LLVM performance optimization for z Systems

Dr. Ulrich Weigand
Senior Technical Staff Member
GNU/Linux Compilers & Toolchain

Date: Mar 27, 2017
Agenda

• LLVM on z Systems Performance History
• Instruction-Set Architecture (ISA) Optimization
• Processor Micro-Architecture Optimization
• Outlook
Performance History
LLVM on z Systems – performance history

Performance vs. GCC pre-7.0 (Mar 2017)

-25% -20% -15% -10% -5% 0%

LLVM Version

3.3 3.4 3.5.0 3.6.0 3.7.0 3.8.0 3.9.0 4.0.0
Back-end changes: overview

• LLVM 3.3 – Initial release
  – No focus on performance; z10 only

• LLVM 3.4 – ISA exploitation
  – Significantly improved z10 code generation; initial z196 & zEC12 support

• LLVM 3.5 – Further ISA exploitation
  – More z196 & zEC12 instructions exploited

• LLVM 3.6 – No performance-related changes

• LLVM 3.7 – z13 vector ISA support

• LLVM 3.8 – Only minor performance-related changes
  – Small improvements to floating-point code generation

• LLVM 3.9 – Misc. code-gen changes / start of micro-arch tuning
  – Avoid false FPR dependencies, conditional sibcall/return, FP test data class

• LLVM 4.0 – Focus on micro-architecture tuning
  – Post-RA scheduler, tune loop unrolling / strength reduction, tune load-on-condition
Instruction-Set Architecture Optimizations
z Systems instruction-set architecture overview

• **z/Architecture publicly documented by IBM**
  – z/Architecture Principles of Operation (SA22-7832-10)

• **Successor to prior architectures going back to 1960s**
  – System/360
  – System/370
  – System/370 extended architecture (370-XA)
  – Enterprise Systems Architecture/370 (ESA/370)
  – Enterprise Systems Architecture/390 (ESA/390)

• **Updated for each new processor generation**
  – Eighth Edition: z10
  – Ninth Edition: z196
  – Tenth Edition: zEC12
  – Eleventh Edition: z13
z/Architecture overview

• Register file
  – 16 64-bit general-purpose register
  – 16 64-bit floating-point registers
  – 32 128-bit vector registers (overlapping FPRs)
  – 16 32-bit access registers
  – Program Status Word (incl. PC and condition code)

• Instruction set
  – >1000 basic instructions, >2000 extended mnemonics
  – CISC operations (reg/reg, reg/mem, mem/mem, …)
  – IEEE floating-point, decimal FP, hexadecimal FP
  – Vector general, integer, floating-point, string instructions
z/Architecture: high-word register operations

- **64-bit GPRs treated as two independent 32-bit parts**
  - Intended to provide register relief (32 “registers” for many operations)

- **For example, to add an immediate:**
  - AGFI %r2, 1 → add 1 to full 64-bit register (64-bit ISA)
  - AFI %r2, 1 → add 1 to low 32-bit part (legacy 32-bit ISA)
  - AIH %r2, 1 → add 1 to high 32-bit part (high-word facility)

- **Modeled as sub-registers in LLVM**
  - GR64 → 64-bit GPRs
  - GR32 → 32-bit lower half GPRs
  - GRH32 → 32-bit upper half GPRs
  - GRX32 → union of GR32 and GRH32
- Used in ISEL for operations supported on both halves
- Post-RA expander selects final instruction depending on register
- AFIMux (GRX32) pseudo → AFI or AIH
z/Architecture: high-word register operations

- **Difficult to implement: instructions with two registers**
  - E.g. COMPARE could be modeled as CMux (GRX32, GRX32)
  - After register allocation, all four combinations possible
  - But ISA only has instructions for three of them:
    - Low/Low → CR
    - High/Low → CHLR
    - High/High → CHHR
  - Special handling for Low/High case required
    - May be convertible to High/Low by updating all users
    - Otherwise 2-3 instruction sequence involving rotates

- **Even more difficult: ADD with three register operands**
  - Only 3 combinations supported in ISA: LLL, HHL, HHH
  - Some cases would require up to 4 instructions to emulate

- **Should ideally be handled in RA directly (like GCC “alternatives”)**
  - But LLVM RA deliberately makes no instruction selection choices ...
### z/Architecture: conditional instructions

- **Condition code – two bits in the PSW**
  - Comparable to flags bits, but used as single value
  - Instructions may set any CC value, no fixed semantics
  - Branch instructions may test for any CC combination

<table>
<thead>
<tr>
<th>Instruction examples</th>
<th>CC 0</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPARE (integer)</td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>-</td>
</tr>
<tr>
<td>COMPARE (floating-point)</td>
<td>Equal</td>
<td>Low</td>
<td>High</td>
<td>Unordered</td>
</tr>
<tr>
<td>ADD</td>
<td>Zero</td>
<td>&lt; Zero</td>
<td>&gt; Zero</td>
<td>Overflow</td>
</tr>
<tr>
<td>ADD LOGICAL</td>
<td>Zero, no carry</td>
<td>Not zero, no carry</td>
<td>Zero, carry</td>
<td>Not zero, carry</td>
</tr>
<tr>
<td>AND</td>
<td>Zero</td>
<td>Not zero</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FIND LEFTMOST ONE</td>
<td>No one bit found</td>
<td>-</td>
<td>One bit found</td>
<td></td>
</tr>
<tr>
<td>TEST UNDER MASK LOW</td>
<td>All zeros</td>
<td>Mixed, left bit zero</td>
<td>Mixed, left bit one</td>
<td>All ones</td>
</tr>
<tr>
<td>VECTOR COMPARE EQUAL</td>
<td>All elements equal</td>
<td>Some elts. equal</td>
<td>-</td>
<td>No elements equal</td>
</tr>
<tr>
<td>CONVERT UTF-8 TO UTF-32</td>
<td>Data processed</td>
<td>Destination full</td>
<td>Invalid UTF-8</td>
<td>Early exit</td>
</tr>
</tbody>
</table>
z/Architecture: more conditional instructions

- **Instructions using the condition code**
  - LOAD ON CONDITION
    - Load from memory/register/immediate if CC in mask
  - (Conditional) trap instruction
    - Special form of (conditional) branch with invalid target

- **Instructions that do not use the CC**
  - COMPARE AND BRANCH / TRAP
    - Compare + conditional branch (or trap) as single insn
  - BRANCH ON COUNT
    - Decrement register and branch if not zero
  - LOAD AND TRAP
    - Load register from memory and trap if zero
**LLVM code generation for conditional instructions**

<table>
<thead>
<tr>
<th>LLVM pass</th>
<th>z/Architecture ISA handling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction selection</td>
<td>Select appropriate compare instructions</td>
</tr>
<tr>
<td></td>
<td>Generate TEST UNDER MASK</td>
</tr>
<tr>
<td>Pre-RA pseudos</td>
<td>Generate LOC from selects</td>
</tr>
<tr>
<td>Early if-conversion</td>
<td>Generate LOCR from if blocks</td>
</tr>
<tr>
<td></td>
<td>Speculative execution of both sides</td>
</tr>
<tr>
<td>Peephole optimizer</td>
<td>Optimize explicit uses of CC (e.g. builtins)</td>
</tr>
<tr>
<td></td>
<td>Generate LOCHI for immediates</td>
</tr>
<tr>
<td>Post-RA pseudos (including z specific pass)</td>
<td>Select low/high instructions</td>
</tr>
<tr>
<td></td>
<td>Expand mixed LOCRMux cases</td>
</tr>
<tr>
<td>Late if-conversion</td>
<td>Detect conditional trap, conditional return,</td>
</tr>
<tr>
<td></td>
<td>conditional sibling call</td>
</tr>
<tr>
<td>Optimize comparison against zero (z specific</td>
<td>Detect branch-on-count, load-and-trap</td>
</tr>
<tr>
<td>pass)</td>
<td>Convert load to load-and-test</td>
</tr>
<tr>
<td></td>
<td>Update CC users with CC mask for other insn</td>
</tr>
<tr>
<td>Fuse compare operations (z specific pass)</td>
<td>Detect compare-and-branch, compare-and-trap,</td>
</tr>
<tr>
<td></td>
<td>compare-and-return, compare-and-sibcall</td>
</tr>
</tbody>
</table>
Processor Micro-Architecture Optimizations
z13 processor micro-architecture overview

- Full micro-architecture is not publicly documented
- Overview targeted at compiler developers here:
  - IBM z Systems Processor Optimization Primer
z13 high-level instruction & execution flow
z13 execution engine pipelines

Only 1 of 2 issue sides shown
- Typical pipeline depths and bypass capabilities shown
- Some instructions may take longer to execute or bypass results
- Access registers not shown

ACC – GR access
WB – GR write back

V-ACC – FPR/VR access
VWB – FPR/VR write back

CC – condition code calculation

BYP – data bypass network cycle

FPD, DFU – functions, e.g. divide, square-root, may take multiple passes through the pipeline

G2F – GR to VR/FPR moves
F2G – VR/FPR to GR moves
Instruction scheduling

• **Goals of scheduling for z13**
  – **No** exact modeling of OOO execution phase possible
    • But: execution latencies still determine critical path length
  – Optimize decoder grouping
    • Sequence instructions so that decoder groups can be as large as possible (3 instructions) to optimize dispatch bandwidth
  – Resource balancing
    • Sequence instructions so that over time, usage of the various execution units is as evenly balanced as possible
  – FPd side steering
    • Distribute long-running instructions (e.g. FP divide) evenly over both execution pipeline sides
  – FXU side steering
    • Distribute dependent instructions to the same side to enable result bypassing with reduced latency
Instruction scheduling

• **Current LLVM implementation**
  – Post-RA scheduler as very last MI pass
    • Using new SchedStrategy and HazardRecognizer
    – Decoder grouping, resource balancing, FPd steering
    – FXU steering not yet implemented due to regressions

• **Future opportunities**
  – Pre-RA MI scheduler
    • “Mix up” register usage to get more freedom post-RA
      – Better decoder grouping; better FXU side steering; ...
    • But must be careful to not cause spilling!
    • No implementation without regressions so far …
    • Area of active research in compiler theory
  – Global scheduling across block boundaries?
Tuning code generation

• **Caveat: performance results hard to predict**
  – Positive effects often dominated by negative second-order issues
  – E.g. increasing use of branch-on-count caused overall performance regression
  – Important goal: tune to avoid “second-order” effects

• **Loop unrolling**
  – Important to eliminate small loops which are sensitive
    • Loops should preferably be >12 instructions
  – Enables "everything" to get rid of small loops, including forced unrolling
  – But: limit on number of stores to avoid running out of store tags

• **Loop strength reduction**
  – z13 supports only 12-bit unsigned offsets for vector memory accesses
  – Scalar code uses 20-bit signed offsets → try to avoid regressions in vectorizer
  – New hook isFoldableMemAccessOffset() to handle this

• **WIP: Cost functions to tune vectorizer decisions**
Outlook
Status and outlook

• Aside: non-performance related work in 2016
  – Profile-directed feedback support (compiler-rt)
  – Support for address sanitizer (via BountySource)
  – Added LLDB support
  – Swift & Rust enablement

• Future work
  – Support next-generation z Systems processor
    • Twelfth Edition of the z/Architecture
    • GCC and binutils support already available
  – Improved scheduling / micro-architecture tuning
  – Ongoing benchmark analysis efforts
LLVM on z Systems – optimization opportunities

-10% -8% -6% -4% -2% 0% 2% 4% 6% 8% 10%

LLVM 4.0 vs. GCC pre-7.0 (Mar 2017)

Questions