CrystalBall: Statically Analyzing Runtime Behavior via Deep Sequence Learning

Steve Zekany, Daniel Rings, Nathan Harada, Michael A. Laurenzano, Lingjia Tang, Jason Mars
University of Michigan, Ann Arbor, MI
Improving Performance

- Architectural Improvements
- Compile-time Optimizations
- Run-time Optimizations

But how can we measure behavior?

Understanding program behavior enables design of performance improvements in architecture & software
Paths Characterize Program Behavior

- **Path**: a sequence of basic blocks
- **Hot path**: sequence of blocks executed frequently
- Identifying hot paths enables optimization
Finding Hot Paths

**Dynamic profiling**

- Entire program
- Requires representative data
- Time-consuming

**Can we do this statically?**
Prior Work

Static Branch Prediction\textsuperscript{1}

- Language-independent, heuristic-based classification system for branches
  
  \textbf{X} Does not capture sophistication of program execution

Hot Path Prediction\textsuperscript{2}

- Traditional machine learning techniques to predict paths
  
  \textbf{X} Hand-engineered, language-specific path features

\textsuperscript{1} Ball & Larus: Static branch frequency and program profile analysis (MICRO 1994)
\textsuperscript{2} Buse & Weimer: The road not taken: Estimating path execution frequency statically (ICSE 2009)
Finding Hot Paths

**Dynamic profiling**
- Entire program
- Requires *representative* data
- Time-consuming

**Static analysis**
- Run on individual functions
- No program data required
- Requires only time to run inference
Static Analysis Design Objectives

• High accuracy
• Language independence
• No hand-engineered features

We designed CrystalBall to meet these objectives...
High Accuracy Hot Path Prediction

Utilize long short-term memory (LSTM) recurrent neural network

- RNN: ideal for sequence-learning
- LSTM: allow the network to selectively “forget”

Goal: teach the neural network to recognize hot paths
Language Independence

Operate on intermediate representation (IR)

• IR gives us *language-independence*

• Use LLVM to instrument program
No Hand-Engineered Features

- Compiler pass reconstructs paths
- Feature vector is a count of opcodes
- Feature vector generated for each block in path

No hand-engineered features needed
CrystalBall Overview

IR Code

Compiler Pass (LLVM) and Instrumentation

Feature Vectors

LSTM Neural Network

Trained Hot Path Model

Hot Path Prediction
CrystalBall Overview

IR Code

Compiler Pass (LLVM) and Instrumentation

Feature Vectors

LSTM Neural Network

Trained Hot Path Model

Hot Path Prediction
Neural Network Structure

- Once trained, run inference on arbitrary code
Characterization

- We define a hot path as a path executed at least once.
- A small number of paths account for majority of runtime.
Choosing Better Metrics

Accuracy is a poor metric for some problems

AUROC is a superior metric
Methodology

• Experiments performed using 6-core Intel Xeon E5-2620, 256 gigabytes of DRAM, and NVidia K40M GPU
  • LLVM 3.3 and Python 2.7.6
  • Neural network models use Theano and Keras
• SPEC 2006 benchmarks
  • compiled with the -02 optimization flag
• Also utilized two kernels from Sirius*, an open-source end-to-end personal assistant pipeline

* Hauswald, Johann, et al., ASPLOS 2015
Accuracy: AUROC
Accuracy: AUROC
Factors

- Logistic regression using hand-engineered features from prior work.
Conclusion

• Sequence-learning LSTM Neural Network
• Operates on intermediate representation (IR)
• No hand-engineered features needed
Questions?
Low-overhead Online Code Transformations with Protean Code

Michael A. Laurenzano
Compilers background
Compilers background

- Translate from high-level programming language (C, C++) to machine-level instructions (x86, ARM)
Compilers background

• Translate from high-level programming language (C, C++) to machine-level instructions (x86, ARM)

• Must generate correct code
Compilers background

• Translate from high-level programming language (C, C++) to machine-level instructions (x86, ARM)

• Must generate correct code

• Many implementations are correct — we want the “best” one (highest performance, security, resilience, etc.)

  • Depends on platform/hardware

  • Depends on environment (other apps, program inputs)
Compilers background

- Translate from high-level programming language (C, C++) to machine-level instructions (x86, ARM)
- Must generate correct code
- Many implementations are correct — we want the “best” one (highest performance, security, resilience, etc.)
  - Depends on platform/hardware
  - Depends on environment (other apps, program inputs)

Compilers make assumptions about environment
Why online code transformations?
Why online code transformations?

- Static implementation decisions are based on assumptions
  - Hardware, co-runners, etc. are often unpredictable at compile time
  - Environments change over time
Why online code transformations?

- Static implementation decisions are based on assumptions
  - Hardware, co-runners, etc. are often unpredictable at compile time
  - Environments change over time
- Instead, use online code transformations to make implementation decisions dynamically
Why online code transformations?

- Static implementation decisions are based on assumptions
  - Hardware, co-runners, etc. are often unpredictable at compile time
  - Environments change over time
- Instead, use online code transformations to make implementation decisions dynamically
Why online code transformations?

- Static implementation decisions are based on assumptions
  - Hardware, co-runners, etc. are often unpredictable at compile time
  - Environments change over time
- Instead, use online code transformations to make implementation decisions dynamically

Optimization

Resilience

Debugging

Security

Portability
Holy grail problem
Holy grail problem

- Continuously deployable — usable in production to solve a broad range of problems
Holy grail problem

- Continuously deployable — usable in production to solve a broad range of problems
- Capabilities needed
Holy grail problem

- Continuously deployable — usable in production to solve a broad range of problems
- Capabilities needed
  - Low overhead — spurned by users, hurts optimization opportunities
Holy grail problem

- **Continuously deployable** — usable in production to solve a broad range of problems

- Capabilities needed
  - Low overhead — spurned by users, hurts optimization opportunities
  - General purpose — programmer involvement, specialized hardware
Holy grail problem

- **Continuously deployable** — usable in production to solve a broad range of problems
- **Capabilities needed**
  - Low overhead — spurned by users, hurts optimization opportunities
  - General purpose — programmer involvement, specialized hardware
  - High transformation power — hoist binary to IR, limits compiler flexibility
Holy grail problem

• Continuously deployable — usable in production to solve a broad range of problems

• Capabilities needed
  • Low overhead — spurned by users, hurts optimization opportunities
  • General purpose — programmer involvement, specialized hardware
  • High transformation power — hoist binary to IR, limits compiler flexibility
  • Extrospective — monitor multiple applications, key for optimization
Holy grail problem

- **Continuously deployable** — usable in production to solve a broad range of problems

- **Capabilities needed**
  - Low overhead — spurned by users, hurts optimization opportunities
  - General purpose — programmer involvement, specialized hardware
  - High transformation power — hoist binary to IR, limits compiler flexibility
  - Extrospective — monitor multiple applications, key for optimization

<table>
<thead>
<tr>
<th></th>
<th>ADAPT</th>
<th>ADORE</th>
<th>DynamoRIO</th>
<th>Protean Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low overhead</td>
<td>Green</td>
<td>Green</td>
<td></td>
<td></td>
</tr>
<tr>
<td>General purpose HW/SW</td>
<td></td>
<td></td>
<td>Green</td>
<td></td>
</tr>
<tr>
<td>High transformation power</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extrospective</td>
<td></td>
<td></td>
<td></td>
<td>Green</td>
</tr>
</tbody>
</table>
Holy grail problem

- Continuously deployable — usable in production to solve a broad range of problems
- Capabilities needed
  - Low overhead — spurned by users, hurts optimization opportunities
  - General purpose — programmer involvement, specialized hardware
  - High transformation power — hoist binary to IR, limits compiler flexibility
  - Extrospective — monitor multiple applications, key for optimization

<table>
<thead>
<tr>
<th></th>
<th>ADAPT [Voss PPoPP'01]</th>
<th>ADORE [Chen MICRO'03]</th>
<th>DynamoRIO [Bruening CGO’03]</th>
<th>Protean Code [Laurenzano MICRO’14]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low overhead</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>General purpose HW/SW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>High transformation power</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extrospective</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pro•te•an Code
<table>
<thead>
<tr>
<th>Conventional Dynamic Compilation</th>
</tr>
</thead>
</table>
Conventional Dynamic Compilation

- Fine-grain control at instruction/BB level
- Frequent transitions from compiler to app
- Instrumentation for monitoring of EE
Conventional Dynamic Compilation

- Fine-grain control at instruction/BB level
- Frequent transitions from compiler to app
- Instrumentation for monitoring of EE

- High overhead
**Conventional Dynamic Compilation**

- Fine-grain control at instruction/BB level
- Frequent transitions from compiler to app
- Instrumentation for monitoring of EE
- High overhead

**Protean Code**

- No fine-grain control, execute directly
- Statically stitch in adaptability
- Recompile asynchronously
Conventional Dynamic Compilation

- Fine-grain control at instruction/BB level
- Frequent transitions from compiler to app
- Instrumentation for monitoring of EE
- High overhead

Protean Code

- No fine-grain control, execute directly
- Statically stitch in adaptability
- Recompile asynchronously
- Low overhead
application

- func1
  - func2
  - func3
    - func4
    - func5
application

func1

func2

func3

func4

func5

static

protean code compiler

protean application

func1

func2

func3

func4

func5

EVT

&func2

&func3

&func4

&func5

IR + Metadata

protean application

application
Virtualize control edges, create edge virt. table
Virtualize control edges, create edge virt. table

Pack intermediate rep. into application binary
1. Compilation begins with IR
1. Compilation begins with IR

2. New code put into code cache

- Runtime
  - Compiler
  - Decision Engine
  - Code Cache
  - EVT Manager
  - Monitoring + Phase Analysis
1. Compilation begins with IR

2. New code put into code cache

3. Control redirected via EVT

---

runtime

protean application

IR + Metadata

EVT

func1

func2

func3

func4

func5

Runtime

Compiler

EVT Manager

Monitoring + Phase Analysis

Decision Engine

Code Cache

&func2

&func3

&func4

&func5

func1

func2

func3

func4

func5

&func5 v4

func4 v2

func5 v3

func5 v4

&func5 v4
1. Compilation begins with IR

2. New code put into code cache

3. Control redirected via EVT

runtime
Application continuously runs!

1. Compilation begins with IR

2. New code put into code cache

3. Control redirected via EVT

runtime
Key feature - no programmer or HW assistance
Key feature - low overhead
Key feature - low overhead

- It is all about the EVT
Key feature - low overhead

- It is all about the EVT
- Execution diverted only at particular points
Key feature - low overhead

- It is all about the EVT
- Execution diverted only at particular points
- Asynchronous compiler
- Application executes continuously
- Cycles used by the compiler is tunable
Key feature - transformation power
Key feature - transformation power

- Fat binary with IR
- High-level semantic and structural information
- Examples used in my work — loop depth and location
Key feature - introspection and extrospection
Key feature - introspection and extrospection

- Hardware performance monitors
Key feature - introspection and extrospection

- Hardware performance monitors
- Program counter samples
Key feature - introspection and extrospection

- Hardware performance monitors
- Program counter samples
- Continuous phase analysis of application + co-runners
  - Revisit compilation choices as environment changes
Overhead of protean code

- DynamoRIO [Bruening CGO’03] is state of the art dynamic compiler
- SPEC CPU2006 applications
Overhead of protean code

- DynamoRIO [Bruening CGO’03] is state of the art dynamic compiler
- SPEC CPU2006 applications

```
<table>
<thead>
<tr>
<th>Application</th>
<th>Mean Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>1.0</td>
</tr>
<tr>
<td>gcc</td>
<td>1.1</td>
</tr>
<tr>
<td>mcf</td>
<td>1.2</td>
</tr>
<tr>
<td>milc</td>
<td>1.3</td>
</tr>
<tr>
<td>namd</td>
<td>1.4</td>
</tr>
<tr>
<td>gobmk</td>
<td>1.5</td>
</tr>
<tr>
<td>dealII</td>
<td>1.6</td>
</tr>
<tr>
<td>soplex</td>
<td>1.7</td>
</tr>
<tr>
<td>povray</td>
<td>1.8</td>
</tr>
<tr>
<td>hmmer</td>
<td>1.9</td>
</tr>
<tr>
<td>sjeng</td>
<td>2.0</td>
</tr>
<tr>
<td>libquantum</td>
<td>2.1</td>
</tr>
<tr>
<td>h264ref</td>
<td>2.2</td>
</tr>
<tr>
<td>lbm</td>
<td>2.3</td>
</tr>
<tr>
<td>omnetpp</td>
<td>2.4</td>
</tr>
<tr>
<td>astar</td>
<td>2.5</td>
</tr>
<tr>
<td>sphinx3</td>
<td>2.6</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>2.7</td>
</tr>
<tr>
<td>MEAN</td>
<td>2.8</td>
</tr>
</tbody>
</table>
```

Diagram: Protean code vs. DynamoRIO

- Protean code: <1% vs. 19%
Head in the Cloud
Original libquantum

```
mov   %r13,%rsi
shl   $0x4,%rsi
mov   (%r14),%r8
mov   (%r8,%rsi,1),%rax
```

Fully non-temporal libquantum

```
mov   %r13,%rsi
shl   $0x4,%rsi
prefetchnta (%r14)
mov   (%r14),%r8
prefetchnta (%r8,%rsi,1)
mov   (%r8,%rsi,1),%rax
```
We need an online code transformation mechanism to apply these hints dynamically.
Protean Code for Cache Contention in Datacenters (PC3D)
Protean Code for Cache Contention in Datacenters (PC3D)

Throttle-based mitigation
Protean Code for Cache Contention in Datacenters (PC3D)

Throttle-based mitigation

PC3D

high priority

zzz...

prefetchnta
Protean Code for Cache Contention in Datacenters (PC3D)

- Goal — enact the program variant with the right mix of non-temporal (NT) hint instructions

![Throttle-based mitigation](image1)

![PC3D](image2)
Protean Code for Cache Contention in Datacenters (PC3D)

- **Goal** — enact the program variant with the right mix of non-temporal (NT) hint instructions

- **My approach**
  - Dynamically identify the right program variant
  - Enact a series of variants and see how they perform
PC3D vs. state-of-the-art

- ReQoS [Tang ASPLOS’13] throttles/naps applications to meet co-runner QoS
- Co-runners include CloudSuite and SPEC
- PC3D and ReQoS hit co-runner QoS targets
PC3D vs. state-of-the-art

- ReQoS [Tang ASPLOS’13] throttles/naps applications to meet co-runner QoS
- Co-runners include CloudSuite and SPEC
- PC3D and ReQoS hit co-runner QoS targets
Bytes, Bytes, Everywhere
Why worry about return oriented programming?
Why worry about return oriented programming?

- **Nefarious** — hijack program execution (get the program to do something on the attacker’s behalf)

  - Change a file, open permissions, launch a shell, etc.
Why worry about return oriented programming?

- **Nefarious** — hijack program execution (get the program to do something on the attacker’s behalf)
  - Change a file, open permissions, launch a shell, etc.

- **Powerful** — ROP is turing complete [Shacham CCS’07]
Why worry about return oriented programming?

- **Nefarious** — hijack program execution (get the program to do something on the attacker’s behalf)
  - Change a file, open permissions, launch a shell, etc.
- **Powerful** — ROP is turing complete [Shacham CCS’07]
- **Widespread** — 95% of Windows exploits discovered between 2013-2015 used ROP [Rain RSA’15]
How ROP works

1. dynamically discover a chunk of program’s code to locate gadgets
2. “compile” the attack, back to Step 1 if more gadgets needed
3. execute the attack
How ROP works

1. dynamically discover a chunk of program’s code to locate gadgets

2. “compile” the attack, back to Step 1 if more gadgets needed

3. execute the attack
How ROP works

1. dynamically discover a chunk of program’s code to locate gadgets

2. “compile” the attack, back to Step 1 if more gadgets needed

3. execute the attack
How ROP works

1. dynamically discover a chunk of program’s code to locate gadgets

2. “compile” the attack, back to Step 1 if more gadgets needed

3. execute the attack
How ROP works

1. dynamically discover a chunk of program’s code to locate gadgets

2. “compile” the attack, back to Step 1 if more gadgets needed

3. execute the attack
How ROP works

1. dynamically discover a chunk of program’s code to locate gadgets

2. “compile” the attack, back to Step 1 if more gadgets needed

3. execute the attack
What to do about ROP?
What to do about ROP?

• The assumption — gadgets remain in place from attack inception to completion
What to do about ROP?

- The assumption — gadgets remain in place from attack inception to completion
- The idea — continuous re-randomization
  - Invalidate this assumption by shuffling all executable bytes in program memory
  - Move bytes quickly enough to make them useless to attacker
  - Introduce low runtime overhead

![Diagram showing conventional vs continuously re-randomized code locations](image)
What to do about ROP?

• The assumption — gadgets remain in place from attack inception to completion

• The idea — continuous re-randomization

  • Invalidate this assumption by shuffling all executable bytes in program memory

  • Move bytes quickly enough to make them useless to attacker

  • Introduce low runtime overhead

“While [re-randomizing code] may be one way [to render our attack ineffective], we expect that re-randomization costs would make such a solution impractical.” [Snow SP’13]
What to do about ROP?

• The assumption — gadgets remain in place from attack inception to completion

• The idea — continuous re-randomization

  • Invalidate this assumption by shuffling all executable bytes in program memory

  • Move bytes quickly enough to make them useless to attacker

  • Introduce low runtime overhead
Re-randomization overhead

- Re-randomize (medium-grain) all code/EVT every 30ms - 30s
- Overhead averages 3% - 30%
- Classical dynamic compilation solution checking call/return pairs [Davi CCS’11] has 2x overhead

Re-randomization Frequency.

We conduct experiments to examine the performance impact of the re-randomization service as a function of the re-randomization frequency. Figure 5.8 presents the results, showing the runtime of each program at a range of different re-randomization frequencies ranging from 30ms to 30s when exclusively using medium-grain re-randomization. The runtime results in this experiment are normalized to the execution time of the stock ELF/Linux program, and are inclusive of all sources of overhead.

The overhead of code re-randomization increases as the re-randomization becomes more frequent. We examine the sources of overhead in more detail shortly. The performance overhead is below 5% at all frequencies larger than 1 second. Moreover, the performance overhead of re-randomizing once every 300ms averages 9%. As we discuss shortly, a frequency of 300ms thwarts current state-of-the-art ROP attacks.
Research opportunities
Research opportunities

- **Optimization** — tiling, prefetching, instruction scheduling, parallelism, etc.
Research opportunities

- **Optimization** — tiling, prefetching, instruction scheduling, parallelism, etc.
- **Security** — control flow integrity, software diversity
Research opportunities

• **Optimization** — tiling, prefetching, instruction scheduling, parallelism, etc.

• **Security** — control flow integrity, software diversity

• **Debugging** — dynamically insert trap, printf, etc. into code
Research opportunities

- **Optimization** — tiling, prefetching, instruction scheduling, parallelism, etc.
- **Security** — control flow integrity, software diversity
- **Debugging** — dynamically insert trap, printf, etc. into code
- **Resilience** — multiple copies of instruction to unreliable unit
Research opportunities

- **Optimization** — tiling, prefetching, instruction scheduling, parallelism, etc.
- **Security** — control flow integrity, software diversity
- **Debugging** — dynamically insert trap, printf, etc. into code
- **Resilience** — multiple copies of instruction to unreliable unit
- **Portability** — specialize code for heterogeneous CPUs/ISAs
Research opportunities

- **Optimization** — tiling, prefetching, instruction scheduling, parallelism, etc.
- **Security** — control flow integrity, software diversity
- **Debugging** — dynamically insert trap, printf, etc. into code
- **Resilience** — multiple copies of instruction to unreliable unit
- **Portability** — specialize code for heterogeneous CPUs/ISAs
- **Protean Code Mechanism**
  - Outlining to get access to important code regions
  - Inlining to minimize overhead
  - Hoist binary to IR — minimize the role of static side