Lightweight Memory Tracing

Mathias Payer*, Enrico Kravina, Thomas Gross
Department of Computer Science
ETH Zürich, Switzerland

* now at UC Berkeley
Memory Tracing via Memlets

Execute code (**memlets**) for every memory access

A memlet inspects a single memory access based on target **address**, **type** of memory access, **instruction**, or prior **state**

Memory tracing enables detailed memory access logs, debugging of memory accesses, security checks, privacy extensions
Memory Tracing by Example

Binary translation weaves memlets into executed code

`memTrace` is general, for talk let’s focus on example:

- Unlimited `watchpoints`: check if R/W watchpoint is set

```
addl (%ebx), %eax
jg bb1
jmp bb2
```

```
/* check */
lea (%ebx), %reg
cmpl 0xshadow(%reg), $0x0
jnz handler_92746
/* translated instruction */
addl (%ebx), %eax
jg bb1
jmp bb2
```
Key to *Lightweight* Memory Tracing

Modern CPUs support multiple ISAs: x86/x86_64
  • Most programs still 32-bit x86

Cross-ISA binary translation allows the tracer to use additional hardware available in target ISA:
  • Wider address space: isolation & performance
  • Additional registers: flexibility & performance
Outline

Motivation and Introduction

*Lightweight Memory Tracing*

- Requirements
- User-defined Memlets
- Cross-ISA Binary Translation (BT)
- Implementation

Evaluation

Related Work

Conclusion
Tracing Requirements

Flexibility

Isolation

Performance
Flexibility through BT

Cross-ISA BT

- Translates individual basic blocks
- Checks branch targets and origins
- Weaves memlets into code

Original x86 code

1
2
3
4

Dynamic translator

Translating x64 kernel

Translated x64 code

1'
2'
3'

Memlets execute alongside application

x64 Kernel
### Isolation: Larger Memory Space

<table>
<thead>
<tr>
<th>Application memory</th>
<th>Shadow memory</th>
<th>Translator memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code &amp; Data</td>
<td>Code &amp; Data'</td>
<td>Translator Code</td>
</tr>
<tr>
<td>Heap</td>
<td>Heap'</td>
<td>Translator Data</td>
</tr>
<tr>
<td>Stack</td>
<td>Stack'</td>
<td>Code Cache &amp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Translator Stack</td>
</tr>
</tbody>
</table>

- **0x0000’0000**
- **0x0’FFFF’FFFF (4GB)**
- **0x?’FFFF’FFFF (x*4GB)**

**Wider memory space**

Isolates tracer from application
Key to Low Overhead

Fast, efficient binary translation
Letting the hardware do most of the work...
  • use 64-bit addressing (aligned 4GB blocks)
  • keep state in additional/wider registers
  • optimize for EFLAGS usage
Implementation

**memTrace** implementation (open source)

- Cross-ISA translator
- Sample memlets

Small, lean implementation

<table>
<thead>
<tr>
<th></th>
<th>Code</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>memTrace</td>
<td>13,800*</td>
<td>3,300</td>
</tr>
<tr>
<td>Memlets</td>
<td>150-200</td>
<td>100-200</td>
</tr>
</tbody>
</table>

*4,900 LOC for the translation tables
Outline

Motivation and Introduction
Lightweight Memory Tracing

*Evaluation*
- Unlimited Watchpoints
- Safe Memory Allocation

Related Work

Conclusion
Unlimited Watchpoints

Watchpoints trigger on memory reads/writes
Memlet checks if read/write watchpoint is set for each memory access

```assembly
addl (%ebx), %eax
jg bb1
jmp bb2
```

/* check */
lea (%ebx), %r8
cmpl 0x100000000(%r8), $0x0
jnz handler_92746
/* translated instruction */
addl (%ebx), %eax
jg bb1
jmp bb2
Evaluation Setup

SPEC CPU2006 benchmarks evaluated
  • System: Ubuntu 12.04, GCC 4.6.3 (64bit)
  • Intel Core i7-2640M @ 2.80GHz, 4GB RAM

Four configurations:
  • Native
  • Binary translation (BT) only
  • Memory Tracing
  • Full Watchpoints
SPEC CPU 2006: Low Perf. Impact

- Binary Translation
- Memory Tracing
- Full Watchpoints

Average
Geo. Mean

SPEC CPU 2006 benchmarks include:
- 400.perlbench
- 401.bzip2
- 403.gcc
- 429.mcf
- 456.hmmer
- 458.sjeng
- 462.libquantum
- 464.h264ref
- 471.omnetpp
- 473.astar
- 483.xalancbmk
- 484.zeusmp
- 485.gromacs
- 486.cactusADM
- 487.leslie3d
- 488.openmpi
- 489.soplex
- 490.ammp
- 493.cтин
- 494.nas3d
- 495.povray
- 496.calculix
- 497.dealII
- 498.gromacs
- 499.tensorflow
- 500.tonto
- 501.lbm
- 502.sphinx3

The chart compares the performance impact of different techniques across various benchmarks.
Safe Memory Allocation

Check for use-after-free bugs and heap corruption

Intercept calls to \texttt{malloc} and \texttt{free}

- Protect metadata of allocated blocks
- Check for read/write accesses to freed blocks until they are reused
Outline

Motivation and Introduction
Lightweight Memory Tracing
Evaluation
Related Work
Conclusion
Related work

Valgrind allows high-level transformations on machine code with performance cost (~7x for nullgrind, ~26x for memcheck)

GDB/Hardware watchpoints allow a limited set of watchpoints with negligible overhead

Limitations of other dynamic tracing systems are (i) limited ISA support, (ii) high overhead, or (iii) limited flexibility
Outline

Motivation and Introduction
Lightweight Memory Tracing
Evaluation
Related Work
Conclusion
Conclusion

*memTrace* enables lightweight, low-overhead <90% memory inspection for unmodified applications
  • Use resources of modern CPUs

Memlets allow user-configurable checks for each memory access
  • Flexible framework for memory tracing

Source:
  • [http://nebelwelt.net/projects/memTrace/](http://nebelwelt.net/projects/memTrace/)
  • [https://github.com/gannimo/memTrace](https://github.com/gannimo/memTrace)