Reference Monitors/Information
Flow Tracking

Slide credits: Raksha presentation based on that original authors
Reference Monitors

- A class of analyses that monitors application code to enforce some invariants

- What kinds of invariants? We’ve seen at least one
  - Control Flow Integrity—what was that for?

- But there are many other reference monitors that have been proposed – a versatile model
  - Not just for security

- Classical implementation: inlined reference monitor (IRM)
  - Software only implementation
Inline reference monitors

- **Software implementation advantages**
  - Flexible in terms of policy
  - Amenable to compiler optimizations – for example, selectively monitor
  - Don’t need hardware support

- **Disadvantages:**
  - Performance tends to be terrible
  - Coverage
  - Legacy binaries
  - ...

Today

- Look at some reference monitors because
  - Some that are important and that have had commercial impact
  - But also, to think about this class of computation in general

- Hopefully discover that:
  - There are many monitors, but they share a common (or at least similar) computation model
  - Each particular monitor can be implemented in a variety of ways from software only, to hardware only to something in between
    - With different implications – not just performance

- How do we provide hardware support for this class of computation?

- How do we think about implementation tradeoffs
  - Can this be automated?
RAKSHA

A Flexible Information Flow Architecture for Software Security

Michael Dalton   Hari Kannan   Christos Kozyrakis

Computer Systems Laboratory
Stanford University
Motivation

- **Software security is in a crisis**
  - Far-reaching financial & social implications

- **Worms now mixing different kinds of attacks**
  - No longer just simple buffer overflows

- **High-level semantic vulnerabilities now most common threats**
  - SQL Injection, Cross Site Scripting, Directory Traversal, etc
  - Easy to exploit; often architecture & OS independent

- **Need a new approach that is**
  - Robust, End-to-end, Practical, Flexible, Fast
Motivation

- High-level semantic vulnerabilities are prevalent in web-based attacks
  - SQL injection – code inserted into entry field
    
    ```
    SELECT * FROM users WHERE name = '' + userName + '';
    ```

  - Cross-Site Scripting (XSS)
    - Injected website sends malicious code

- Real-World Examples:
  - Website database breach
    - The Wall Street Journal database in July 2014
  - Twitter worms
Dynamic Information Flow Tracking

- **DIFT tags (taints) data from untrusted sources**
  - Each byte or word of memory, register has a taint bit

- **Taint status is propagated across instructions**
  - If any source operand is tainted, destination becomes tainted

- **Trap to OS if tainted data used unsafely**
  - Tainted pointer dereference
  - Tainted jump address
  - Tainted code

- Can prevent memory corruption on unmodified binaries
- **Potential: Protect from low-level & high level threats**
What is DIFT?

**DIFT – Dynamic Information Flow Tracking**

- Associates a tag with every word of memory
- Tag is used to mark tainted data from untrusted sources
  - Data produced from tainted data is also tainted
  - Check tag when data is used for potentially unsafe operations (ex. Code Execution)
- Detects both low and high-level attacks

**User input (untrusted)**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>userName=</td>
</tr>
<tr>
<td>x</td>
<td>'bob'</td>
</tr>
<tr>
<td>x</td>
<td>OR</td>
</tr>
<tr>
<td>x</td>
<td>'1'='1'</td>
</tr>
</tbody>
</table>

**Tag Check**

**SECURITY TRAP**
Limitations of Current DIFT Systems

- **Software-based DIFT**
  - On source code (incomplete coverage)
  - Or transparently through dynamic binary instrumentation

- **Advantages**
  - Runs on existing hardware
  - Flexible security policies

- **Disadvantages**
  - High overhead
  - Cannot protect O/S
  - Cannot protect self-modifying binaries
  - Does not work well with multi-threaded applications
Limitations of Current DIFT Systems

- **Hardware-based DIFT uses one, fixed security policy**
  - Can only solve one problem (e.g., memory corruption) ⇒ unsafe
    - High-level attacks cannot be addressed
  - Cannot adapt to code that violates policy assumptions ⇒ impactical
    - E.g. glib uses alternate bounds checking instructions (index & 0x00FF)
    - Vulnerable to attacks that exploit inflexibility of policies

- **BUT really fast**

- **Hardware security exceptions generate OS traps**
  - Cannot protect OS ⇒ not end-to-end
  - Cannot combine HW and SW to cover difficult cases ⇒ inflexible
    - On a trap, just terminate the program...
Raksha (Protection) Overview

- Get the best of both worlds
  - From H/W: fast check and propagation, work with any binary
  - From S/W: flexible policies, high-level analysis & decisions

- Goals
  - Protect against high-level & low-level attacks
  - Protect against multiple concurrent attacks
  - Protect O/S code
    - At least a major part of it
RAKSHA Overview

- Raksha follows the general DIFT model
  - All state is extended by a 4-bit tag (registers & memory)
    - 4 independent 1-bit states
  - Operations propagate tags from sources to destinations
  - Operations check tags to identify security traps

- New features
  - Software-controlled check & propagate policies ⇒ flexibility
    - Specify policy using check, propagate registers
    - Fine-grain software control to avoid common pitfalls
    - Flexibility allows us to catch wide range of bugs
  - Up to 4 concurrently active policies ⇒ robustness
    - One policy per tag bit
    - Provide comprehensive protection against many bugs
  - Low-overhead, user-level, security traps ⇒ end-to-end, flexibility
    - Can extend with software; can check operating system
Raksha Architecture and Features

- **Unmodified binaries**
  - User 1
    - App Binary
  - User 2
    - App Binary
  - SysAdmin
    - Security Manager

- **Operating System**
  - Tag Aware

- **HW Architecture**
  - Tags

Set security policies
Control HW check/propagate
Further SW analysis

Cross-process info flow
Save/restore tags

4 tag bits per word
HW check/propagate
User-level security traps
Policy Specification

- Up to 4 1-bit policies in parallel
- One check & propagate register per active security policy
- Policies specified at granularity of primitive operation
  - Int/FP Arithmetic, Move, Logical, Comparison, Execute
- Instructions are decoded into $\geq 1$ primitive operations
  - Apply rules specified by check/prop regs to each operation
  - Addresses basic pitfalls of previous designs
  - Conditional Move is logic and move
  - Good for CISC
Security Checking Policies

- Memory address AND data tainted
- Load address is tainted
- Store address is tainted
- Jump destination is tainted*
- Branch condition is tainted*
- System call arguments are tainted*
- Return address register is tainted
- Stack pointer is tainted
- Memory address OR data tainted*

* indicates special treatment or additional checks
Policy Specification Registers

Tag Check Register (Check Enables)

Other Predefined operation Check Enables
[0] Source Check Enable (On/Off)
[1] Destination Check Enable (On/Off)

Custom operation Check Enables
[0] Source 1 Check Enable (On/Off)
[1] Source 2 Check Enable (On/Off)
[2] Destination Check Enable (On/Off)

Execute operation Check Enables
[0] Address (Program Counter) Check Enable (On/Off)
[1] Instruction Check Enable (On/Off)

Move operation Check Enables
[0] Source Check Enable (On/Off)
[1] Source Address Check Enable (On/Off)
[2] Destination Address Check Enable (On/Off)
[3] Destination Check Enable (On/Off)

Tag Propagation Register

Custom Operation Enables
[0] Source Propagation Enable (On/Off)
[1] Source Address Propagation Enable (On/Off)

Move Operation Enables
[0] Source Propagation Enable (On/Off)
[1] Source Address Propagation Enable (On/Off)
[2] Destination Address Propagation Enable (On/Off)

Mode Encoding
00 – No Propagation
01 – AND source operand tags
10 – OR source operand tags
Check Policy: LOAD

Load r2,M[r1+offset]

- **Check Enables**
  - Check source address
    - If Tagged(r1) then security trap
  - Check source
    - If Tagged(M[r1+offset]) then security trap
  - Both Checks can be running simultaneously

- **Propagate Enables**
  - Propagate only from source address
  - Propagate only from source
  - Propagate both tags
    - AND or OR mode
Low Overhead Security Traps

- **On security Trap invoke pre-registered handler**
  - Handler in same address space as code under inspection
    - Fast switch to S/W
  - Handler invocation triggers a special “trusted mode”
  - Trusted mode is orthogonal to user/kernel mode

- **Untrusted Mode**
  - Transparent tags
  - Normal Checks and Propagations

- **Trusted Mode**
  - Visible tags, Tag Access & Manipulation Instructions
  - No Checks / No Propagations
Low Overhead Security Traps

- **Benefits**
  - Can check security of (most of the) OS
    - Reduce the amount of code you really trust
  - Coupling HW and SW security analysis is practical
    - Low performance overhead

- **Drawback:** *A memory corruption can corrupt the security handler*
  - Solution: A security policy used to protect handler code & data
  - Handler Code & data are tainted on initialization
  - Policy does not allow access outside of trusted mode
  - 3 bits remaining...
Tag Granularity & Storage

**Tag granularity**
- HW maintains per word tag bits
- What if SW wants byte or bit granularity for some data?
- Maintain in SW using sandboxing & fast user-level traps
  - Acceptable performance if not common case...

**Tag storage**
- Initial HW $\Rightarrow$ +4 bits/word in registers, caches, memory
  - 12.5% storage overhead
- Multi-granularity tag storage scheme [Suh’04]
  - Exploit tag similarity to reduce storage overhead
  - Page-level tags $\Rightarrow$ cache line-level tags $\Rightarrow$ word-level tags
Raksha Prototype System

- Full-featured Linux system
  - On-line since October 2006...

- HW: modified Leon-3 processor
  - Open-source, Sparc V8 processor, RISC
  - Single-issue, in-order, 7-stage pipeline
  - Modified RTL for processor & system
  - First DIFT system on FPGA
Raksha Software Infrastructure

- **Goal: run real-world software stack**
  - Running a full-featured Linux 2.6 on Raksha hardware
    - Custom distribution booting over NFS
  - Full GNU toolchain + glibc
    - Over 120 packages total
  - Set HW policies using preloaded shared libraries

- **Support enterprise software**
  - SSH
  - Postgresql
  - wu-ftp
  - Apache
  - ...
Raksha Implementation Summary

- Full-system prototype based on LEON 3
  - Open source processor from Gaisler Research
  - SPARC V8 compliant

- Synthesized on Virtex 2 FPGA board

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline depth</td>
<td>7 stages</td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>8KB</td>
</tr>
<tr>
<td>Data Cache</td>
<td>32KB</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>20 Mhz</td>
</tr>
<tr>
<td>Block RAM utilization</td>
<td>22%</td>
</tr>
<tr>
<td>4 input LUT utilization</td>
<td>42%</td>
</tr>
<tr>
<td>Total increase in gates due to tags</td>
<td>7.17%</td>
</tr>
</tbody>
</table>
Raksha Implementation Summary

- **Overhead over original**
  - Logic: 7%
  - Storage: 12.5%
  - Clock frequency: none

- **Application performance**
  - Check/propagate tags ⇒ no slowdown
  - Overhead depends on SW analysis
    - Frequency of traps, SW complexity, ...

- **Worst-case example from experiments**
  - Filtering low-level false positives/negatives
  - Bzip2: +33% with Raksha’s user-level traps
  - Bzip2: +280% with OS traps

- **Memory Speed???
Performance Results

- **Overhead is analysis-dependent**
  - Proportional to exceptions frequency and handler duration
  - Many analyses are very cheap
    - Most high-level analyses invoked infrequently
  - Buffer overflow protection can be most expensive
    - If software is used to correctly filter false-positives/negatives
    - AND with $2^n-1$ is bounds checking

- **Buffer overflow overhead**

<table>
<thead>
<tr>
<th>Program</th>
<th>Exception</th>
<th>OS trap</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>1.01x</td>
<td>1.04x</td>
</tr>
<tr>
<td>crafty</td>
<td>1.01x</td>
<td>1.02x</td>
</tr>
<tr>
<td>gzip</td>
<td>1.31x</td>
<td>3.60x</td>
</tr>
<tr>
<td>bzip2</td>
<td>2.99x</td>
<td>18.80x</td>
</tr>
<tr>
<td>vortex</td>
<td>1.34x</td>
<td>3.41x</td>
</tr>
</tbody>
</table>
Performance Results

<table>
<thead>
<tr>
<th></th>
<th>Compare Filter</th>
<th></th>
<th>AND Filter</th>
<th></th>
<th>Combined Filter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Raksha</td>
<td>Filter</td>
<td>Raksha</td>
<td>Filter</td>
<td>Raksha</td>
</tr>
<tr>
<td>bzip2</td>
<td>2.98x</td>
<td>13.20x</td>
<td>1.19x</td>
<td>1.75x</td>
<td>1.33x</td>
</tr>
<tr>
<td>crafty</td>
<td>1.00x</td>
<td>1.00x</td>
<td>1.00x</td>
<td>1.00x</td>
<td>1.00x</td>
</tr>
<tr>
<td>gap</td>
<td>1.12x</td>
<td>1.70x</td>
<td>1.00x</td>
<td>1.01x</td>
<td>1.49x</td>
</tr>
<tr>
<td>gcc</td>
<td>1.01x</td>
<td>1.04x</td>
<td>1.00x</td>
<td>1.00x</td>
<td>1.00x</td>
</tr>
<tr>
<td>gzip</td>
<td>1.31x</td>
<td>2.92x</td>
<td>2.39x</td>
<td>7.20x</td>
<td>2.66x</td>
</tr>
<tr>
<td>mcf</td>
<td>1.00x</td>
<td>1.04x</td>
<td>1.00x</td>
<td>1.00x</td>
<td>1.00x</td>
</tr>
<tr>
<td>parser</td>
<td>1.04x</td>
<td>1.04x</td>
<td>1.24x</td>
<td>2.28x</td>
<td>1.07x</td>
</tr>
<tr>
<td>twolf</td>
<td>1.58x</td>
<td>4.19x</td>
<td>1.19x</td>
<td>1.86x</td>
<td>1.85x</td>
</tr>
<tr>
<td>vpr</td>
<td>1.00x</td>
<td>1.02x</td>
<td>1.00x</td>
<td>1.00x</td>
<td>1.00x</td>
</tr>
</tbody>
</table>

Table 4: Performance slowdown for the SPEC benchmarks with a pointer tainting analysis that filters false positives by clearing tags for select compare and AND instructions. A slowdown of 1.34x implies that the program runs 34% slower with security checks enabled.
Performance – O/S and S/W traps

Figure 5: The performance degradation for a microbenchmark that invokes a security handler of controlled length every certain number of instructions. All numbers are normalized to a baseline case which has no tag operations.
H/W Policies for Security Experiments

- Concurrent protection using 4 policies

1. Memory corruption (LL attacks)
   - Propagate on arithmetic, load/store, logical
   - Check on tainted pointer/PC use
   - Trap handler untaints data validated by user code

2. String tainting (LL & HL attacks)
   - Propagate on arithmetic, load/store, logical
   - No checks

3. System call interposition (HL attacks)
   - No propagation
   - Check on system call in untrusted mode
   - Trap handler invokes proper SW analysis

4. Sandboxing policy (for trap handler protection)
   - Handler taints its own code & data
   - Check on fetch/loads/stores in untrusted mode
Security Experiments

<table>
<thead>
<tr>
<th>Program</th>
<th>Lang.</th>
<th>Attack</th>
<th>Detected Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traceroute</td>
<td>C</td>
<td>Double Free</td>
<td>Tainted data ptr</td>
</tr>
<tr>
<td>Polymorph</td>
<td>C</td>
<td>Buffer Overflow</td>
<td>Tainted Code ptr</td>
</tr>
<tr>
<td>Wu-FTPD</td>
<td>C</td>
<td>Format String</td>
<td>Tainted”%n” in vfprintf string</td>
</tr>
<tr>
<td>Gzip</td>
<td>C</td>
<td>Directory Traversal</td>
<td>Open tainted dir</td>
</tr>
<tr>
<td>Wabbit</td>
<td>PHP</td>
<td>Directory Traversal</td>
<td>Escape Apache root w/ tainted ‘..’</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>C</td>
<td>Command Injection</td>
<td>Excve tainted file</td>
</tr>
<tr>
<td>ProFTPD</td>
<td>C</td>
<td>SQL Injection</td>
<td>Tainted SQL command</td>
</tr>
<tr>
<td>Htdig</td>
<td>C++</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
<tr>
<td>PhpSysInfo</td>
<td>PHP</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
<tr>
<td>Scry</td>
<td>PHP</td>
<td>Cross-site Scripting</td>
<td>Tainted &lt;script&gt; tag</td>
</tr>
</tbody>
</table>

- DIFT modified SPlinter listening from real world programs
- Basic Computing with application of a language extension
- Protection against low-level memory corruptions
  - Both Control & non-control data attacks
- Protection is independent of programming language
  - Catch suspicious behavior, regardless of language choice

- Unmodified Sparc binaries from real-world programs
Lessons Learned

- **HW support for fine-grain tainting is crucial**
  - For both high-level and low-level attacks
  - Provides fine-grain info to separate legal uses from attacks

- **Lesson from high-level attacks**
  - Check for attacks at system calls
  - Provides complete mediation, independent language/library

- **Lessons from low-level attack**
  - Fixed policies from previous DIFT systems are broken

- **False positives & negatives even within glibc**
  - Problem: what constitutes validation of tainted data?
  - Need new SW analysis to couple with HW tainting
    - Raksha’s flexibility and extensibility are crucial
SIFT: SMT based Information Flow Tracking
Another design point

- New ideas (to us):
  - Parallelize the reference monitor thread
    - Checking in parallel with execution – what are the implications?
  - Hardware support for reference monitor generation
    - Little hardware support for execution
    - Use normal instructions and a spare SMT hardware context
  - Some optimizations…
Existing DIFT Schemes and Limitations

- **Hardware solutions:**
  - Taint propagation with extra busses
  - Additional checking units

  **Limitations**
  - Intrusive changes to datapath design

- **Software solutions:**
  - More instructions to propagate and check taint

  **Limitations**
  - High performance cost
  - Source code recompilation
Hardware–based DIFT

```
add r3, r1, r4
```
Software –based DIFT

**add r3 r1 r4**

- add r3 r1 r4
  - compiler
    - add r3 r1 r4
      - shr r2 r5 2
      - shr r0 r5 1
      - and r2 r2 16
      - and r0 r0 16
      - or r0 r2 r0
      - or r5 r5 r0

- RF
  - r0
  - r1
  - r2
  - r3
  - r4
  - 01100000

- RF
  - 00111100

- MEM
  - data
  - data
  - data
  - data

- A small region of memory is used to store taint information of memory

- r5 is used for storing taint information of remaining register file

- Existing

- DIFT
SIFT (SMT-based DIFT)

- Execute two threads on SMT processor
  - Primary thread executes real program
  - Security thread executes taint tracking instructions

- Committed instructions from main thread generate taint checking instruction(s) for security thread

- Instruction generation is done in hardware

- Taint tracking instructions are stored into a buffer from where they are fed to the second thread context

- Threads are synchronized at system calls
Instruction Flow in SIFT
SMT Datapath with SIFT Logic

SMT Datapath with SIFT Support

- Instruction Cache
- Fetch Unit
- Decode/Dispatch
- Register Rename
- IFQ
- PC
- IQ
- ROB
- Arch State
- SIFT Instruction Generator
- Data Cache
- Mem Units
- LSQ
- addr
- Inst

Shared Resources
Private Resources
SIFT Example

add r3 r1 r4
SIFT Instruction Generation Logic

1. Taint Code Generation

2. Security Instruction Opcodes are read from COT

3. Rest of the instructions are taken from Register Organizer and stored Instruction Buffer

4. Load and Store Instruction’s memory addresses are stored in Address Buffer
Die Floorplan with SIFT Logic

- SUN T1 Open Source Core
- IGL synthesized using Synopsys Design Compiler using a TSMC 90nm standard cell library
- COT, IB and AB implemented using Cadence Virtuoso
- The integrated processor netlist placed and routed using Cadence SoC Encounter
- Cost 4.5% of whole processor area
Benefits of Taint Checking with SMT

- Software is not involved, transparent to user and applications (although the checking code can also be generated in software)

- Hardware instruction generation is faster than software generation

- Additional hardware is at the back end of the pipeline, it is not on the critical path

- No inter-core communication
SIFT Performance Overhead

Performance Loss

0% 10% 20% 30% 40% 50% 60% 70%

astar bzip2 cactusADM cactux dealII gamess gcc GensFDTD gobmk gromacs h264ref hmmer lbm lesi3d libquantum mcf milc namd omnet++ povray sjeng spbench spez xz avg
SIFT Performance Optimizations

- Reduce the number of checking instructions by eliminating the ones that never change the taint state.
- Reduce data dependencies in the checker by preloading taint values into its cache once the main program encounters the corresponding address.
- Reduce the number of security instructions depending on taint state of registers and TLB.
## Eliminating Checking Instructions

### Primary Thread
- `lda r0,24176(r0)`
- `xor r9,3,r2`
- `addq r0,r9,r0`
- `ldah r16,8191(r29)`
- `ldq_u r1,0(r0)`
- `lda r16,-26816(r16)`
- `lda r0,1(r0)`
- `lda r18,8(r16)`
- `extqh r1,r0,r1`
- `sra r1,56,r10`
- `bne r2,0x14`
- `and r9,255,r9`
- `stl r3,32(r30)`
- `ldl r2,64(r2)`
- `lda r16,48(r30)`
- `bic r2,255,r2`
- `bis r3,r2,r2`

### SIFT Security Thread
- `bis r0,r0,r0`
- `bis r9,r9,r2`
- `bis r0,r9,r0`
- `bis r29,29,r16`
- `ldq_u r1,0(r0)`
- `bis r1,r0,r1`
- `bne r1,0xfffffffffffff080`
- `bis r16,r16,r16`
- `bis r0,r0,r0`
- `bis r16,r16,r18`
- `bis r1,r0,r1`
- `bis r1,r1,r10`
- `bne r2,0xfffffffffffff080`
- `bis r9,r9,r9`
- `bne r3,r30,r3`
- `bne r3,0xffffffffffff080`
- `stl r3,32(r30)`
- `ldl r2,64(r30)`
- `stl r3,32(r30)`
- `ldl r2,64(r30)`
- `stl r3,32(r30)`
- `stl r2,64(r2)`
- `bis r2,2,r2`
- `bne r2,0xfffffffffffff080`
- `bis r30,r30,r16`
- `bis r2,2,r2`
- `bis r3,r2,r2`

### SIFT – F Security Thread
- `bis r9,r9,r2`
- `bis r0,r9,r0`
- `bis r29,29,r16`
- `ldq_u r1,0(r0)`
- `bis r1,r0,r1`
- `bne r1,0xfffffffffffff080`
- `bis r16,r16,r18`
- `bis r1,r0,r1`
- `bis r1,r1,r10`
- `bne r2,0xfffffffffffff080`
- `bis r3,r30,r3`
- `bne r3,0xffffffffffff080`
- `stl r3,32(r30)`
- `ldl r2,64(r30)`
- `bne r2,0xfffffffffffff080`
- `bis r30,r30,r16`
- `bis r3,r2,r2`
SIFT Logic with Instruction Elimination
Performance Impact of Eliminating Security Instructions

SIFT and SIFT-F Performance

- SIFT
- SIFT-F

Percentage of Instructions Filtered by SIFT-F
Performance Impact of Cache Prefetching

SIFT and SIFT-P Performance

SIFT L1 Cache Misses
SIFT Performance on a 8-way Issue Processor

PErformance Loss

SIFT  SIFT-F  SIFT-P  SIFT-FP

0%  10%  20%  30%  40%  50%  60%  70%

Performance Loss on an 8-way Issue Processor
Performance Effect of ISA Support

Performance Loss

- SIFT-FP
- SIFT-FP (ISA Support)
Performance Effect of Cache Sharing

SIFT-FP  SIFT-FP(Separate Cache)  SIFT-FP(Separate Perfect Cache)

Performance Loss

PErformance Loss

0% 5% 10% 15% 20% 25% 30% 35% 40% 45% 50%

aastar bzip2 cactusADM calculix dealii gamess gcc GemsFDTD gobmk gromacs h264ref hmmer ibm lesie3d libquantum mcf milc milc planu ompt norun povray sjeng spiking suse zeusze avg
SIFT-FP with Different Policies

Policy 1 – No Checking only propagation
Policy 2 – Jump Targets, Return Address, Stack Pointer
Policy 3 – Jump Targets, Branch Conditions, System Call Arguments, Memory data and address
Multicore SIFT

L2 Cache

L1 Cache

Pipeline

SIFT

L1 Cache

Pipeline

Core 0

Core 1

SIFT Buffer (Addresses & Instructions)
Multicore SIFT Performance

- MSIFT
- MSIFT-FP
MSIFT with Different Buffer Sizes

Performance Loss

-10% 0% 10% 20% 30% 40% 50%

aastar bzip2 cactusADM calculix dealii gamses gcc
GemsFDTD gobmk gromacs h264ref hmmer lbm
leslie3d libquantum mcf milc
omnetpp povray sjeng sjeng3d sultan zeusmp

MSIFT MSIFT-FP MSIFT-FP 32 Entry Buffer MSIFT-FP 128 Entry Buffer MSIFT-FP Inf. Buffer
Multicore SIFT with Perfect Cache

Performance Loss

-5% 0% 5% 10% 15% 20% 25% 30%

astar  bzip2  cactusADM  dealII  gamesh  gcc  GemsFDTD  gohmkr  gromacs  h264ref  hmmer  ibm  leslie3d  libquantum  mcf  milc  mpirun  omnetpp  povray  sjeng  sphinx3  Zeusmp  blm

MSIFT-FP  MSIFT-FP Perfect Cache  MSIFT-FP Perfect Cache (32 Entry)  MSIFT-FP Perfect Cache (128 Entry)  MSIFT-FP Perfect Cache Inf. Buffer
Discussion

- Collapse multiple checking instructions via ISA?
- Optimize resource sharing between two threads?
- Provide additional execution units for the checker?
- Using memory page granularity taint?
General RM Hardware support

- Is this a good idea?
  - What are the benefits

- What would it look like? Parallel checking model? Custom Hardware?

- How do you retask it for different reference monitors?

- Can we benefit from both software optimizations and hardware acceleration?