Code Density Concerns for New Architectures

Vincent M. Weaver
Cornell University

Sally A. McKee
Chalmers University of Technology

7 October 2009
Introduction

• Benchmark ported to 21 different assembly languages
• Hand-optimized for minimum size
• Code tested and works on all architectures

What ISA features lead to high code density?

Can this help designers of new ISAs?
New ISAs? Really?

- ISA design still a concern
- FPGAs make it easy
- Embedded architectures want dense code
- Linux has 12 embedded architectures and counting
Benefits of Code Density

- L1 iCache holds more instructions
- More data fits in unified L2 cache
- Less bandwidth required to memory and disk
- Fewer TLB misses
- Compact loops can be executed from instruction buffer
- Smaller cache footprint can lead to energy savings
What about Performance?

• Hard to optimize performance
• Varies across implementations
• Dense code often performs well
The Benchmark

LZSS decompression
System calls, including disk I/O
String manipulation and search
Integer to ASCII conversion
ISA Categories

- VLIW
- CISC
- RISC
- Embedded
- 8/16-bit
VLIW Processors

ia64

- 16-byte bundle holds 3 instructions
- Instruction has 3 arguments
- Hundreds of integer registers
- Predication
CISC Processors

m68k, s390, VAX, x86, x86_64

- Variable instruction length, 1-54 bytes
- Instruction has 2 arguments
- 16 integer registers (x86 has 8)
- Status flags
- Unaligned loads
- Complex addressing modes
RISC Processors

Alpha, ARM, m88k, microblaze, MIPS, PA-RISC, PPC, SPARC

- 4 byte instruction length
- Instruction has 3 arguments
- 32 integer registers (except ARM, SPARC)
- Most have a zero register
- Many have branch delay slot
Embedded Processors

avr32, crisv32, sh3, ARM Thumb

- 2 byte instruction length
- Instruction has 2 arguments
- Most have 16 integer registers
- Auto-incrementing loads
- Status flags
8/16-bit Processors

6502, PDP-11, z80

- Variable instruction length (1-6 bytes)
- Instruction has 1-2 arguments
- Status flags
Results – LZSS Decompression

The bar chart shows the results for different processors and instruction sets, categorized by VLIW, RISC, CISC, embedded, and 8/16-bit architectures. The x-axis represents the processors and instruction sets, including ia64, alpha, mips, parisc, sparc, mblaze, 6502, m88k, arm, s390, ppc, pdp-11, vax, z80, m68k, thumb, avr32, sh3, x86, 64, crisv32, and i386. The y-axis represents the number of bytes, ranging from 0 to 256.
Results – String Concatenation

- ia64
- alpha
- sparc
- mips
- parisc
- m68k
- mblaze
- 6502
- arm
- ppc
- vax
- thumb
- sh3
- avr32
- m68k
- s390
- z80
- crisv32
- x86_64
- pdp-11
- i386

Bytes:
- 64
- 48
- 32
- 16
- 8
- 4
- 2
- 1

Categories:
- VLIW
- RISC
- CISC
- embedded
- 8/16-bit

Cornell University
Computer Systems Laboratory

CHALMERS
Results – Integer → ASCII

- ia64
- parisc
- 6502
- alpha
- arm
- m88k
- sparc
- z80
- sha
- ppc
- mips
- thumb
- mblaze
- m68k
- crisv32
- pdp-11
- s390
- avr32
- vax
- x86_64
- i386

Bar chart showing the number of bytes for different processors, with categories for VLIW, RISC, CISC, embedded, 8/16-bit, and No HW divide.
## Correlations

<table>
<thead>
<tr>
<th>Correlation Coefficient</th>
<th>Architectural Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9381</td>
<td>Smallest possible instruction length</td>
</tr>
<tr>
<td>0.9116</td>
<td>Low number of integer registers</td>
</tr>
<tr>
<td>0.7823</td>
<td>Low Virtual address of first instruction</td>
</tr>
<tr>
<td>0.6607</td>
<td>Architecture lacks a zero register</td>
</tr>
<tr>
<td>0.6159</td>
<td>Low Bit-width</td>
</tr>
<tr>
<td>0.4982</td>
<td>Few operands in each instruction</td>
</tr>
<tr>
<td>0.3854</td>
<td>Hardware divide in ALU</td>
</tr>
</tbody>
</table>
## More Correlations

<table>
<thead>
<tr>
<th>Correlation Coefficient</th>
<th>Architectural Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3653</td>
<td>Unaligned load/store available</td>
</tr>
<tr>
<td>0.3129</td>
<td>Year the architecture was introduced</td>
</tr>
<tr>
<td>0.2521</td>
<td>Hardware status flags (zero/overflow/etc.)</td>
</tr>
<tr>
<td>0.2121</td>
<td>Auto-incrementing addressing scheme</td>
</tr>
<tr>
<td>0.0809</td>
<td>Machine is big-endian</td>
</tr>
<tr>
<td>0.0021</td>
<td>Branch delay slot</td>
</tr>
</tbody>
</table>
Results – C Comparison (x86/Linux)

Size (bytes)

~500kB

GLIBC / STATIC GLIBC / DYNAMIC uCLIBC SYSCALL ONLY ASM
What is holding back the C version?

- Stack frame (Calling convention)
- Pointer aliasing
- Full program register allocation
- Constant loading optimizations
- String instructions
Related Work

- RISC Code Compression
- Kozuch and Wolfe – investigate VAX, MIPS, SPARC, m68k, RS6000, PPC
- Hasegawa et al. – gcc generated code on m68k, x86, i960, Sparclite, SPARC, MIPS, AMD29k, m88k, Alpha, RS6000
- Flynn et al. – synthetic architectures
Conclusions / Future Work

- New ISAs are continually being developed; code density is still a concern
- Short instruction codings are key
- High code density requires co-operation of ISA, operating system, system libraries, and compiler
- More architectures should be investigated, as well as more and larger benchmarks
Questions?

All code is available:

http://www.deater.net/weave/vmwprod/asm/ll