Dynamic Binary Translation for Generation of Cycle Accurate Architecture Simulators

Institut für Computersprachen
Technische Universität Wien
Austria

Andreas Fellnhofer  Andreas Krall  David Riegler

Part of this work was supported by OnDemand Microelectronics and the Christian Doppler Forschungsgesellschaft
Overview
Previous Projects

- incremental compilers
- static assembly language instrumentation
- Molecule (ATOM clone)
- STonX
- CACAO
- bintrans
- reverse compilation (DSP VLIW to C)
- compiled cycle accurate instruction set simulation
- iboy
STonX

- AtariST on X windows
- generated 68k instruction set emulator
- work on binary translation unfinished
CACAO

- JIT-only Java Virtual Machine
- ultra-fast basic compiler
- recompilation with optimizations
- on-stack replacement
- deoptimization when assumptions become invalid
bintrans

- generator for user mode binary translators
- LISP like source and target architecture specification
- direct translation of blocks/traces without intermediate representation
- hybrid fixed register mapping and local register allocation
- local register liveness analysis with global propagation between different runs
- 1.8 to 2.5 overhead compared to native code
iboy

- gameboy emulator for iPod
- full system level cycle accurate emulation
- uses only dynamic binary translation
- self modifying code leads to recompilation of basic block
- template based generated compiler
- local flag constant propagation and liveness analysis
- ROM/RAM code caches
Overview
Architecture Description Language

- mostly structural
- xml syntax
- graphical user interface available
- no redundant information
- MIPS R2000 specification is about 1000 lines
<Operation name="addu" syntax="op3_s" >
  <Syntax syntax="op3_s" token="op" value="addu" />
  <Body>
    <add a="Rs_i" b="Rt_i" d="Rd_o" o="overflow" c="carry"/>
  </Body>
</Operation>
Simulator Basics

- generated mixed interpreting/translating simulator
- translation of blocks and traces
- common IR for interpreter and translator
- backend is LLVM just-in-time compiler
- own and LLVM optimizations used
Differences to Standard Binary Translation

- cycle accurate simulator
- full system simulation
- simulation of in-order pipelined architectures
- instructions cross basic block borders
Overlapping Instructions

- **Stage Number**: 1, 2, 3, 4
- **Instructions Executed in Block A**:
  - Cycle 1: A4, inst 1
  - Cycle 2: A3, A4
  - Cycle 3: A2, A3, A4
  - Cycle 4: A1, A2, A3, A4

- **Instructions Executed in Block B**:
  - Cycle 5: B1, B2
  - Cycle 6: inst 3, B1
  - Cycle 7: jump, inst 3, B1
  - Cycle 8: inst 1, jump, inst 3, B1

- **Instructions Which Depend on the Predecessor Basic Block**

---

*Andreas Krall (TU Wien)*

*Dynamic Binary Translation for Generation of Cycle Accurate Architecture Simulators*
Similar Predecessor Blocks

program memory

0x100: add
0x101: add
0x102: store
0x103: sub
0x104: cbranch 0x101

basic block 1

0x100: add
0x101: add
0x102: store
0x103: sub
0x104: cbranch 0x101

basic block 2

0x102: store
0x103: sub
0x104: cbranch 0x101

basic block 3

0x102
0x103
0x104

0x105: store
0x106: add
Basic Block Duplication

program memory

0x100: add
0x101: add
0x102: store
0x103: sub
0x104: cbranch 0x101

basic block 1

0x104: cbranch 0x101

basic block 2a

0x102: store
0x103: sub
0x104: cbranch 0x101

basic block 2b

0x102: store
0x103: sub
0x104: cbranch 0x101

basic block 3a

0x105: store
0x106: add

basic block 3b

0x105: store
0x106: add
Trace Formation

- entry point of trace

- basic block 1
  - successor array
  - bb1
    - pipeline restore info
    - 0: bb2
    - 1: bb3

- basic block 2
  - successor array
  - bb2
    - pipeline restore info
  - 0: bb4
  - 1: bb2

- basic block 3
  - successor array
  - bb3
    - pipeline restore info
    - 0: bb4

- basic block 4
  - successor array
  - bb4
    - pipeline restore info

- exit point for basic block 1
- exit points for basic blocks 2 to 4
Optimizations

- LLVM optimizations (e.g. constant propagation, dead store elimination)
- local copy of global values
- linking of basic blocks
- constant forward optimization
- LLVM JIT very slow (mostly instruction selection)
Empirical Evaluation

Simulation Speed MIPS

Andreas Krall (TU Wien)  Dynamic Binary Translation for Generation of Cycle Accurate Architecture Simulators  October 28, 2008  18 / 1
Simulation Speed CHILI

Andreas Krall (TU Wien) Dynamic Binary Translation for Generation of Cycle Accurate Architecture Simulators October 28, 2008 19 / 1
Breakdown of Simulation Cycles MIPS

- prime
- jpeg
- crc32
- sha
- dijkstra
- bitcount
- blowfish
- stringsearch
- adpcm
- gsm
- rijndael

Trace: purple
BasicBlock: blue
Interpreter: yellow
Empirical Evaluation

Breakdown of Simulation Cycles CHILI

Trace
BasicBlock
Interpreter

prime
crc32
sha
dijkstra
bitcount
blowfish
stringsearch
adpcm
gsm
rijndael

0%
20%
40%
60%
80%
100%
Compile and overall Run Time MIPS
Compile and overall Run Time CHILI

![Bar chart showing compile and run time for various benchmarks.](image-url)
Empirical Evaluation

Performance with increasing Simulation Time

CHILI

Andreas Krall (TU Wien)
Simulation Speed with different Compilation Thresholds
Empirical Evaluation

Optimized Block Linkage MIPS

Andreas Krall (TU Wien)
Optimized Block Linkage CHILI

Andreas Krall (TU Wien)
Local Copy of Global Values MIPS

![Bar Chart]

- prime: 173%
- jpeg: 125%
- crc32: 354%
- sha: 174%
- dijkstra: 241%
- bitcount: 440%
- blowfish: 170%
- stringsearch: 181%
- adpcm: 209%
- gsm: 133%
- rijndael: 142%

**Local Scope Opt.**

**Local Scope**
Local Copy of Global Values CHILI

Andreas Krall (TU Wien) Dynamic Binary Translation for Generation of Cycle Accurate Architecture Simulators

October 28, 2008 29 / 1
Empirical Evaluation

Forwarding Optimization MIPS

Andreas Krall (TU Wien) Dynamic Binary Translation for Generation of October 28, 2008 30 / 1
Conclusion

- cycle accurate simulation rises additional problems
- binary translation is efficient
- LLVM JIT is slow