DISE: A Programmable Macro Engine for Customizing Applications

Marc Corliss, E Lewis, Amir Roth
University of Pennsylvania
Overview

- **Application customization functions (ACFs)**
  - E.g., safety-checking, debugging, decompression, optimization ...
  - Useful when applied to other programs
  - Why? new platforms γ new requirements: power, safety ...

- **ACF implementation**
  - Binary rewriting (BR): + flexible functionality, – performance
  - Hardware widget (HW): + little performance loss, – ACF specific

- **Dynamic Instruction Stream Editing (DISE)**
  - Combo: a hardware widget that does rewriting
    + Rich functionality, + low cost
DISE

- **Programmable instruction macro-expander**
  - Like a hardware SED (DISE = dynamic instruction SED)
  - Executes rewrite rules (productions) on fetch stream
    - Looks for instructions that match patterns...
    - Replaces them with parameterized sequences (repseq)

- **Orthogonal to μops**
  - μops: finer-granularity, same functionality
  - DISE: same granularity, different functionality

Corliss, Lewis, + Roth ISCA-30
Advantages of This Picture

+ **Generality** (wrt HW)
  - Instruction-based ACFs
  - That can modify (not just observe) application

+ **Performance** (wrt BR)
  - Avoid costs of inserting ACF code into binary

+ **Convenience** (wrt both)
  - Avoid pain of rewriting binary
  - Avoid pain of implementing new widget

+ **Non-subvertability** (new)
  - The last thing that can/will manipulate code, guaranteed
Talk Outline

- Introduction
- Basic Capabilities
- Implementation
- Formulating ACFs
- Numbers
- Related work, Status, Future
What DISE Does

- Example customization functionality
  - Compare upper store address bits to const, ERROR on mismatch

- DISE implementation
  - Pattern
    
    ```
    store *, *(*)  
    // store = match, * = don’t care
    ```

  - Repseq
    
    ```
    srl R9,#4,R1  
    cmp R1,R2,R1  
    bne R1,ERROR  
    T.INSN  
    ```

    ```
    store R4,8(R9)
    ```

    ```
    srl R9,#4,R1  
    cmp R1,R2,R1  
    bne R1,ERROR  
    store R4,8(R9)
    ```
More DISE Capabilities (And Restrictions)

- Capabilities
  - **Dedicated Registers**
    - Scratch storage (no need to scavenge)
    - Inter-repseq communication (synthesize global behavior)
  - **DISEPC**
    - Precise state within repseq’s (repseq’s are atomic)
    - Control-flow within repseq’s
  - **DISAWK?**

- Restriction: peephole transformations only
How DISE Does It

- Three components (well-understood, resemble μop)
  - **Pattern Table (PT)**: small associative table holds pattern specs
  - **Replacement Table (RT)**: bigger RAM holds repseq specs
  - **Instantiation Logic (IL)**: executes directives to produce actual insns

- Logically: DISE then decode
How DISE Really Does It

- Physically: interleave DISE/decode implementations
  - RISC: PT + decoded RT in parallel with native decoder
  - CISC(+$\mu$ops): PT + $\mu$op RT in same physical structures

- Decoded/$\mu$op RT? PT/RT programmed via controller
  - Abstracts PT/RT formats: external DISE is annotated native
  - Virtualizes PT/RT sizes
Talk Outline

- Introduction
- Basic Capabilities
- Implementation
- Formulating ACFs
- Numbers
- Related Work, Status, Future
Formulating ACFs

- Two kinds: transparent and aware (wrt application)

  - **Transparent**: unmodified applications work without DISE
    - DISE redefines “naturally-occurring” instructions
    - Examples: fault isolation (just saw), profiling, etc.

  - **Aware**: modified applications rely on DISE
    - DISE converts “planted” codewords to runtime code
    - Examples: decompression (see soon), dynamic optimization, etc.

- ACFs can be composed
Transparent ACF: Memory Fault Isolation

• [Wahbe+95]: multi-module safety in one address space
  • Why? Extensible kernel, no VM...
  • How? Check upper address bits of loads, stores, ijumps
  • Have seen, but to recap...

• DISE
  R: \texttt{srli T.RS,#4,DR1} \\
  \texttt{cmpeq DR1,DR2,DR1} \\
  \texttt{bne DR1,ERROR} \\
  \texttt{store T.RD,T.IMM(T.RS)}

  + DISE registers \texttt{DR1,DR2}

  + No branch retargeting

• Binary rewriting
  R: \texttt{move T.RS, R3} \\
  \texttt{srli T.RS,#4,R1} \\
  \texttt{cmpeq R1,R2,R1} \\
  \texttt{bne R1,ERROR} \\
  \texttt{store T.RD,T.IMM(R3)}

  + DISE registers \texttt{DR1,DR2}

  + No branch retargeting

  + Scavenge \texttt{R1,R2,R3}

  + Retarget surrounding branches

  + \texttt{move} thwarts malicious jumps
Aware ACF: Decoder Decompression

- [Lefurgy+97]: decoder + dictionary \( \gamma \) compressed I$
  - How? Codewords = reserved opcode + dictionary index

**Uncompressed code**

```plaintext
add R2, R3, R2
load R4, 8(R2)
...
add R1, R3, R1
load R4, 8(R1)
```

**Compressed code**

```plaintext
reserved R2 #0
...
reserved R1 #0
```

**Hardware Widget**

- D0: add R2, R3, R2
  load R4, 8(R2)
- D1: add R1, R3, R1
  load R4, 8(R1)

**DISE: RT = dictionary**

- Exploit parameterized replacement

```plaintext
R0: add T.RS, R3, T.RS
load R4, 8(T.RS)
```

+ More efficient dictionary usage
+ Can compress branches

Corliss, Lewis, + Roth  ISCA-30
Composing ACFs

- Consider: applying both fault isolation and decompression
  - Server compresses, client decompresses
  - Fault isolate uncompressed code

- Who adds fault isolation?
  - Server, before compression
    - How does server know to do this?
  - Client, after decompression
    - Client needs two widgets

- DISE: **client composes the two ACFs to form one**
  - Apply fault isolation productions to decompression repseq’s
  + Two ACFs, one widget (DISE)
Talk Outline

- Introduction
- Basic Capabilities
- Implementation
- Formulating ACFs
- Numbers
- Related Work, Status, Future
Experimental Setup

- Alpha ISA SimpleScalar-based timing simulator
  - 4-way superscalar, 12-stage pipeline, 128 instruction window
  - 32KB primary caches, 1MB L2
  - DISE configuration
    - 32-entry PT (256 bytes), 512-entry RT (3KB)
    - PT/RT miss handler: flush + 30 cycle stall on

- SPEC2K integer benchmarks

- Binary rewriting: perl script on .S files
Fault Isolation: DISE vs. Binary Rewriting

- **Point**: DISE performs better than corresponding BR
- **Metric**: exec. time norm. to 4-wide, 32KB I$, unmodified

- **Exp1**: increase I$ size
  - Footprint cost disappears
  - Bandwidth cost dominates
  - DISE adds fewer instructions
  - I$ size effectively decreasing

- **Exp2**: increase proc. width
  - Dynamic cost disappears
  - Static cost dominates
  - DISE adds no footprint
Decompression: DISE vs. Hardware Widget

- **Point**: DISE more flexible (?) than corresponding HW
- **Metric**: code size norm. to uncompressed

- Exp1: parameterization
  + Parameterization (DISE) helps
  + DISE is general-purpose

- Also in paper: RT sensitivity, composed ACF performance
- More about DISE decompression [Corliss+, LCTES03], Fri. 2:30
Related Work (not exhaustive)

- Translation/emulation
  - Hardware: X86 μOPs, AMD ROPs.
  - DISE is native-to-native and programmable
  - Software: FX!32, DAISY [Ebcioğlu+], Crusoe

- ACF tools
  - Hardware: PipeRench [Goldstein+], I-COP [Chou+]
  - DISE is before execute
  - Software (offline): ATOM [Eustace+], Etch [Romer+], EEL [Larus+]
  - Software (online): DELI [Desoli+]
  - Most similar in spirit (and name) to DISE
To Summarize

• **DISE: a HW/SW solution for customizing applications**
  - Like a hardware SED or AWK
  - More general than hardware
  - Faster than software
  - Has some other nice properties (e.g., non-subvertability)

• Near future: apps to exploit “other nice properties”
  - Non-subvertability \( \gamma \) security applications
  - Painless code modification \( \gamma \) runtime optimization

• Your questions