How Did We Get Here and Where Can We Go Next?

Laurence Tratt

KING'S College
LONDON

Software Development Team
2019-10-22
What happens if we’re wrong?
VM development is a destination
VM development is a destination

Completeness

Time

http://soft-dev.org/
VM development is a destination

Completeness vs Time
VM development is a destination

![Graph showing completeness over time.](http://soft-dev.org/)
VM development is a destination
VM development is a destination

![Graph showing the relationship between time and completeness. The graph depicts a curve that starts almost flat and then rises steeply to a plateau.](http://soft-dev.org/)
VM development is a destination

![Graph showing the relationship between Time and Perfection with Completeness as a vertical axis. The graph illustrates the concept that perfection is a destination that is never fully reached.](http://soft-dev.org/)
VM development is a destination
VM development is a never-ending process
We know where the performance ceiling is
We know where the performance ceiling is

What is the best possible performance for an input $P$?
We know where the performance ceiling is
We know where the performance ceiling is
We know where the performance ceiling is
We know where the performance ceiling is
We know where the performance ceiling is

Performance

Me

Other2

Other1

Other3

?
We know where the performance ceiling is
We know where the performance ceiling is
We know where the performance ceiling is

We don’t know how well we’re doing
We’re good at optimising abstractions
Most optimisations are ad-hoc and/or unpredictable
Most optimisations are ad-hoc and/or unpredictable

e.g. mono $\rightarrow$ poly $\rightarrow$ megamorphic JS calls
We’re good at optimising abstractions

How to communicate optimisations to users?
We know what the impact of individual features is
We know what the impact of individual features is.

What is the effect of e.g. pointer tagging?
We know what the impact of individual features is

What is the effect of e.g. pointer tagging?

GC and register allocation
only deeply studied topics?
We know what the impact of individual features is.

Hardware
We know what the impact of individual features is.

Hardware: caches
We know what the impact of individual features is

Hardware: caches, predictors
We know what the impact of individual features is.

Hardware: caches, predictors, temperature
We know what the impact of individual features is.

Hardware: caches, predictors, temperature, etc.
We know what the impact of individual features is

Hardware: caches, predictors, temperature, etc.

OS
We know what the impact of individual features is

Hardware: caches, predictors, temperature, etc.

OS: other processes
We know what the impact of individual features is

Hardware: caches, predictors, temperature, etc.

OS: other processes, context switches
We know what the impact of individual features is:

**Hardware:** caches, predictors, temperature, etc.

**OS:** other processes, context switches, etc.
We know what the impact of individual features is.

Hardware: caches, predictors, temperature, etc.

OS: other processes, context switches, etc.

VM
We know what the impact of individual features is.

Hardware: caches, predictors, temperature, etc.

OS: other processes, context switches, etc.

VM: compilation heuristics
We know what the impact of individual features is.

**Hardware:** caches, predictors, temperature, etc.

**OS:** other processes, context switches, etc.

**VM:** compilation heuristics, GC heuristics
We know what the impact of individual features is.

Hardware: caches, predictors, temperature, etc.

OS: other processes, context switches, etc.

VM: compilation heuristics, GC heuristics, etc.
We know how features interact
Good news:
Good news: we do machine learning!
We know how features interact

Fannkuch Redux, LuaJIT, OpenBSD, Proc. exec. #14 (warmup)
Bad news:
Bad news: we do machine learning!
We know how features interact

Richards, HotSpot, Linux_{E3-1240v5}, Proc. exec. #8 (slowdown)
We know how features interact

Binary Trees, V8, Linux, Proc. exec. #15 (warmup)

Binary Trees, V8, Linux, Proc. exec. #6 (no steady state)
We know how features interact

Performance non-determinism is rife
Performance non-determinism
Performance non-determinism
Performance non-determinism
Performance non-determinism

- Graphs representing performance over time.
Performance non-determinism
Solution to performance non-determinism
Solution to performance non-determinism: non-determinism?
We understand how large systems perform
We understand how large systems perform

Microbenchmarks behave poorly
We understand how large systems perform

Microbenchmarks behave poorly

But that doesn’t affect big benchmarks
We understand how large systems perform

How convenient!
We understand how large systems perform

What about compositionality?
Multi-language VMs
Multi-language VMs

VMs are expensive to create
Multi-language VMs

VMs are expensive to create

Why not reuse that hard work?
VMs are expensive to create

Why not reuse that hard work?

CPython vs. Jython parable
Semantic mismatch
Semantic mismatch

Generic  Specific
Semantic mismatch

x86

Generic  Specific
Semantic mismatch

x86
Generic

Python bytecode
Specific
Semantic mismatch

x86

Generic

JVM bytecode

Specific

Python bytecode
Semantic mismatch

Generic Specific

x86  JVM bytecode  Python bytecode

Specific
Semantic mismatch

Generic

x86

Continuations

JVM bytecode

Python bytecode

Specific
Semantic mismatch

x86

Feature \(_n\)

Generic

JVM bytecode

Specific

Python bytecode
Semantic mismatch

- x86
- JVM bytecode
- Python bytecode

Generic

Specific

http://soft-dev.org/
Semantic mismatch

Generic

x86

Specific

Java

Python

bytecode

bytecode

11 / 20 HTTP://SOfT-DEV.ORG/
One solution: language design tweaks
WASM will not solve the semantic mismatch
WASM will not solve the semantic mismatch

Meta-VMs suffer much less
My benchmark suite is good, yours is bad
Fix memory leak in pdfjs.js. #42

Itratt wants to merge 2 commits into chromium:master from Itratt:master

Itratt commented on Oct 2, 2016 • edited

A large amount of data is pushed into the global variable canvas_logs which isn’t cleared in runPdfJS. On each iteration the list grows, eventually significantly so.

On a Linux machine with a recent-ish V8, it manages 2777 iterations before an allocation fails (at which point it’s allocated over 2GiB of virtual memory, and used about 1.4GiB) and V8 crashes (Fatal error in CALL_AND_RETRY_LAST).
Fix memory leak in pdfjs.js. #42

lttratt wants to merge 2 commits into chromium:master from ltratt:master

ltratt commented on Oct 2, 2016 • edited

A large amount of data is pushed into the global variable canvas_logs which isn't cleared in runPdfJS. On each iteration the list grows, eventually significantly so.

On a Linux machine with a recent-ish V8, it manages 2777 iterations before an allocation fails (at which point it's allocated over 2GiB of virtual memory, and used about 1.4GiB) and V8 crashes (Fatal error in CALL_AND_RETRY_LAST).

...  

natorion commented on Feb 7, 2018 • edited

It won't be merged. Octane is retired and no longer maintained. Sorry for the long communication cycle.

natorion closed this on Feb 7, 2018
A year with Spectre: a V8 perspective

Published 23 April 2019 · tagged with security

On January 3, 2018, Google Project Zero and others disclosed the first three of a new class of vulnerabilities that affect CPUs that perform speculative execution, dubbed Spectre and Meltdown. Using the speculative execution mechanisms of CPUs, an attacker could temporarily bypass both implicit and explicit safety checks in code that prevent programs from reading unauthorized data in memory. While processor speculation was designed to be a microarchitectural detail, invisible at the architectural level, carefully crafted programs could read unauthorized information in speculation and disclose it through side channels such as the execution time of a program fragment.

We have experimented with (1) by inserting the recommended speculation barrier instructions, such as Intel’s LFENCE, on every critical conditional branch, and by using retpolines for indirect branches. Unfortunately, such heavy-handed mitigations greatly reduce performance (2–3× slowdown on the Octane benchmark). Instead, we chose approach (2), inserting mitigation sequences that prevent reading secret data due to mis-speculation. Let us illustrate the technique on the following code snippet:
A year with Spectre: a V8 perspective
Published 23 April 2019 · tagged with security

On January 3, 2018, Google Project Zero and others disclosed the first three of a new class of vulnerabilities that affect CPUs that perform speculative execution, dubbed Spectre and Meltdown. Using the speculative execution mechanisms of CPUs, an attacker could temporarily bypass both implicit and explicit safety checks in code that prevent programs from reading unauthorized data in memory. While processor speculation was designed to be a microarchitectural detail, invisible at the architectural level, carefully crafted programs could read unauthorized information in speculation and disclose it through side channels such as the execution time of a program fragment.

- 
- 
- 

We have experimented with (1) by inserting the recommended speculation barrier instructions, such as Intel’s LFENCE, on every critical conditional branch, and by using retpolines for indirect branches. Unfortunately, such heavy-handed mitigations greatly reduce performance (2–3× slowdown on the Octane benchmark). Instead, we chose approach (2), inserting mitigation sequences that prevent reading secret data due to mis-speculation. Let us illustrate the technique on the following code snippet:
My benchmark suite is good, yours is bad
Benchmark suites a finite representation of infinite behaviour
My benchmark suite is good, yours is bad

Benchmark suites a finite representation of infinite behaviour

All benchmark suites are imperfect
My benchmark suite is good, yours is bad

Benchmark suites a finite representation of infinite behaviour

All benchmark suites are imperfect

We need more and more benchmarks!
Managed languages are safe
Managed languages are safe

Vulnerabilities By Year

- 2010: 1
- 2011: 3
- 2012: 59
- 2013: 80
- 2014: 69
- 2015: 56
- 2016: 17
- 2017: 17
- 2018: 17
- 2019: 17

Vulnerabilities By Type

- Execute Code: 17
- Denial of Service: 41
- Bypass Something: 35
- Overflow: 6
- Memory Corruption: 4
- XSS: 1
- Gain Information: 7
- Gain Privilege: 1

Source: https://www.cvedetails.com/product/19117/Oracle-JRE.html?vendor_id=93
Managed languages are safe

C/C++ aren’t very safe
Managed languages are safe

C/C++ aren’t very safe

And what about JITted code?
Managed languages are safe

C/C++ aren’t very safe

And what about JITted code?

Prediction: VM security apocalypse is possible
We’re stuck with C/C++

HAT ABOUT 2UST?

FOR 6-S

CAN WE MAKE IT SO?

14/20

HTTP://SOFT-DEV.ORG/
We’re stuck with C/C++

What about Rust?
We’re stuck with C/C++

What about Rust?

Not an obvious fit for VMs
We’re stuck with C/C++

What about Rust?

Not an obvious fit for VMs

Can we make it so?
VMs use dynamic dispatch extensively
Thin pointers for dynamic dispatch

VMs use dynamic dispatch extensively

use std::mem::size_of;

trait T {
}

fn main() {
    assert_eq!(size_of::<&bool>(), size_of::<&u128>());
    assert_eq!(size_of::<&bool>(), size_of::<usize>());
    assert_eq!(size_of::<&dyn T>(), size_of::<usize>() * 2);
}
let x: &dyn T = ...;
let (ptr, vtable) = unsafe {
    mem::transmute<_, (*mut u8, *mut u8)>(x)
};
Thin pointers for dynamic dispatch

`#[repr(C)]`
struct ThinPtr { objptr: *mut u8 }

impl ThinPtr {
    fn new<U>(o: U) -> ThinPtr
        where *const U: CoerceUnsized<*const (dyn T + 'static)>,
            U: T + 'static
    {
        let (dataptr, vtable) = unsafe { mem::transmute<_, (*mut u8, *mut u8)>(x) };  
        let objptr = malloc(size_of::<*mut u8>() + size_of::<U>());  
        unsafe { ptr::write(objptr, &vtable, size_of::<*mut u8>()) };  
        unsafe { ptr::write(objptr + 1, ptr, size_of::<U>()) };  
        ThinPtr { objptr }
    }
}

impl Deref for ThinPtr {
    type Target = dyn T;
    fn deref(&self) -> & (dyn T + 'static) {
        let vtable = unsafe { ptr::read(objptr, size_of::<*mut u8>()) };  
        unsafe { transmute::::<(*const _, *const _), _>((self.objptr + 1, vtable)) }
    }
}
#[repr(C)]
struct ThinPtr { objptr: *mut u8 }

impl ThinPtr {
    fn try_downcast<U: T>(&self) -> Option<&U> {
        let t_vtable = unsafe {
            transmute::<&dyn T, (*mut u8, *mut u8)>(ptr::null() as *const U) 
        };
        let vtable = unsafe { ptr::read(objptr, size_of::<*mut u8>()) };
        if vtable == t_vtable {
            Some(unsafe { &* (self.objptr + 1) as *const U })
        } else {
            None
        }
    }
}
#[narrowable_abgc(ThinObj)]

trait Obj {
}

struct VMInt { x: u64 }

impl Obj for VMInt {
}

fn f(v: ThinObj) {
    if let Some(o) = v.try_downcast::<VMInt>() {
        println!(o.x);
    }
}
Can we use Rust for VMs?
So far, so good
Can we use Rust for VMs?

So far, so good

Major challenge: idiomatic GC support
The security landscape is changing
The security landscape is changing

CHERI:
The security landscape is changing

CHERI: capabilities in 128-bit pointers
We’re good at explaining what we do
We’re good at explaining what we do

IronPython; Jython; Nuitka; Psyco; PyPy; Pyston; Shed Skin; Stackless; Starkiller; TrufflePython; Unladen Swallow; WPython; Zippy
We’re good at explaining what we do

IronPython; Jython; Nuitka; Psyco; PyPy; Pyston; Shed Skin; Stackless; Starkiller; TrufflePython; Unladen Swallow; WPython; Zippy

e.g. compiling to LLVM fails every time...
We’re good at explaining what we do

We often pretend trade-offs don’t exist
We’re good at explaining what we do

We often pretend trade-offs don’t exist

Huge burden for newcomers to the field
Where can we go next?
Where can we go next?

We understand less than we should
Clear problems
Where can we go next?

Opportunities!