Low Overhead Debugging with DISE

Marc L. Corliss    E Christopher Lewis    Amir Roth
Department of Computer and Information Science
University of Pennsylvania

Overview

- **Goal:** Low overhead interactive debugging
- **Solution:** Implement efficient debugging primitives
  - e.g. breakpoints and watchpoints
  - using Dynamic Instruction Stream Editing (DISE) [ISCA '03]: General-purpose tool for dynamic instrumentation

Breakpoints and Watchpoints

- **Breakpoint**
  - Interrupts program at specific point
  - `break test.c:100`

- **Watchpoint**
  - Interrupts program when value of expression changes
  - `watch x`

- **Conditional Breakpoint/Watchpoint**
  - Interrupts program only when predicate is true
  - `break test.c:100 if i==93`

Debugging Architecture

- User/debugger transitions
- Debugger/application transitions
  - High overhead
  - May be masked by user/debugger transitions
  - Otherwise perceived as latency

User

`int main()` {
    ...
}

Spurious Transitions

Debugger

Application
Eliminating Spurious Transitions
- Instrument app. with breakpoint/watchpoint logic
  - No spurious transitions
- Static approaches already exist
  - During compilation or post-compilation (binary rewriting)
- We propose dynamic instrumentation
  - Using DISE

Talk Outline
- Introduction
  - Watchpoint implementations
- DISE background
- Watching with DISE
- Evaluation
- Related work and conclusion

Watchpoint Implementations
- Single-stepping
- Virtual memory support
- Dedicated hardware registers

Single-Stepping
- Trap after every statement
  - Easy to implement
  - Poor performance (many spurious transitions)
Virtual Memory Support
Trap when pages containing watched variables written

- Reduces spurious transitions
- Coarse-granularity (still may incur spurious transitions)
- Spurious transitions on silent writes

Dedicated Hardware Registers
Trap when particular address is written

- Reduces spurious transitions
- Spurious transitions on silent writes
- Number and size of watchpoints limited

Conditional Watchpoints
Trap like unconditional, debugger evaluates predicate

- Simple extension of unconditional implementation
- Introduces more spurious transitions

Instrumenting the Application
Embed (conditional) watchpoint logic into application

- Eliminates all spurious transitions
- Adds small overhead for each write
Low Overhead Debugging with DISE – Marc Corliss

**DISE**

- Dynamic Instruction Stream Editing (DISE) [ISCA '03]
  - Programmable instruction macro-expander
  - Like hardware SED (DISE = dynamic instruction SED)
  - General-purpose mechanism for dynamic instrumentation
  - Example: memory fault isolation

```
store r4,8(r9)
```

### DISE Productions

- **Production:** static rewrite rule
  - Parameterized replacement sequence
  - Expansion: dynamic instantiation of production

```
srli r9,4,r1
cmp r1,r2,r1
bne r1,Error
store r4,8(r9)
```

### Watchpoint Production

- Interactive debugger injects production:

```
T.OPCLASS == store
  =>
  T.INST
  # original instruction
  lda ddr,T.IMM(T.RS)  # compute address
  bic ddr,T.drr        # quad align address
  cmpeq ddr,T.drr      # cmp to watched address
  ccall ddr,HNDLR      # if equal call handler
```

Less Overhead Debugging with DISE – Marc Corliss

**Watching with DISE**

- Monitor writes to memory
- Check if watched value(s) modified
  - Requires expensive load(s) for every write
- Optimization: address match gating
  - Split into address check (fast) and value check (slow)
  - Check if writing to watched address
  - If so, then handler routine called
  - Handler routine does value check

```
store r4,8(r9)
```

DISE Productions

- Production: static rewrite rule
- Parameterized replacement sequence
- Expansion: dynamic instantiation of production

```
srli r9,4,r1
cmp r1,r2,r1
bne r1,Error
store r4,8(r9)
```

DISE Productions

- Production: static rewrite rule
- Parameterized replacement sequence
- Expansion: dynamic instantiation of production

```
srli r9,4,r1
cmp r1,r2,r1
bne r1,Error
store r4,8(r9)
```

**Watchpoint Production**

- Interactive debugger injects production:

```
T.OPCLASS == store
  =>
  T.INST
  # original instruction
  lda ddr,T.IMM(T.RS)  # compute address
  bic ddr,T.drr        # quad align address
  cmpeq ddr,T.drr      # cmp to watched address
  ccall ddr,HNDLR      # if equal call handler
```

**Watchpoint Production**

- Interactive debugger injects production:

```
T.OPCLASS == store
  =>
  T.INST
  # original instruction
  lda ddr,T.IMM(T.RS)  # compute address
  bic ddr,T.drr        # quad align address
  cmpeq ddr,T.drr      # cmp to watched address
  ccall ddr,HNDLR      # if equal call handler
```
Other Implementation Issues

- Conditional watchpoints
  - Inline simple predicates in replacement sequence
  - Put complex predicates in handler routine
- Multiple watchpoints/complex expressions
  - For small #, inline checks in replacement sequence
  - For large #, use bloom filter

Key point: **DISE is flexible**

Virtues of DISE

- Versus dedicated hardware registers
  - General-purpose: DISE has many other uses
    - Safety checking [ISCA '03], security checking [WASSA '04], profiling [TR '02], (de)compression [LCTES '03], etc.
  - Efficient: no spurious transitions to the debugger
  - Flexible: more total watchpoints permitted
- Versus static binary transformation
  - Simple-to-program: transformation often cumbersome
  - Efficient: no code bloat, no transformation cost
  - Less intrusive: Debugger and application separate

Evaluation

- Show DISE efficiently supports watchpoints
  - Compare performance to other approaches
- Analyze debugging implementations in general
  - Characterize performance of different approaches

Methodology

- Simulation using SimpleScalar Alpha
  - Modeling aggressive, 4-way processor
- Benchmarks
  - (subset of) SPEC Int 2000
- Watchpoints for each benchmark
  - HOT, WARM1, WARM2, COLD
- Debugger/application transition overhead
  - 100,000 cycles
Low Overhead Debugging with DISE – Marc Corliss

Unconditional Watchpoints

- Single-stepping has slowdowns from 6,000-40,000

Unconditional Watchpoints

- VM sometimes good, sometimes awful
  - Erratic behavior primarily due to coarse-granularity

Unconditional Watchpoints

- Hardware registers usually good (no overhead)
- Hardware registers perform poorly for HOT
  - Significant number of silent writes

Unconditional Watchpoints

- DISE overhead usually less than 25%
Conditional Watchpoints

- In many cases DISE outperforms hardware regs.
  - Spurious transitions for HW regs. whenever WP written
  - DISE/HW registers can differ by 3 orders of magnitude

Conditional Watchpoints

- Instrumentation overhead more consistent
  - Instrumentation adds small cost on all writes
  - Non-instrumentation adds high cost on some writes

Multiple Watchpoints

- For <5 watchpoints can use hardware registers
  - Performance good 1-3, degrades at 4 due to silent writes
- For >4 watchpoints must use virtual memory
  - Performance degrades due to coarse-granularity

Multiple Watchpoints

- For <4 watchpoints DISE/Inlined slightly worse
  - DISE/Inlined much better for >3 watchpoints
Low Overhead Debugging with DISE – Marc Corliss

Multiple Watchpoints

Evaluation Results

• DISE watchpoints have low overhead
  • DISE overhead usually less than 25%
  • In many cases DISE outperforms other approaches
  • Silent writes/conditionals ⇒ spurious transitions
  • DISE flexibility helps keep low overhead in all scenarios
• Overhead of instrumentation more consistent
  • Small cost on all writes rather than occasional large cost
  • Non-instrumentation has 1x to 100,000x slowdown

Related Work

• iWatcher [Zhou et. al ‘04]
  • Hardware-assisted debugger
    • Associates program-specified functions with memory locations
  • Address-based versus instruction-based
    • Not general-purpose mechanism like DISE
  • More significant hardware modifications than DISE
• Other related areas
  • Static transformation [Kessler ’90, Wahbe et al. ’93]
  • Instrumentation mechanisms [Valgrind, ATOM, EEL, Etch]

Conclusion

• DISE effectively supports low overhead debugging
  • Virtues: general-purpose, flexible, simple-to-program, efficient, non-intrusive
  • Characterize interactive debugging implementations
  • Instrumentation has consistently low overhead