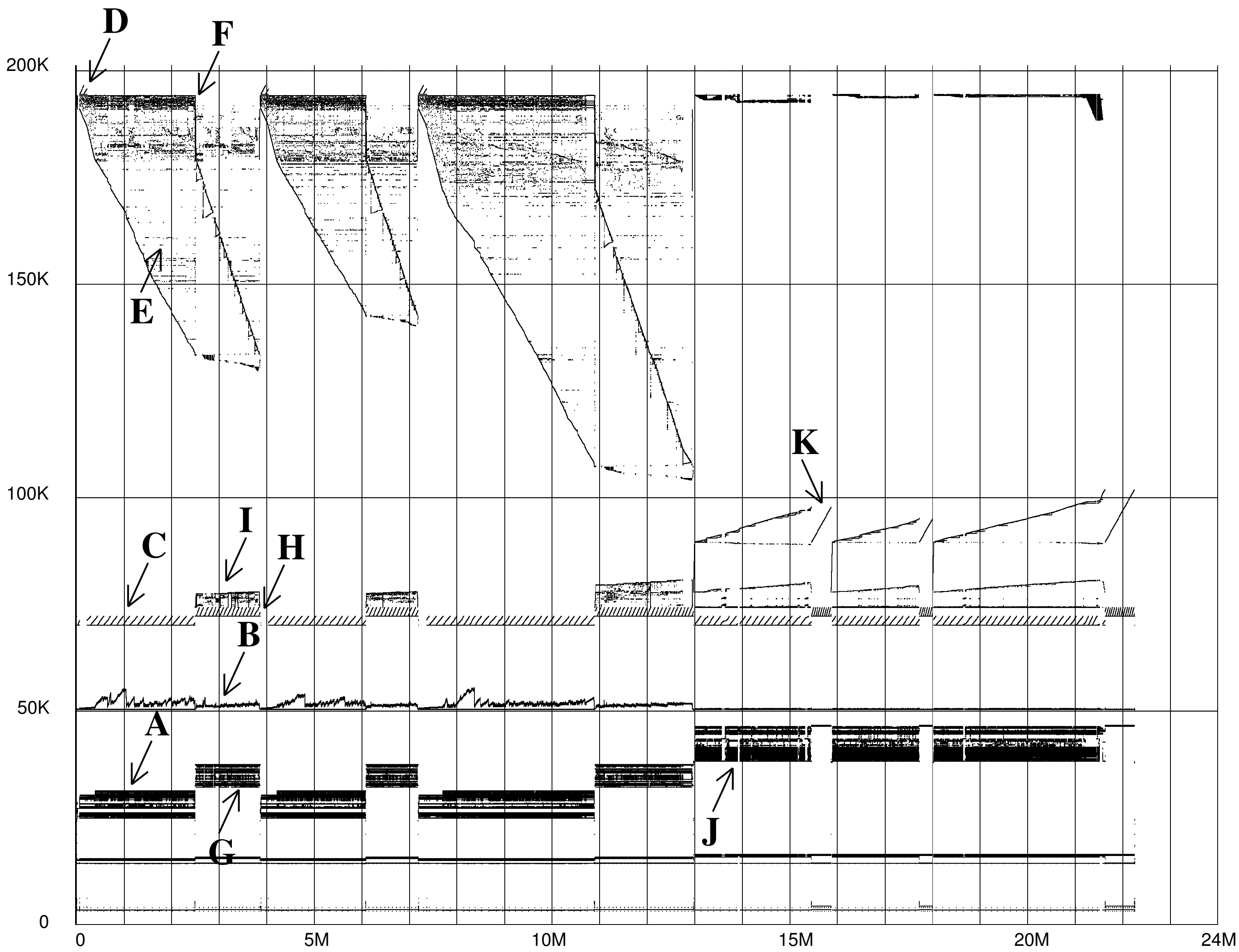
Benchmarks

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


D
F
E
K
C
I
H
B
A
G
J
200K
150K
100K
50K
0
0
5M
10M
15M
20M
24M

Execution Statistics

Self Compilation Test

Count	Instruction	Meaning
1,059,706	LP3	b := a; a := p[3]
527,561	LP4	b := a; a := p[4]
1,406,834	LG n	b := a; a := g[n]
464,778	SP3	p[3] := a
546,386	JLE l	if b $\leq$ a goto l
136,949	MUL	a := b * a
1,333,284	RTN	procedure return
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 oparands

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

Code Design

Strategy

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

Load Integer Instructions

```
LM1          b := a; a := -1
 L0          b := a; a := 0
...
L10          b := a; a := 10
  L   n      b := a; a := n
 LH   hh     b := a; a := hh
LMH   hh     b := a; a := -hh
 LW   wwww   b := a; a := wwww
```

Cintcode Instructions

	0	32	64	96	128	160	192	224
0	-	K	LLP	L	LP	SP	AP	A
1	-	KH	LLPH	LH	LPH	SPH	APH	AH
2	BRK	KW	LLPW	LW	LPW	SPW	APW	AW
3	K3	K3G	K3G1	K3GH	LP3	SP3	AP3	L0P3
4	K4	K4G	K4G1	K4GH	LP4	SP4	AP4	L0P4
5	K5	K5G	K5G1	K5GH	LP5	SP5	AP5	L0P5
6	K6	K6G	K6G1	K6GH	LP6	SP6	AP6	L0P6
7	K7	K7G	K7G1	K7GH	LP7	SP7	AP7	L0P7
8	K8	K8G	K8G1	K8GH	LP8	SP8	AP8	L0P8
9	K9	K9G	K9G1	K9GH	LP9	SP9	AP9	L0P9
10	K10	K10G	K10G1	K10GH	LP10	SP10	AP10	L0P10
11	K11	K11G	K11G1	K11GH	LP11	SP11	AP11	L0P11
12	LF	S0G	S0G1	S0GH	LP12	SP12	AP12	L0P12
13	LF$	L0G	L0G1	L0GH	LP13	SP13	XPBYT	S
14	LM	L1G	L1G1	L1GH	LP14	SP14	LMH	SH
15	LM1	L2G	L2G1	L2GH	LP15	SP15	BTC	MDIV
16	L0	LG	LG1	LGH	LP16	SP16	NOP	CHGCO
17	L1	SG	SG1	SGH	SYS	S1	A1	NEG
18	L2	LLG	LLG1	LLGH	SWB	S2	A2	NOT
19	L3	AG	AG1	AGH	SWL	S3	A3	L1P3
20	L4	MUL	ADD	RV	ST	S4	A4	L1P4
21	L5	DIV	SUB	RV1	ST1	XCH	A5	L1P5
22	L6	REM	LSH	RV2	ST2	GBYT	RVP3	L1P6
23	L7	XOR	RSH	RV3	ST3	PBYT	RVP4	L2P3
24	L8	SL	AND	RV4	STP3	ATC	RVP5	L2P4
25	L9	SL$	OR	RV5	STP4	ATB	RVP6	L2P5
26	L10	LL	LLL	RV6	STP5	J	RVP7	L3P3
27	FHOP	LL$	LLL$	RTN	GOTO	J$	ST0P3	L3P4
28	JEQ	JNE	JLS	JGR	JLE	JGE	ST0P4	L4P3
29	JEQ$	JNE$	JLS$	JGR$	JLE$	JGE$	ST1P3	L4P4
30	JEQ0	JNE0	JLS0	JGR0	JLE0	JGE0	ST1P4	-
31	JEQ0$	JNE0$	JLS0$	JGR0$	JLE0$	JGE0$	-	-

Demonstration

To demonstrate

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

BUT . . .

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 intepreter, because
  - byte access is relatively slow
  - multi-byte immediate operands are expensive
  - big/little ended problems
  - instruction dispatch is difficult to code efficienctly

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:            ...
      }
```

Alpha AssemblyCode

```
fetch:
# s6 = address of cintcode instruction
#               whose bytes are F, A, ...
# ra = address of jtbl
  ldq_u  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
  - Convential 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

Recall fact.b

```
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
}
```

First Attempt – CIAL

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					

More Readable Form of CIAL

Conversion to CASM

```
MODSTART
SECTION K7 C102 C97 C99 C116 C32 C32 C32

//Entry to: f
ENTRY   K7 C102 C32 C32 C32 C32 C32 C32
LAB     L1
JNE0    L3
L1
RTN
LAB     L3
LM1
AP3
LF      L1
K4
LP3
MUL
RTN
...
```

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

  `LAB LAB LAB ...`

  and

  `LP SP LP SP ...`

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

Loading Compressed SIAL

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

The End

BUT . . .

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
  ...
```