JIT-Compiled Frameworks: Compiler/Hardware Co-optimization

Wonsun Ahn

Assistant Professor

Department of Computer Science
University of Pittsburgh

http://cs.pitt.edu/~wahn/

Pitt

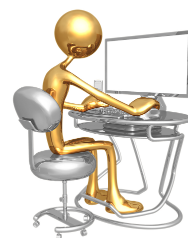
CS2001, October 2016

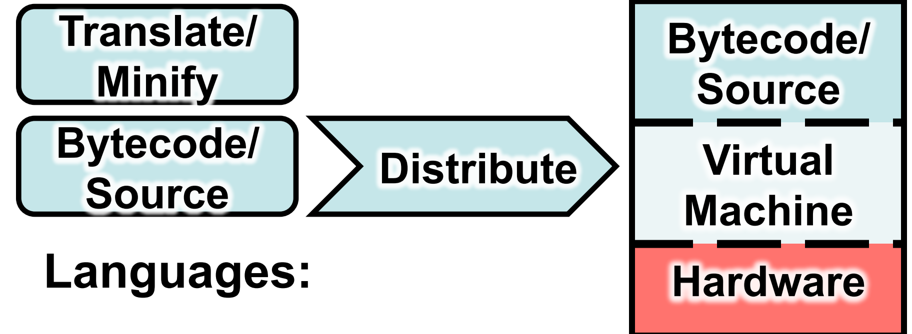
What is Just-In-Time (JIT) Compilation?

Ahead-Of-Time (AOT) Compilation

Languages:

Just-In-Time (JIT) Compilation





Languages:

JavaScript

OpenCL

Pitt

Growth of JIT-Compiled / Scripting Languages

* Number of programmers coding in given language over time
(Reference: www.ohloh.net)

20%
15%
10%
5%
2005 2006 2007 2008 2009 2010 2011 2012 2013 201

Mobile Web
Big Data Analysis

C
C++
AOT
Java
JavaScript
PHP
Python
JIT

- C/C++ dominated in 2005
- Now, JavaScript and Python are the most used languages

Pitt

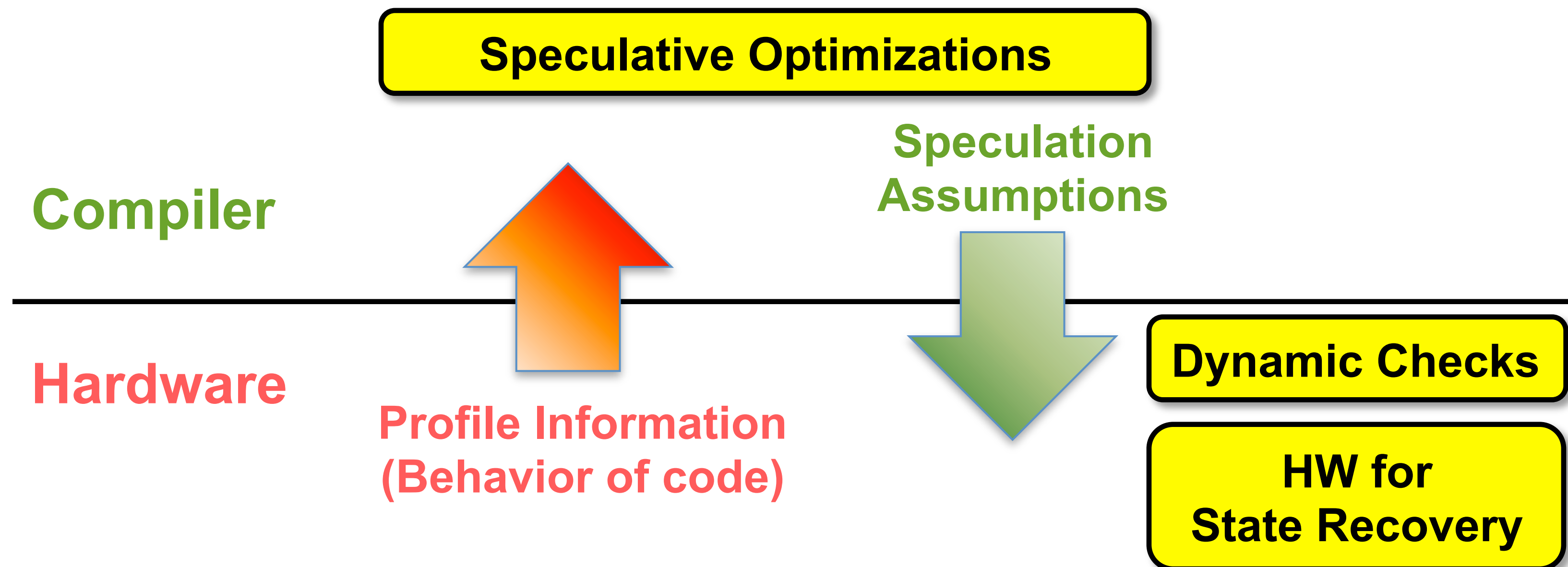
Will JIT / Scripting be Popular with IoT?

- Reasons IoT devices will want to use JIT:
  - JIT languages are **easier to use**
    - Big data scientists use Python, R, Matlab
  - JIT languages are **portable**
    - JavaScript is the only language that runs on all devices
  - JIT languages are more **secure**
    - Executes in a safe managed environment
- However many IoT devices still use C
  - Many IoT devices are battery-powered or energy-harvesting
  - C is more energy / memory efficient than JIT languages
  - Challenge: how to make JIT / Scripting efficient

Pitt

My Answer: Hardware/Compiler Co-Optimization

- One advantage JIT compilers have: **profile information**
- But often limited by:
  - Cost of collecting profile information
  - Cost of checking if profile-based prediction is correct
  - Cost of recovering state when prediction is incorrect



My Contributions

Compiler

Live Code Update
for IoT Devices [NVMSA’16]

Boosting JavaScript Efficiency
by Rethinking the Type System [PLDI’14]

BulkCompiler:
Memory Ordering Speculation Across Threads Using TM Hardware [MICRO’09]

Hardware

Pitt

My Contributions

Compiler

Live Code Update
for IoT Devices [NVMSA'16]

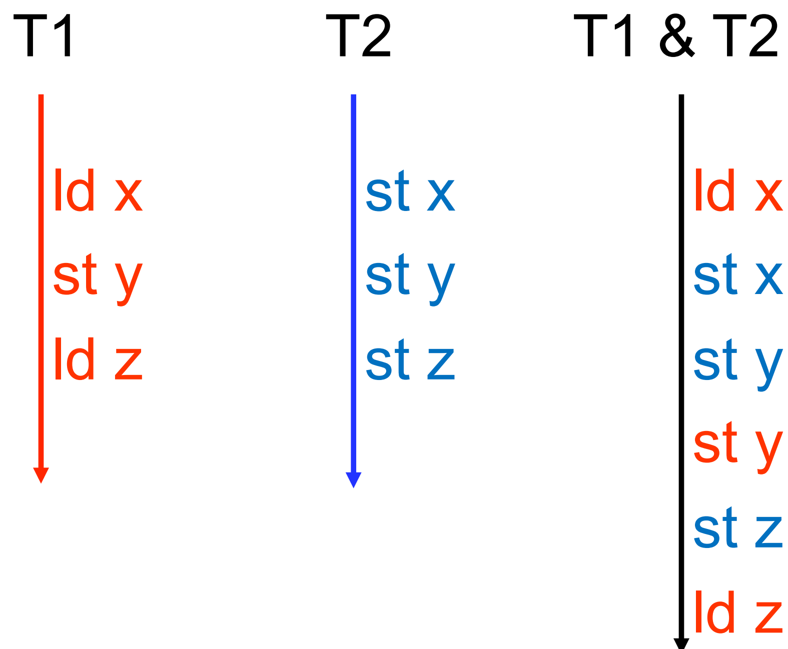
Boosting JavaScript Efficiency
by Rethinking the Type System [PLDI'14]

BulkCompiler:
Memory Ordering Speculation Across Threads Using TM Hardware [MICRO'09]

Hardware

Memory Consistency (Ordering) Model

- Memory Consistency Model:
  - Defines how memory accesses can interleave across threads
  - Sequential Consistency (SC): only model that is intuitive to humans



- Accesses must appear to:
  - Interleave in a single total order
  - Follow per-thread program order

SC: Ideal Language Memory Consistency Model

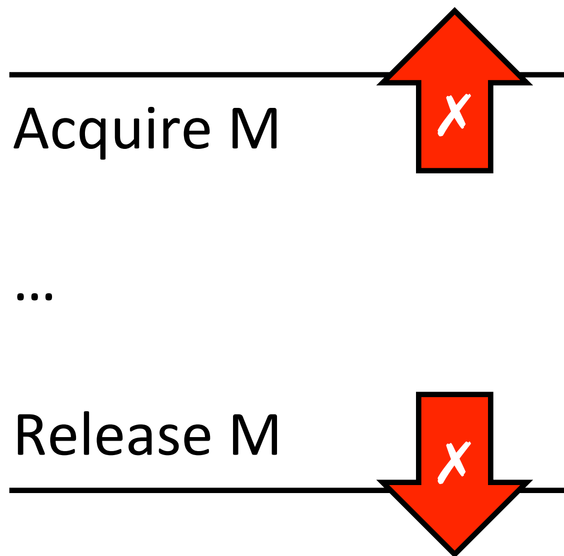
- Much easier to **debug**:
  - Only memory model programmers can reason about
  - Debuggers can reproduce buggy interleaving
- **Software verification** tools assume SC:
  - More relaxed memory models result in too many interleavings
- Provides fool-proof **secure** language memory model:
  - Java Memory Model (JMM) took years to design / verify
  - Complexity of JMM is the cause of insecure implementations

- Current memory consistency models are much more relaxed
  - To allow Compiler / HW reordering of memory accesses
  - For the purposes of performance optimization

Pitt

Current Memory Models Sometimes Prevent Optimizations

- All current memory models impose strict ordering around synchronization points (e.g. acquire / release)



✗ Illegal

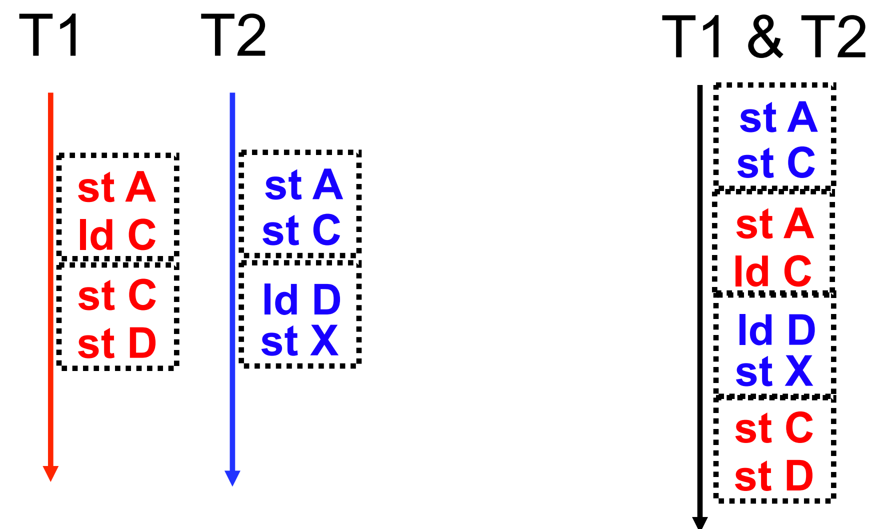
```
for (…) {
  Acquire M
  a = b + c;
  A[i] = a;
  Release M
}
```

- Current memory consistency models have two problems
  - Too relaxed: hard to program and debug
  - Too restrictive: prevents optimization across sync points

Pitt

Idea: Use Transactions for Memory Ordering Speculation

- Group chunks of dynamic instructions into transactions
- Commit transactions in a total order ➔ **SC is achieved**



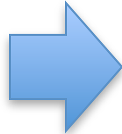
- **Higher performance** through memory ordering speculation
  - Within a transaction, Compiler / HW can freely reorder instructions
  - Transaction guarantees isolation (rollback if violated)
- Transactional Memory (TM) hardware is already commercialized
  - Intel Haswell, IBM Power 8, IBM Bluegene/Q

Pitt

BulkCompiler: Transformations on Synchronization

- **Low-contention** critical sections:
  - Group many of them in same transaction and **optimize**

beginAtomic
i_1
acquire M1
i_2
release M1
i_3
acquire M2
i_4
release M2
i_5
endAtomic



beginAtomic
i_1
while (M1 == taken) { }
i_2
while (M1 == taken) { }
while (M2 == taken) { }
i_5 i_2 i_3 i_1
i_3 i_4
while (M2 == taken) { }
i_4
endAtomic
i_5
endAtomic

- Remove acquire / release
- **Optimize** and reorder the code

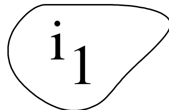
Speedups relative to current compilers

Pitt

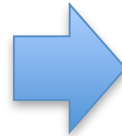
BulkCompiler: Transformations on Synchronization

- **High-contention** critical sections:
  - Tight-fit a transaction
  - Reduce chance of rollbacks

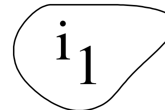
beginAtomic
acquire M1



release M1
endAtomic



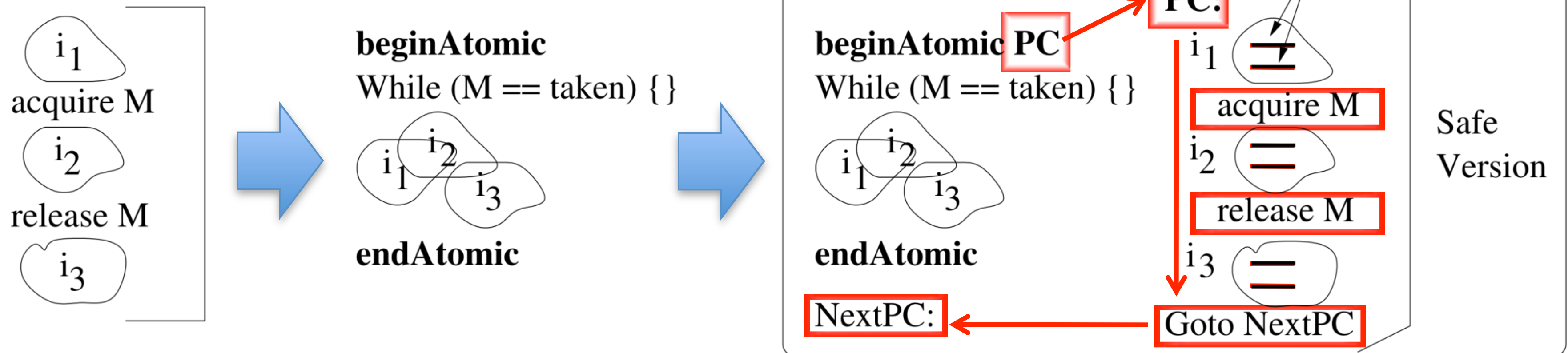
beginAtomic
while (M1 == taken) { }



endAtomic

➔ Prevent slowdowns due to chunk rollbacks

Guaranteeing Forward Progress: Safe Versions



- Semantically a duplicate of the transaction. But..
  - Does not use transactions to ensure SC (instead use fences)
  - Includes original synchronization accesses
- Transactions fall back on Safe Versions when forward progress problem
  - Safe Version is guaranteed forward progress (no rollbacks)

Pitt

BulkCompiler: Speedup over HotSpot Java Compiler

- Average speedup is 37% (23% from compiler opts)
- While still supporting SC

Pitt

BulkCompiler Summary

- Built BulkCompiler on top of Sun Hotspot Java Compiler
  - Optimizing inside transactions by leveraging atomicity
- **37% speedup** over original Sun Hotspot Java Compiler
  - While upgrading Java Memory Model to **Sequential Consistency**
  - Improving **programmability** and **security**

Pitt

My Contributions

Compiler

Live Code Update
for IoT Devices [NVMSA'16]

Boosting JavaScript Efficiency
by Rethinking the Type System [PLDI'14]

BulkCompiler:
Memory Ordering Speculation Across Threads Using TM Hardware [MICRO'09]

Hardware

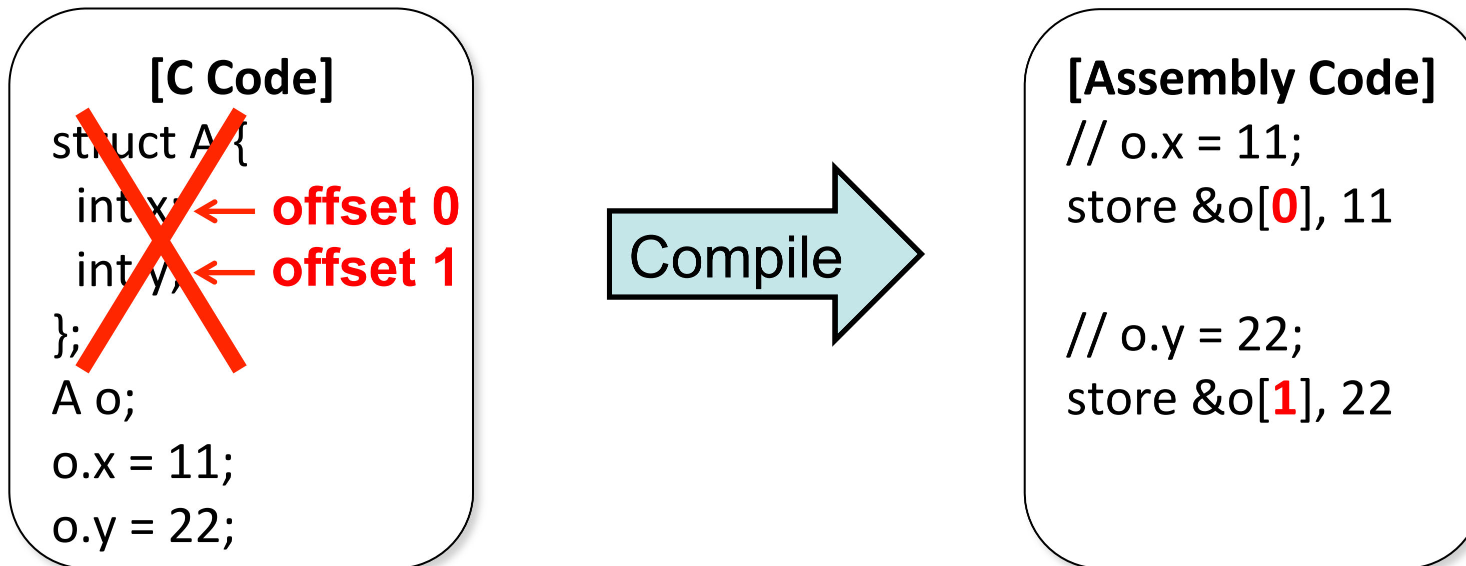
Pitt

Scripting Languages and Efficiency

- JavaScript, PHP, Python, Perl, Ruby, Lua, R
- Have common features that make programming easier but efficient code generation much much harder
  - **Dynamic Typing**
  - Dynamic Dispatch
  - First Class Functions
  - Runtime Evaluation (eval(‘string’))

Why Types Are Important for Performance

- Types are crucial to generating efficient code

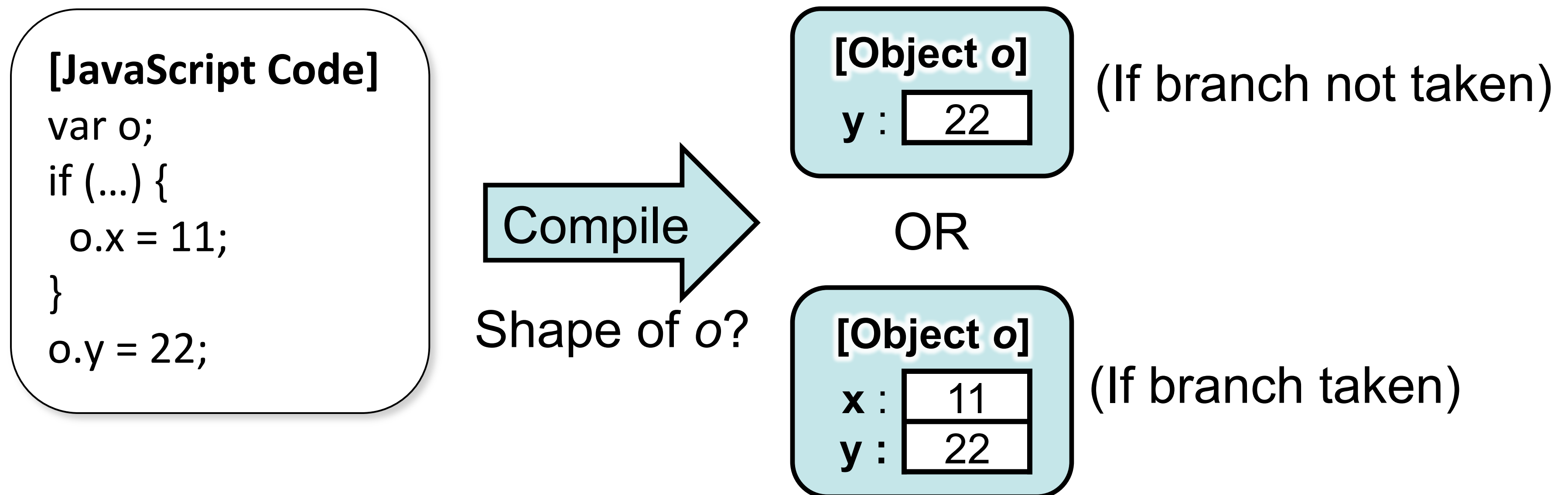


Types tell compiler the shape of *o* (fields and their offsets)

Pitt

Scripting Languages Have No Types ➔ Slow

- Objects are simply dictionaries from properties to values
- Properties can be added and removed at any time



Pitt

Scripting Languages Have No Types ➔ Slow

- Objects are simply dictionaries from properties to values
- Properties can be added and removed at any time

[JavaScript Code]

```
var o;
if (...) {
  o.x = 11;
}
o.y = 22;
```

Compile

Shape of *o*?

[Object *o*]
y : 22
type_ptr

[Type A]

Property	Offset
y	0

OR

[Object *o*]
x : 11
y : 22
type_ptr

[Type B]

Property	Offset
x	0
y	1

Hash tables

Compilers introduce a type system behind the scenes

Pitt

Scripting Languages Have No Types ➔ Slow

[Object *o*]
y : 22
type_ptr → [Type A]

[Type A]

Property	Offset
y	0

[Object *o*]
x : 11
y : 22
type_ptr → [Type B]

[Type B]

Property	Offset
x	0
y	1

[JavaScript Code]

```
var o;
if (…) {
  o.x = 11;
}
o.y = 22;
```

Compile ➔

[Object *o*]
y : 22
type_ptr → [Type A]

[Type A]

Property	Offset
y	0

OR

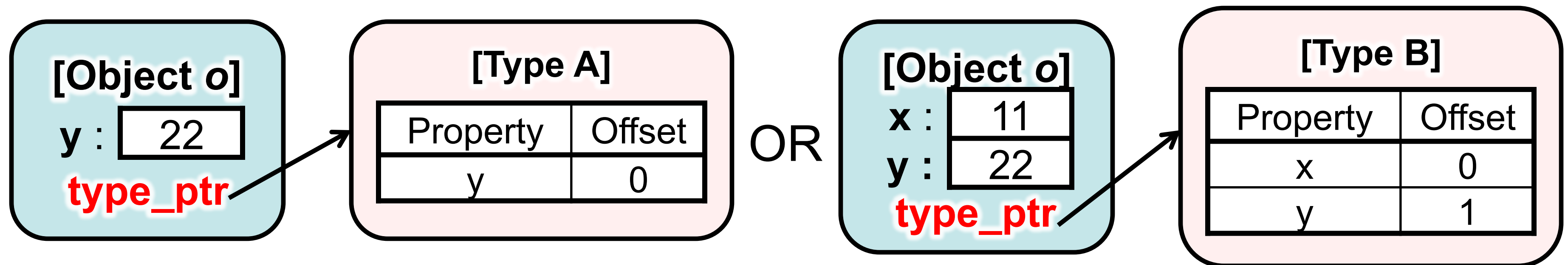
[Object *o*]
x : 11
y : 22
type_ptr → [Type B]

[Type B]

Property	Offset
x	0
y	1

Pitt

Scripting Languages Have No Types ➔ Slow



[JavaScript Code]

```
var o;
if (…) {
  o.x = 11;
}
o.y = 22;
```

Compile

[Pseudo Code for *o.y = 22*]

```
offset = lookup (o.type_ptr, "y");
store &o[offset], 22;
```

A field access always entails a **hash table lookup** to get the offset

Pitt

State-of-the-art Compilers do Better: Type Specialization Optimization

[Object *o*]
y : 22
type_ptr

[Type A]

Property	Offset
y	0

OR

[Object *o*]
x : 11
y : 22
type_ptr

[Type B]

Property	Offset
x	0
y	1

IC Miss
IC Hit
IC Hit

[JavaScript Code]

```
var o;
if (…) {
  o.x = 11;
}
o.y = 22;
```

Compile

[Pseudo Code for *o.y = 22*]

```
If (o.type_ptr == A)
  offset = 0;
else if (o.type_ptr == B)
  offset = 1;
else
  offset =  lookup (o.type_ptr, "y");
store &o[offset], 22
```

- **Type specialization**: Optimizing co… s
- **Inline Cache (IC)**: Actual optimized code
- **IC Hit**: Execution encounters one of the recorded types ➔ Fast
- **IC Miss**: Execution results in a hash table lookup ➔ Very Slow

Pitt

Google Chrome V8 JavaScript Compiler: Ineffective for Real Websites

- Researchers & Industry were focused on industry benchmarks
- Industry benchmarks
  - Kraken, Octane, Sunspider
  - Mostly compute intensive codes (e.g. raytracing, cryptography)
  - Do not reflect the behavior of most websites

Chrome V8 | Chrome V8 (Optimizations Disabled)

Normalized Instructions

100
10
1
0.1

84.6
43.6
31.9
1.4

yahoo-opera
google-opera
facebook-chrome
amazon-chrome
google-chrome
google-firefox
yahoo-firefox
amazon-opera
twitter-webkit
website-average
kraken-average
octane-average
sunspider-average

Chrome V8 compiler not optimized for dynamism in real websites

Pitt

Problem: V8 Type System Too Brittle [PLDI 2014]

- Chrome V8 type system encodes (other than property offsets):
  - **Inheritance** (i.e. address of parent object)
  - **Method bindings** (i.e. addresses of method properties)
- Helps in generating efficient code during type specialization
  - Inheritance: helps when accessing parent properties
  - Method bindings: helps resolve targets for method calls
- V8 Assumption: inheritance and method bindings rarely change
  - Reasonable since always true for statically typed languages

Reality: Assumption **NOT TRUE** for dynamic website code
➜ Leads to type dynamism and frequent inline cache misses (and terrible performance)

Pitt

Solution: Decouple Inheritance from Type System

- Decoupling Inheritance from the Type System

[Object *child₁*]
x : 11
type_ptr
parent_ptr

[Type A]

Property	Offset

[Type A+B]

Property	Offset
x	0

Property	Offset
x	0

parent_ptr

[Object *parent₁*]
y : 22
type_ptr

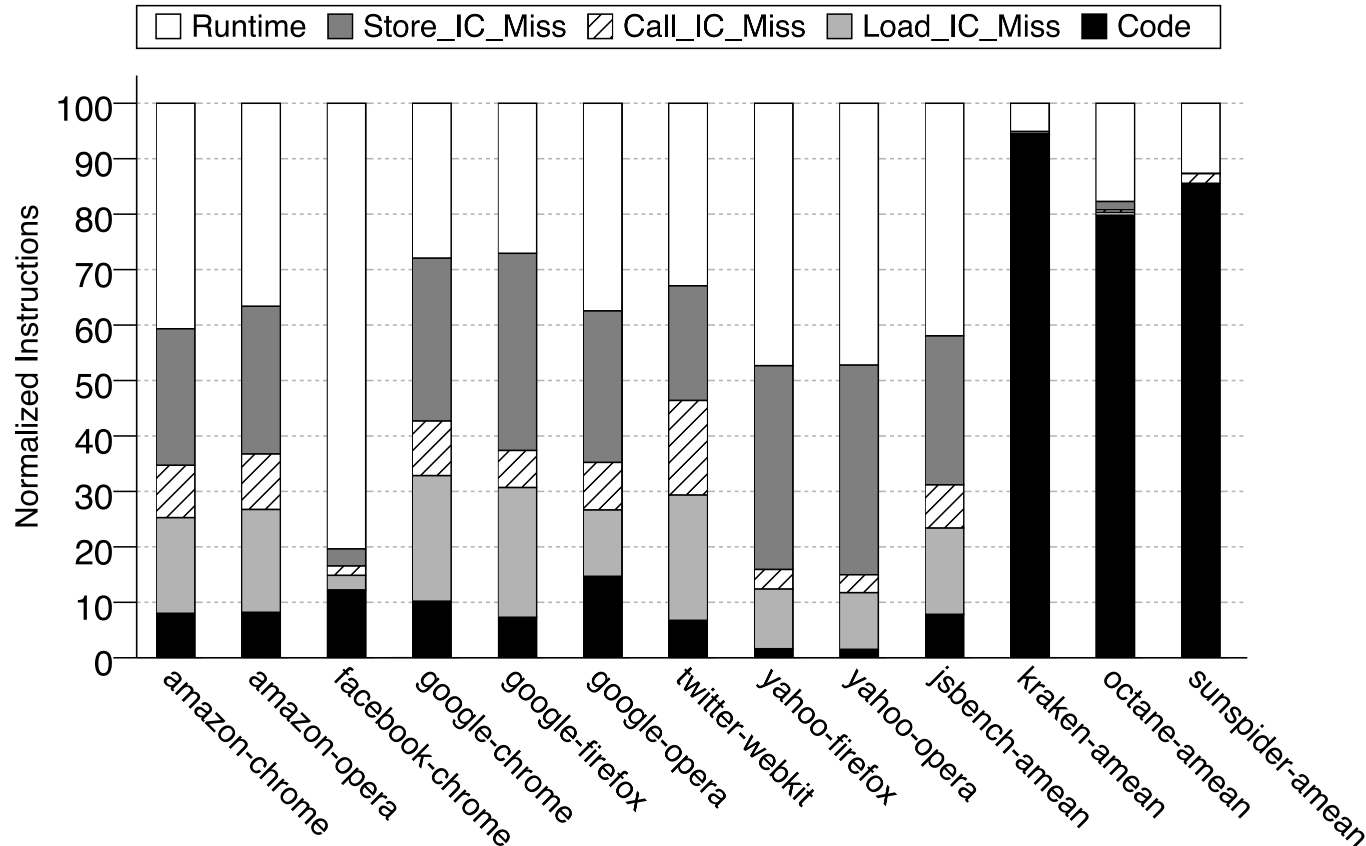
[Object *child₂*]
x : 33
type_ptr
parent_ptr

[Object *parent₂*]
y : 44
type_ptr

- Types A & B have exactly same shapes but different types
- Leads to needless type dynamism and inline cache misses

Pitt

Instruction Breakdown for Chrome V8



- JSBench: 50% of instructions spent on inline cache miss handling
- Kraken, Octane, Sunspider: Most instructions spent on generated code

Pitt

Instruction Count Reduction over Chrome V8

Runtime
Store_IC_Miss
Call_IC_Miss
Load_IC_Miss
Code

Normalized Instructions

0 10 20 30 40 50 60 70 80 90 100 110

B B* P C

amazon-chrome amazon-opera facebook-chrome google-chrome google-firefox google-opera twitter-webkit yahoo-firefox yahoo-opera jsbench-average

- Almost all instructions due to inline cache misses removed
- Total instruction count decreased by 49% compared to Chrome V8
- Negligible change for Kraken, Octane, Sunspider (not shown)

Summary

- Made the type system of Chrome V8 JS Compiler more flexible:
  - By decoupling inheritance and method binding from types

Results in:

49% in dynamic instruction count reduction

20% in heap memory usage reduction

Pitt

My Contributions

Compiler

Live Code Update
for IoT Devices [NVMSA'16]

Boosting JavaScript Efficiency
by Rethinking the Type System [PLDI'14]

BulkCompiler:
Memory Ordering Speculation Across Threads Using TM Hardware [MICRO'09]

Hardware

Pitt

Code Update for IoT Devices Design Goals

- Minimize **device down time**
  - Consists of reboot time after code update
  - Prolonged down time can impact QoS
  - For energy harvesting devices, it may take days
- Minimize **reprogramming time**
  - Consists of code transfer time + code write time
  - Important in iterative debug / tuning scenarios
  - For energy harvesting devices, it may take weeks
- Minimize **extra code memory**
  - Consists of memory to store new updated image
  - Larger memory means more costly devices

Pitt

Previous Code Update Approaches

Reprogramming time

- Naïve:
  1. Rebuild code image at base station
  2. Transfer image wirelessly to each IoT device
  3. Write image in separate area in code memory in device
  4. Reboot device from image

Extra memory

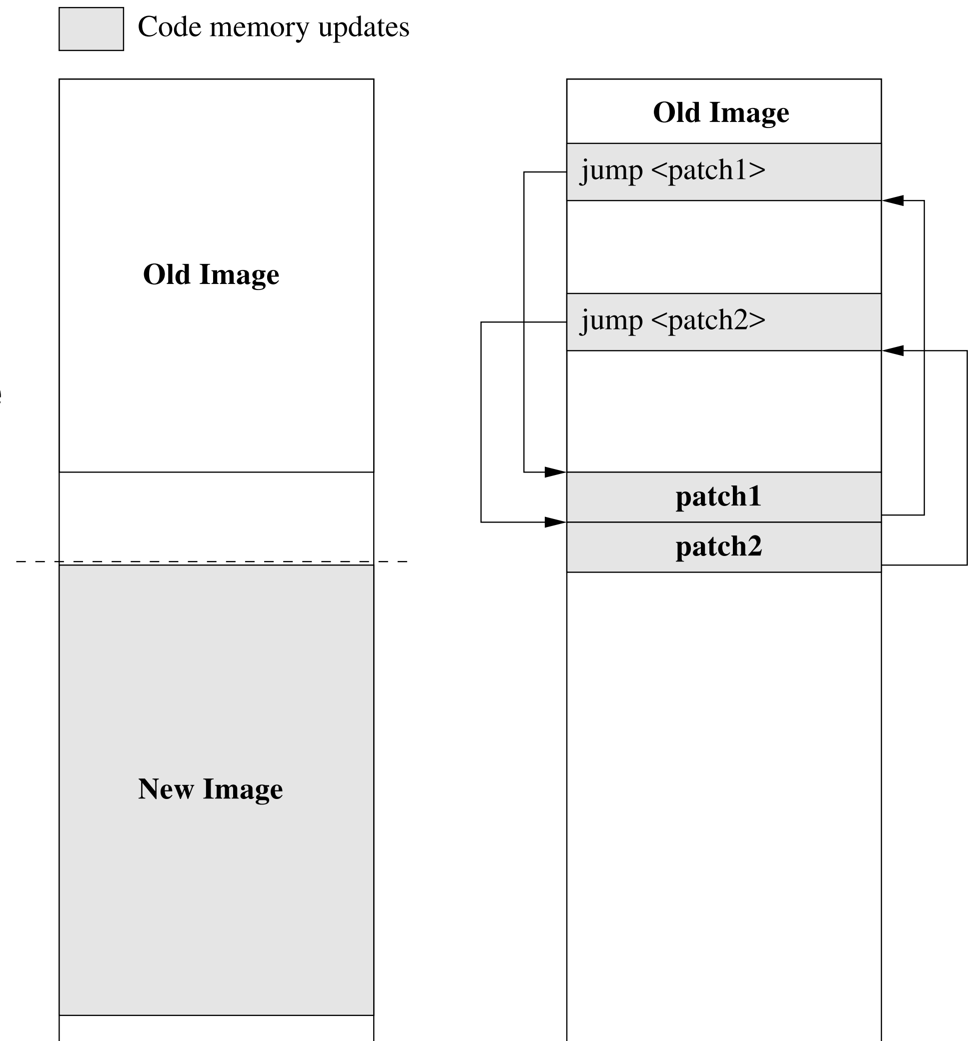
Device down time

- Incremental: Improves upon naïve by transferring only diff
  - Decreases transfer time
  - But must still rewrite image and reboot

Pitt

Our Code Update Approach [NVMSA 2016]

- Live update:
  - Apply patches to **live** image in device **in-place**
  - No down time
  - Minimal reprogramming time
  - Minimal extra memory



Previous Approaches **Our Approach**

Pitt

Atomic Code Patching

- Problem: Need to perform all patches **atomically**
  - Cannot expose live code in an inconsistent state
- Solution: Use **indirect jump vectors**
- Indirect jump vectors
  - Table holding addresses pointing to pieces of code
  - Used to implement interrupt service routines in OSes

➜ Idea: change functionality instantaneously by switching over from one jump vector table to another jump vector table
➜ Perform the switch through an (atomic) **single word write**

Pitt

Phase 1: Trampoline Insertion

Code updates that need to be atomic

Code updates that do not need to be atomic

Inserted code

Deleted code

Modified code

jump <trampoline1>

return1:

trampoline1: load R1, [<base>]
load R2, [R1 + 0]
jump R2

patch1:
jump <return1>

jump <trampoline2>

temp2:

return2:

trampoline2: load R1, [<base>]
load R2, [R1 + 1]
jump R2

patch2: jump <return2>

jump <trampoline3>

temp3:

return3:

trampoline3: load R1, [<base>]
load R2, [R1 + 2]
jump R2

patch3:
jump <return3>

Global Variable Memory

base	orig_targets	patch_targets
orig_targets	return1	patch1
	temp2	patch2
	temp3	patch3

Phase 2: Trampoline Switching

Code updates that need to be atomic

Code updates that do not need to be atomic

Inserted code

Deleted code

Modified code

return1:

jump <trampoline1>

trampoline1:
load R1, [<base>]
load R2, [R1 + 0]
jump R2

patch1:

jump <return1>

temp2:

jump <trampoline2>

return2:

Single Word Write

trampoline2:
load R1, [<base>]
load R2, [R1 + 1]
jump R2

patch2:
jump <return2>

temp3:

jump <trampoline3>

return3:

trampoline3:
load R1, [<base>]
load R2, [R1 + 2]
jump R2

patch3:

jump <return3>

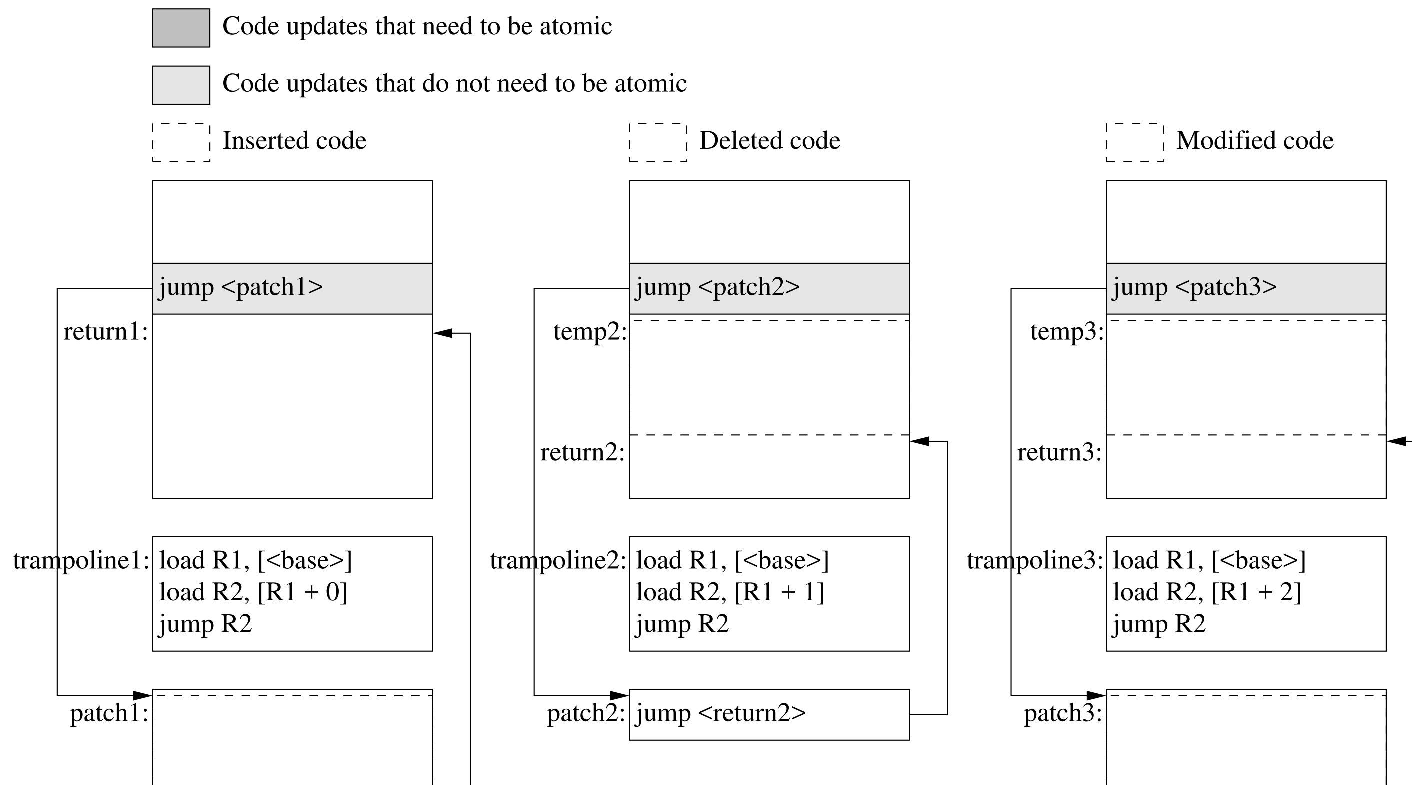
Global Variable Memory

base: patch_targets

orig_targets	patch_targets
return1	patch1
temp2	patch2
temp3	patch3

Pitt

Phase 3: Trampoline Removal



- ➜ Control flow is streamlined for better performance
- ➜ Trampolines are recycled to implement next patch

Pitt

Reprogramming Time Reduction

Total Energy (mJ)
2.5
2
1.5
1
0.5
0
reboot
transfer
write
1 2 3 4 5 6 7 8 9
RSync (Incremental)
1 2 3 4 5 6 7 8 9
Live Patching

- Reboot eliminated for live patching (no device down time)
- Transfer & write reduced due to no code shifts for live patching

Pitt

Summary

- Proposed live code update scheme for IoT devices that can:
  - Improve QoS by **eliminating down time**
  - Improve usability by **minimizing reprogramming time**
  - With minimal runtime overhead (< 2.7%)

Pitt

Future Work: Hardware/Compiler co-optimization

- **Scripting on IoT** for improved **security** / **programmability**
  - Re-engineer VM to fit in limited device memory
    - Offload compilation to base station
    - Device VM needs only runtime (garbage collector etc.)
  - More efficiency improvements through SW / HW co-design
    - Decrease heap usage through HW-assisted compiler optimizations such as stack promotion
    - Energy efficient GC through co-design
  - Security improvements through SW / HW co-design

Pitt

Web Browser Sandboxing – Same Origin Policy

Main Website

Web Server Code

Ad Website

Origin:
Website.net

Origin:
Ad.net

Main iframe

JavaScript Code

DOM

Cookies

?

Ad iframe

Client Web Browser

Pitt

Conventional Vulnerability – Cross Site Scripting

Main Website

print “<html>”
print ”You searched for:”
print request.query[‘keyword’]
print “</html>”

Malicious Website

GET http://website/search?keyword=**<script>…</script>**

Check this out: http://website/search?keyword=**<script>…</script>**

Main iframe

<html>
You searched for:
<script>…</script>
</html>

Ad iframe

Client Web Browser

Pitt

New Vulnerability with IoT – Holes in the Sandbox

hybrid application (web code)
iframe
malicious advertiser
advertising broker
embedded web browser
bridge
hybrid framework
bridge
device resources

Pitt

Future Work: Hardware/Compiler co-optimization

- **Bytecode as ISA**
  - Design rich bytecode that expresses
    - More semantic / behavioral information
    - Security expectations
    - Fault tolerance expectations
    - Quality of Service expectations
    - Approximate computing expectations
  - JIT compile bytecode into underlying hardware
    - Use combination of compiler / HW to meet expectations
    - Use HW support for profiling / runtime checking / speculation