Ideas for improving meta-tracing warmup

Laurence Tratt

Lukas Diekmann

Edd Barrett

King's College London

Software Development Team

2021-06-09

Virtual Machine Warmup Blows Hot and Cold*

EDD BARRETT, King's College London, UK
CARL FRIEDRICH BOLZ-TEREICK, King's College London, UK
REBECCA KILLICK, Lancaster University, UK
SARAH MOUNT, King's College London, UK
LAURENCE TRATT, King's College London, UK

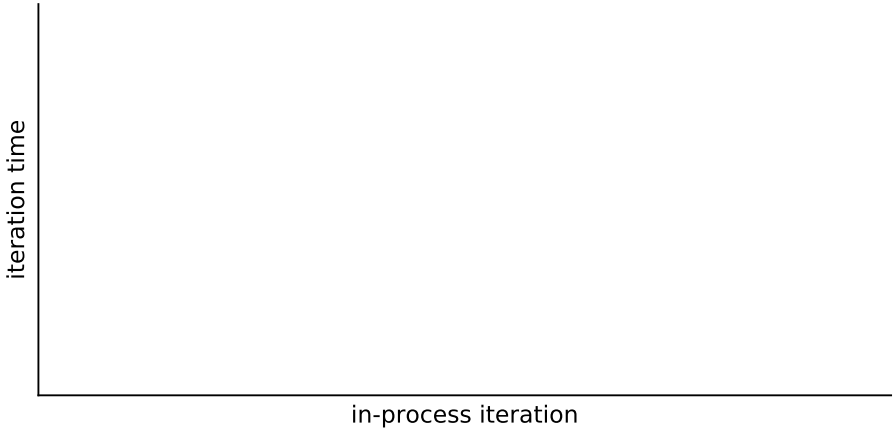
Virtual Machines (VMs) with Just-In-Time (JIT) compilers are traditionally thought to execute programs in two phases: the initial warmup phase determines which parts of a program would most benefit from dynamic compilation, before JIT compiling those parts into machine code; subsequently the program is said to be at a steady state of peak performance. Measurement methodologies almost always discard data collected during the warmup phase such that reported measurements focus entirely on peak performance. We introduce a fully automated statistical approach, based on changepoint analysis, which allows us to determine if a program has reached a steady state and, if so, whether that represents peak performance or not. Using this, we show that even when run in the most controlled of circumstances, small, deterministic, widely studied microbenchmarks often fail to reach a steady state of peak performance on a variety of common VMs. Repeating our experiment on 3 different machines, we found that at most 43.5% of ⟨VM, benchmark⟩ pairs consistently reach a steady state of peak performance.

CCS Concepts: • **Software and its engineering → Software performance**; **Just-in-time compilers**; *Interpreters*;

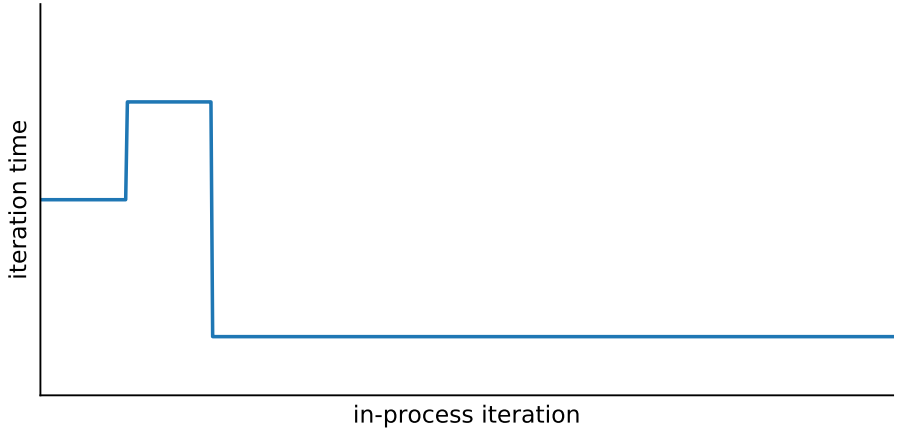
Additional Key Words and Phrases: Virtual machine, JIT, benchmarking, performance

ACM Reference Format:

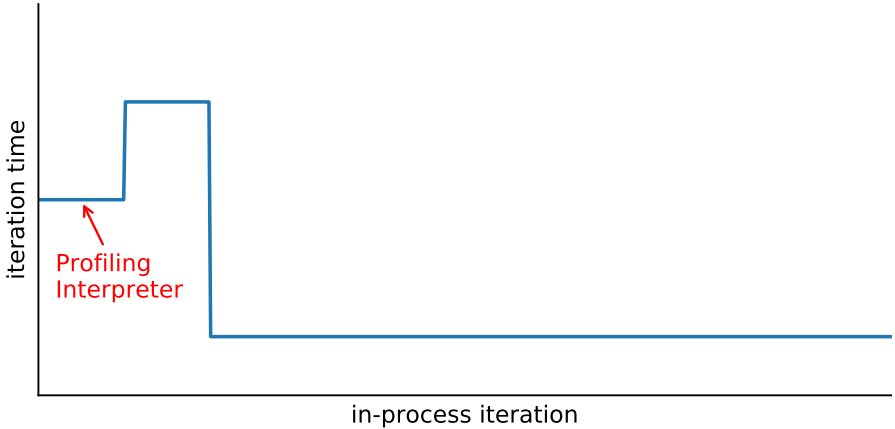
JIT Warmup



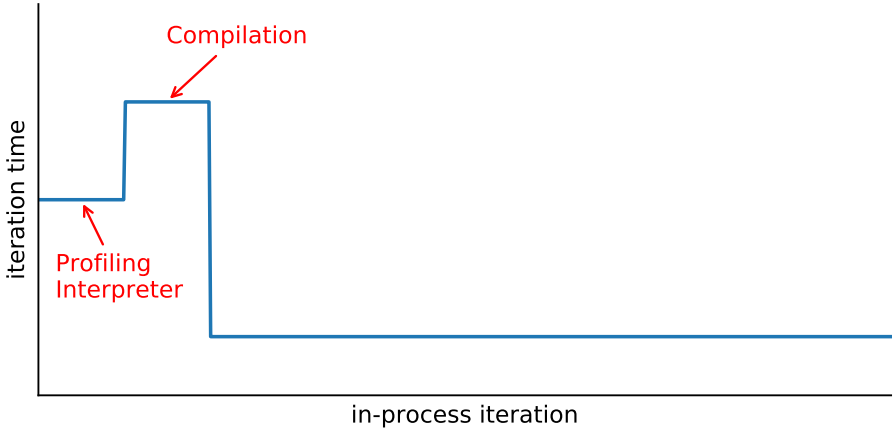
JIT Warmup



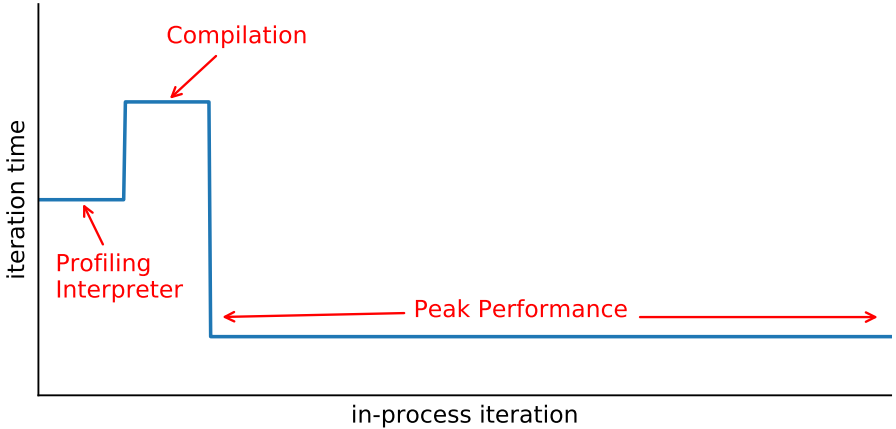
JIT Warmup


iteration time
Profiling
Interpreter
in-process iteration

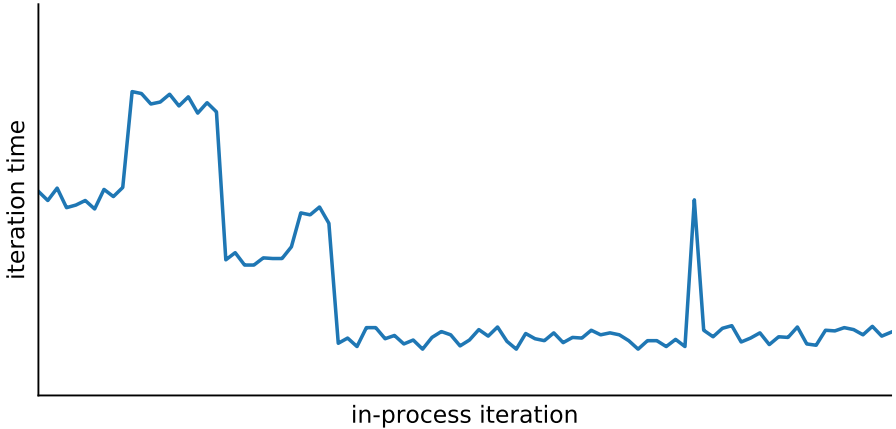
JIT Warmup


Compilation
iteration time
Profiling
Interpreter
in-process iteration

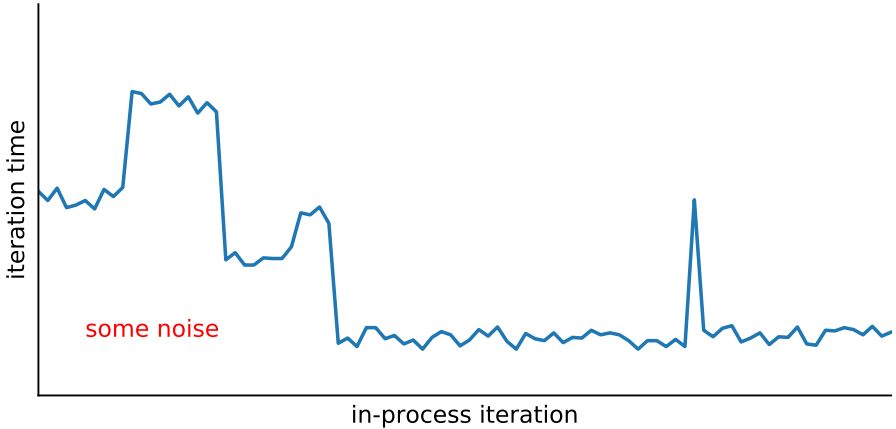
JIT Warmup


Compilation
iteration time
Profiling
Interpreter
Peak Performance
in-process iteration

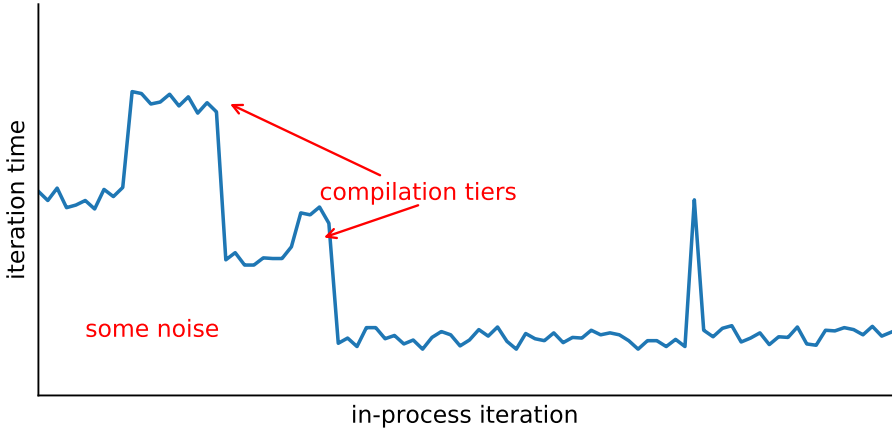
JIT Warmup



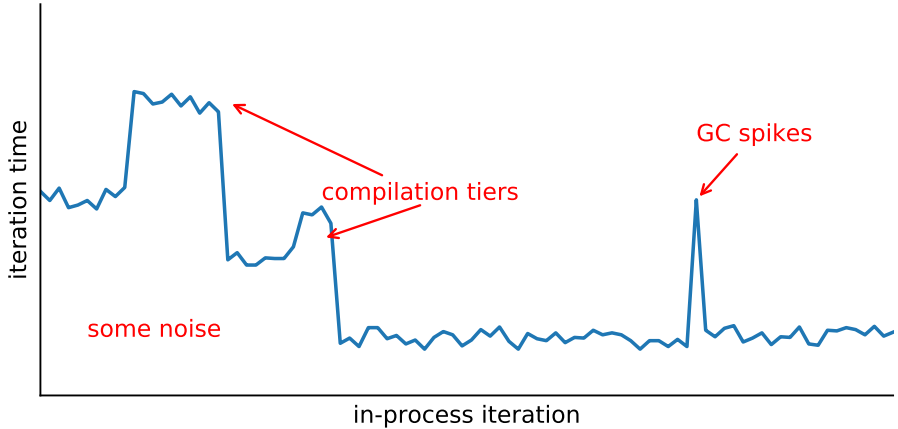
JIT Warmup



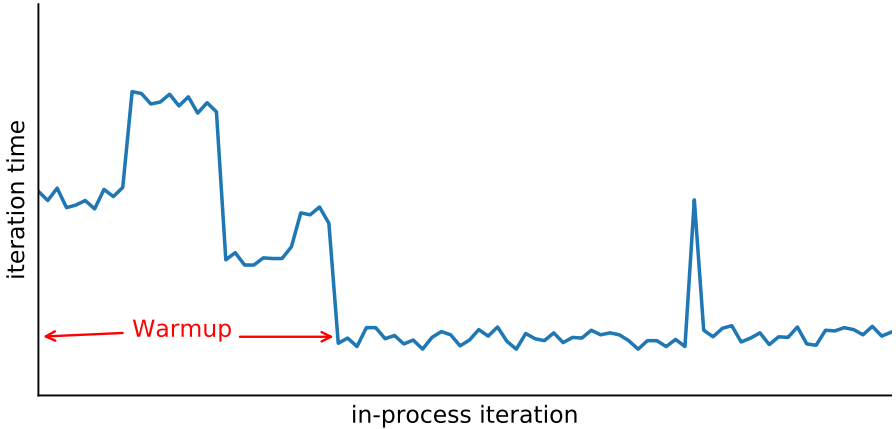
JIT Warmup


iteration time
compilation tiers
some noise
in-process iteration

JIT Warmup


iteration time
in-process iteration
compilation tiers
GC spikes
some noise

JIT Warmup



JIT Warmup

JIT Warmup

- Users dislike poor warmup.

JIT Warmup

- Users dislike poor warmup.
- VM authors dislike poor warmup.

We should measure the warmup of modern language implementations

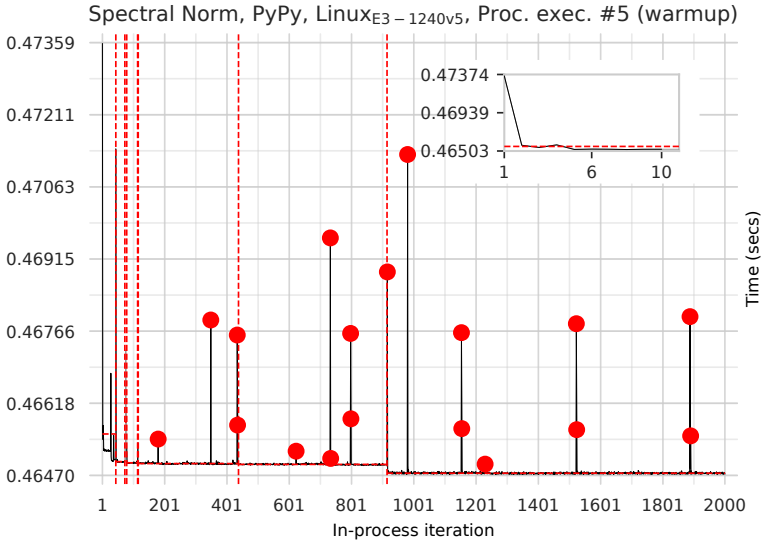
The Warmup Experiment

- Well-known micro-benchmarks
- State of the art JIT compilers
  - Graal, HHVM, TruffleRuby, Hotspot, LuaJIT, PyPy, V8
- 3 different machines
- Controlled benchmarking environment
- Ran the benchmarks for a long time
  - 2000 in-process iterations.

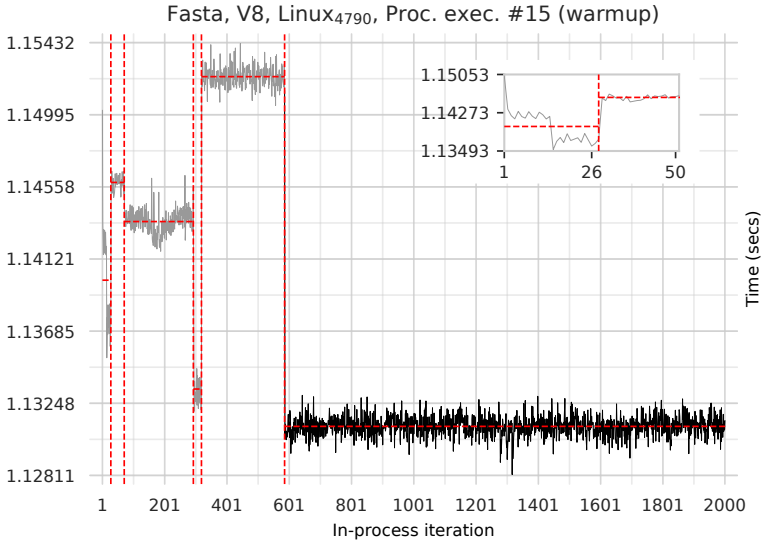
Hypothesis:

Small, deterministic programs reach a steady state of peak performance.

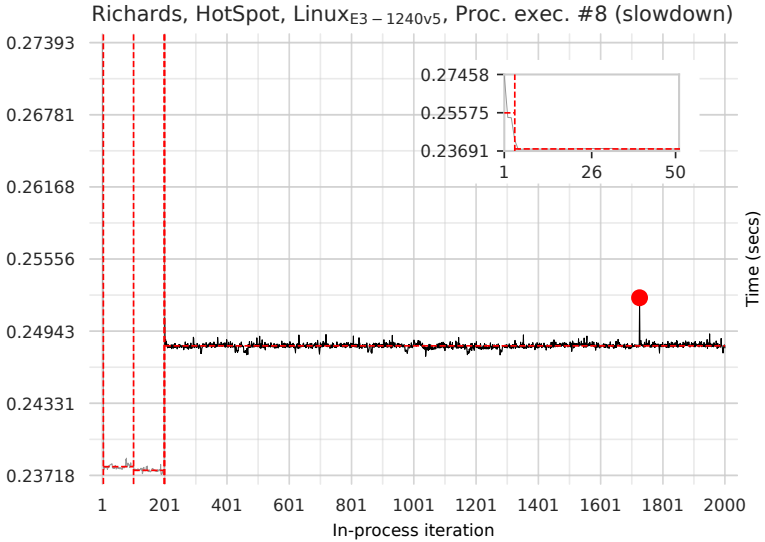
Results: Warmup



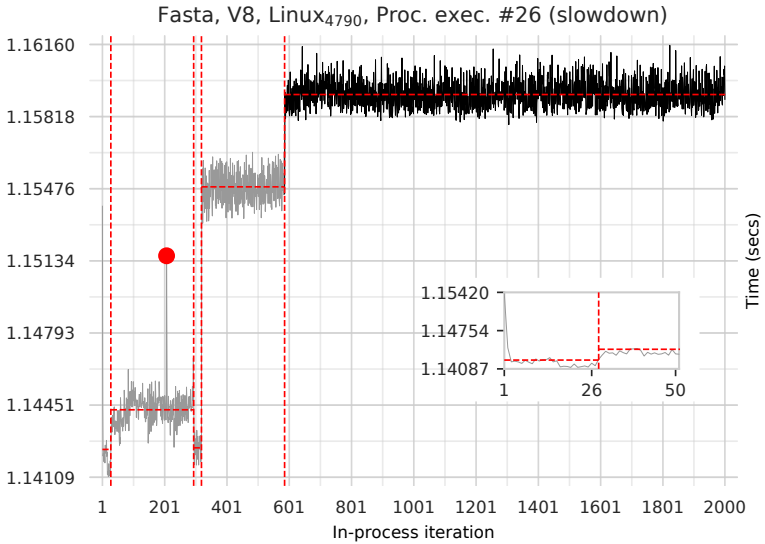
Results: Warmup



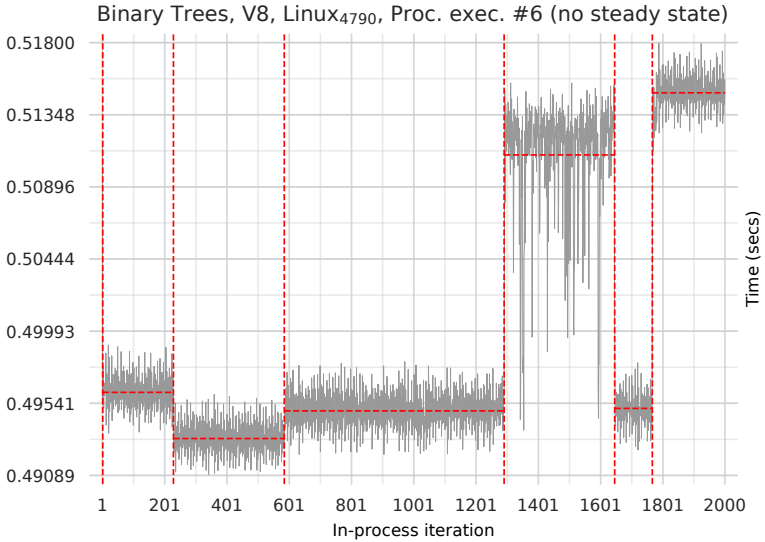
Results: Slowdown



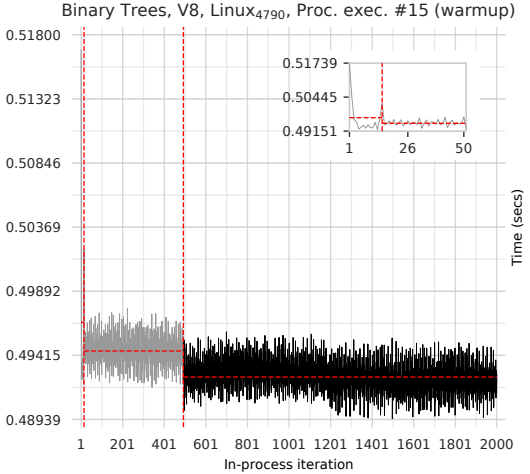
Results: Slowdown

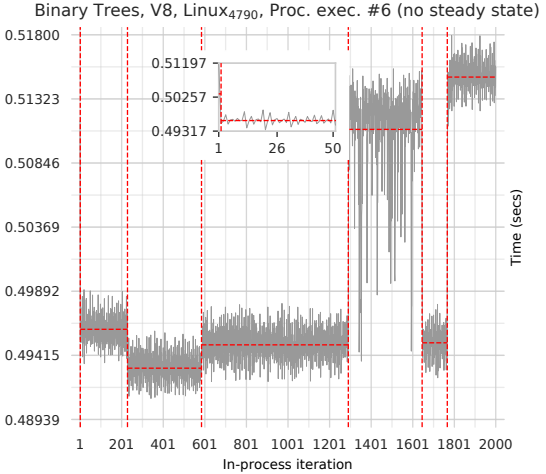


Results: No Steady State



Results: Inconsistent Process-executions





(Same machine)

Results

	Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)
C	↯					–			0.40555 ±0.005110
Graal	✖ (27𝕃, 3𝕁)	32.0 (17.0, 193.0)	6.60 (3.729, 36.608)	0.18594 ±0.000315		𝕃	8.0 (7.5, 8.5)	1.22 (1.126, 1.358)	0.13334 ±0.000045
HHVM	✖ (24𝕃, 4𝕁, 2●)					✖ (16𝕃, 11𝕁, 3●)			
HotSpot	✖ (25𝕃, 5𝕁)	7.0 (7.0, 53.5)	1.19 (1.182, 9.703)	0.18279 ±0.000116		𝕃	2.0 (2.0, 2.0)	0.14 (0.141, 0.143)	0.13699 ±0.000052
JRuby+Truffle	𝕁	1082.0 (999.0, 1252.5)	2219.59 (2059.304, 2516.021)	2.05150 ±0.017738		𝕃	69.0 (69.0, 70.0)	17.95 (17.716, 18.127)	0.20644 ±0.001568
LuaJIT	✖ (23𝕃, 4𝕁, 2–, 1●)					–			0.25399 ±0.004471
PyPy	✖ (27𝕁, 3●)					–			1.85835 ±0.012893
V8	✖ (15–, 9𝕃, 6𝕁)	1.5 (1.0, 794.0)	0.25 (0.000, 391.026)	0.49237 ±0.005198		= (25–, 5𝕃)	1.0 (1.0, 361.0)	0.00 (0.000, 87.367)	0.24138 ±0.000389
C	✖ (21–, 6𝕃, 2𝕁, 1●)					✖ (19–, 5𝕁, 4●, 2𝕃)			
Graal	✖ (28𝕃, 1●, 1𝕁)					✖ (28𝕃, 1●, 1𝕁)			
HHVM	𝕃	10.0 (10.0, 10.0)	52.66 (52.660, 52.700)	1.35779 ±0.011948					
HotSpot	𝕃	390.0 (2.0, 390.0)	153.70 (0.407, 153.234)	0.36202 ±0.022767		✖ (26𝕁, 2●, 2𝕃)			
JRuby+Truffle	𝕁	1016.5 (999.0, 1023.1)	1039.04 (1014.290, 1059.967)	1.08833 ±0.038580		𝕁	1021.0 (1014.9, 1027.0)	917.30 (901.708, 946.683)	0.89509 ±0.027590
LuaJIT	–			0.56285 ±0.000097		✖ (21●, 7𝕁, 2𝕃)			
PyPy	✖ (15𝕃, 13–, 2𝕁)	2.0 (1.0, 28.5)	1.57 (0.000, 43.483)	1.55442 ±0.020549		𝕃	2.0 (2.0, 3.0)	1.12 (1.114, 4.034)	0.96809 ±0.010926
V8	= (19𝕃, 11–)	2.0 (1.0, 25.5)	0.31 (0.000, 7.525)	0.30401 ±0.000154		= (29𝕃, 1–)	4.0 (4.0, 16.0)	1.44 (1.434, 7.135)	0.47421 ±0.001218
C	–			0.07048 ±0.000210		✖ (28𝕃, 1–, 1𝕁)	997.0 (2.0, 1001.0)	546.38 (0.546, 547.717)	0.54547 ±0.002562
Graal	✖ (29𝕃, 1●)					✖ (29𝕁, 1–)	15.0 (2.0, 21.0)	12.40 (0.812, 17.804)	0.89293 ±0.000687
HHVM	✖ (27𝕃, 2●, 1𝕁)					𝕃	35.0 (34.0, 41.0)	139.18 (136.909, 147.626)	1.40690 ±0.011135
HotSpot	✖ (18𝕃, 12𝕁)	261.0 (6.0, 595.0)	30.73 (0.614, 70.021)	0.11744 ±0.001723		𝕃	7.0 (7.0, 8.6)	1.90 (1.901, 2.425)	0.31470 ±0.000029
JRuby+Truffle						𝕃	1011.0 (1007.5, 1014.0)	893.38 (888.985, 896.562)	0.83633 ±0.014403
LuaJIT	↯					–			0.22435 ±0.000020
PyPy	↯					𝕃	75.0 (75.0, 75.0)	34.43 (34.429, 34.437)	0.46489 ±0.000046
V8	✖ (19𝕃, 10𝕁, 1●)					𝕃	3.0 (3.0, 3.0)	0.55 (0.554, 0.554)	0.24963 ±0.000039

binary trees · *n-body* · *famiktech redux* · *Richards* · *fasta* · *spectralnorm*

Results

		Class.	Steady iter (#)	Steady iter (s)	Steady perf (s)
C	*spectralnorm*	⪇ (27╘, 2╌, 1╒)	775.0 (1.5,780.0)	425.16 (0.246,426.809)	0.54581 ±0.033116
Graal		╒	14.0 (2.0,94.6) 	13.60 (0.830,98.737) 	1.05685 ±0.000126 
HHVM		⪇ (29╘, 1≈)			
HotSpot		╘	7.0 (7.0,7.5) 	1.91 (1.902,3.645) 	0.31472 ±0.169143 
LuaJIT		—			0.22181 ±0.000039
PyPy		= (27╌, 3╘)	1.0 (1.0,45.2) 	0.00 (0.000,20.597) 	0.46480 ±0.000085 
TruffleRuby		⪇ (25╒, 5≈)			
V8		╘	3.0 (3.0,3.0)	0.52 (0.523,0.526) 	0.25362 ±0.000034 

“Good warmup” (flat/warmup) occurs for only:

“Good warmup” (flat/warmup) occurs for only:

67.2%–70.3% of process executions

"Good warmup" (flat/warmup) occurs for only:

67.2%–70.3% of process executions

37.8%–40.0% of ⟨VM, benchmark⟩ pairs

~~Hypothesis:~~

~~Small, deterministic programs reach a steady state of peak performance~~

In the short term:

- Run benchmarks for longer to uncover issues.
- Be prepared to discard some of your results.
- Report the warmup time in results.

In the longer term:

- Find ways to improve warmup.

Improving Warmup

CHAPTER 35
INTEL® PROCESSOR TRACE

35.1 OVERVIEW

Intel® Processor Trace (**Intel PT**) is an extension of Intel® Architecture that captures information about software execution using dedicated hardware facilities that cause only minimal performance perturbation to the software being traced. This information is collected in **data packets**. The initial implementations of Intel PT offer **control flow tracing**, which generates a variety of packets to be processed by a software decoder. The packets include timing, program flow information (e.g. branch targets, branch taken/not taken indications) and program-induced mode related information (e.g. Intel TSX state transitions, CR3 changes). These packets may be buffered internally before being sent to the memory subsystem or other output mechanism available in the platform. Debug software can process the trace data and reconstruct the program flow.

Later generations include additional trace sources, including software trace instrumentation using PTWRITE, and Power Event tracing.

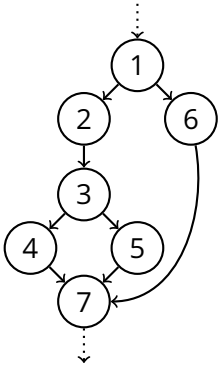
Intel Processor Trace

Last Branch Record (LBR) Low overhead, but can only record short traces.

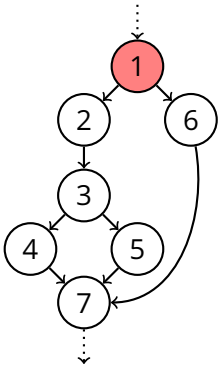
Branch Trace Store (BTS) Can record long traces, but high overhead.

Architecture Event Trace (AET) Requires specialised hardware probe.

Processor Trace (PT) Long traces, low *collection* overhead.

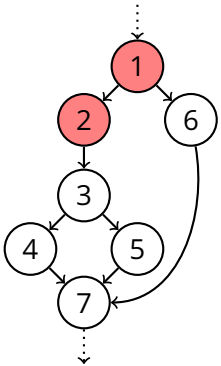


Profiling



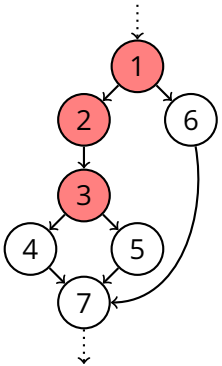
- interp(1)
- profile(1)

Profiling



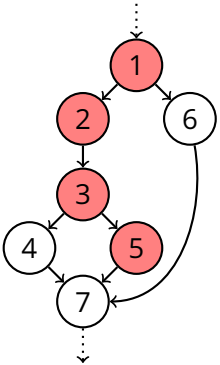
- interp(1)
- profile(1)
- interp(2)
- profile(2)

Profiling



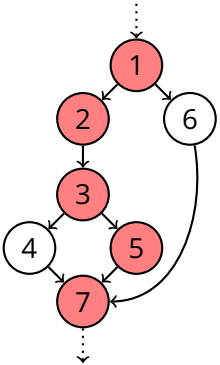
- interp(1)
- profile(1)
- interp(2)
- profile(2)
- interp(3)
- profile(3)

Profiling



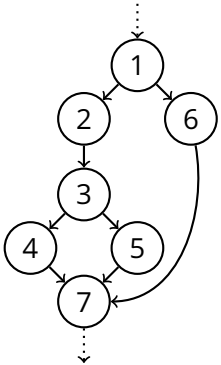
- interp(1)
- profile(1)
- interp(2)
- profile(2)
- interp(3)
- profile(3)
- interp(5)
- profile(5)

Profiling



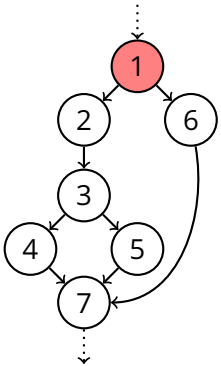
- interp(1)
- profile(1)
- interp(2)
- profile(2)
- interp(3)
- profile(3)
- interp(5)
- profile(5)
- interp(7)
- profile(7)

Profiling



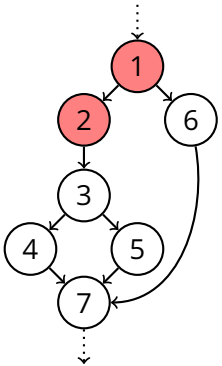
Tracing

Trace buffer = ⟨ ⟩



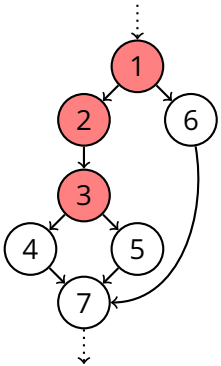
- interp(1)
- record(1)

Tracing Trace buffer = ⟨ 1 ⟩



- interp(1)
- record(1)
- interp(2)
- record(2)

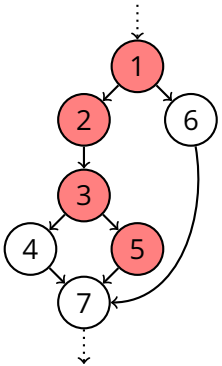
Tracing Trace buffer = ⟨ 1 , 2 ⟩



- interp(1)
- record(1)
- interp(2)
- record(2)
- interp(3)
- record(3)

Tracing

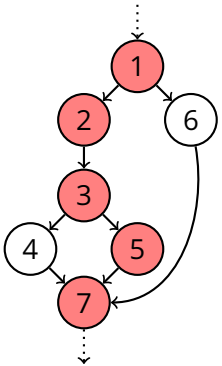
Trace buffer = ⟨ 1 , 2 , 3 ⟩



- interp(1)
- record(1)
- interp(2)
- record(2)
- interp(3)
- record(3)
- interp(5)
- record(5)

Tracing

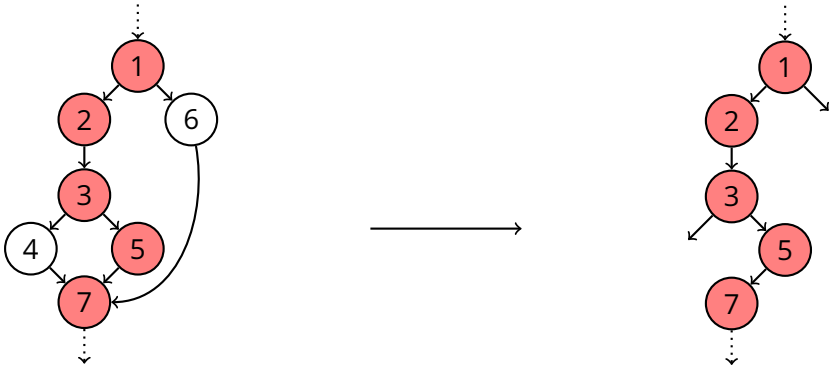
Trace buffer = ⟨ 1 , 2 , 3 , 5 ⟩



- interp(1)
- record(1)
- interp(2)
- record(2)
- interp(3)
- record(3)
- interp(5)
- record(5)
- interp(7)
- record(7)

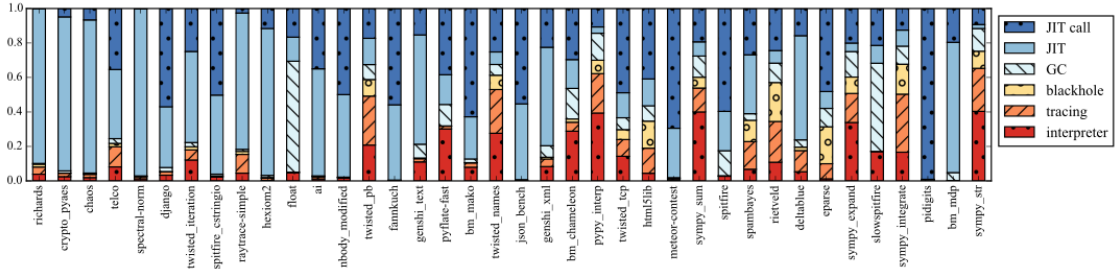
Tracing

Trace buffer = $\langle 1 , 2 , 3 , 5 , 7 \rangle$



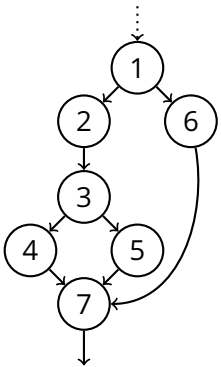
Compiling

Tracing JITs



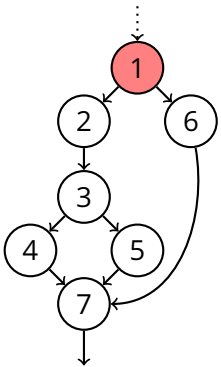
Cross-Layer Workload Characterization of Meta-Tracing JIT VMs. Ilbeyi et al.

PT-enabled Trace Collection



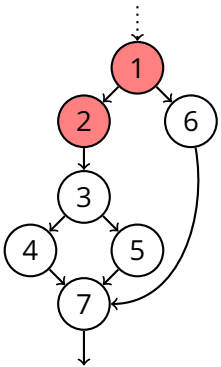
- Start Intel PT

PT-enabled Trace Collection



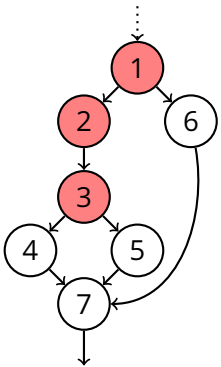
- Start Intel PT
- interp(1)

PT-enabled Trace Collection



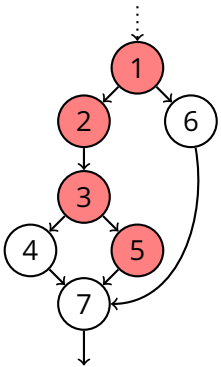
- Start Intel PT
- interp(1)
- interp(2)

PT-enabled Trace Collection



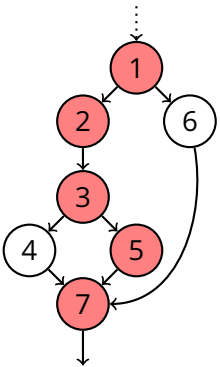
- Start Intel PT
- interp(1)
- interp(2)
- interp(3)

PT-enabled Trace Collection



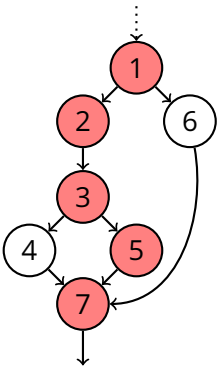
- Start Intel PT
- interp(1)
- interp(2)
- interp(3)
- interp(5)

PT-enabled Trace Collection



- Start Intel PT
- interp(1)
- interp(2)
- interp(3)
- interp(5)
- interp(7)

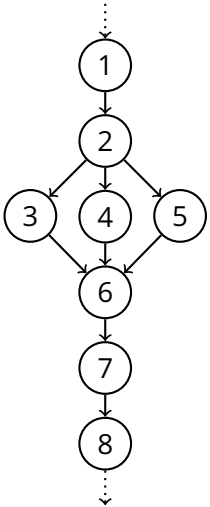
PT-enabled Trace Collection



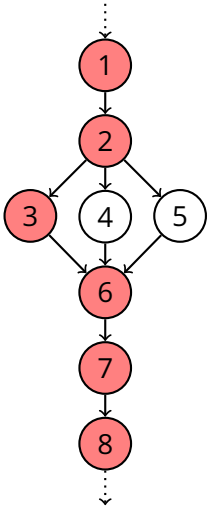
- Start Intel PT
- interp(1)
- interp(2)
- interp(3)
- interp(5)
- interp(7)
- Stop Intel PT

If tracing can be cheap, new opperunities arise

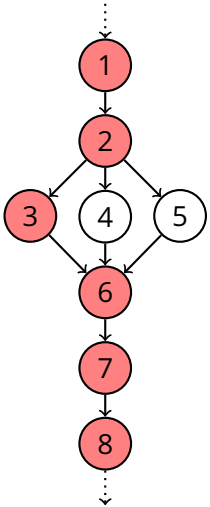
Avoiding trace tree explosions with trace sampling

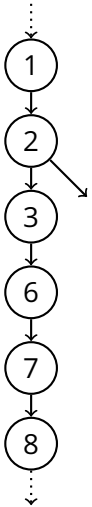


Avoiding trace tree explosions with trace sampling

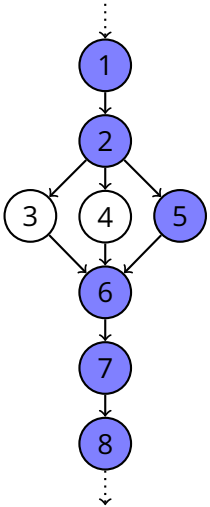


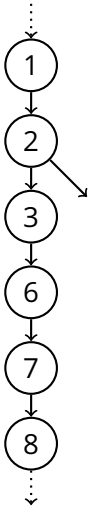
Avoiding trace tree explosions with trace sampling



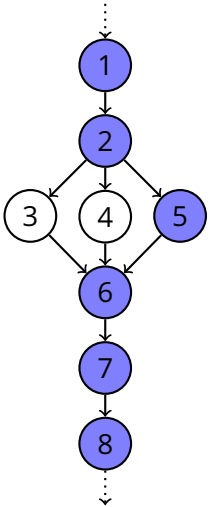


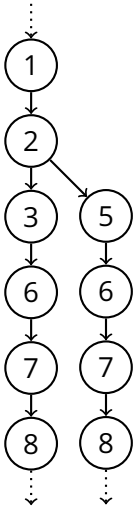
Avoiding trace tree explosions with trace sampling



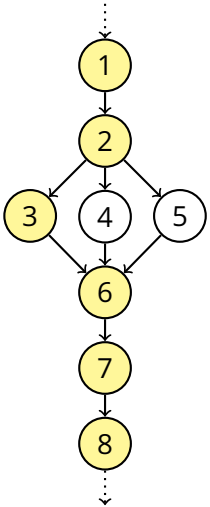


Avoiding trace tree explosions with trace sampling

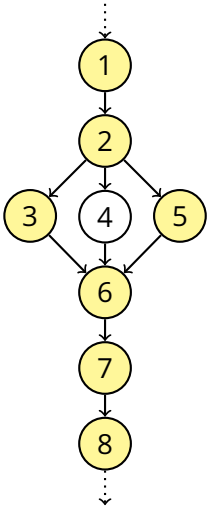




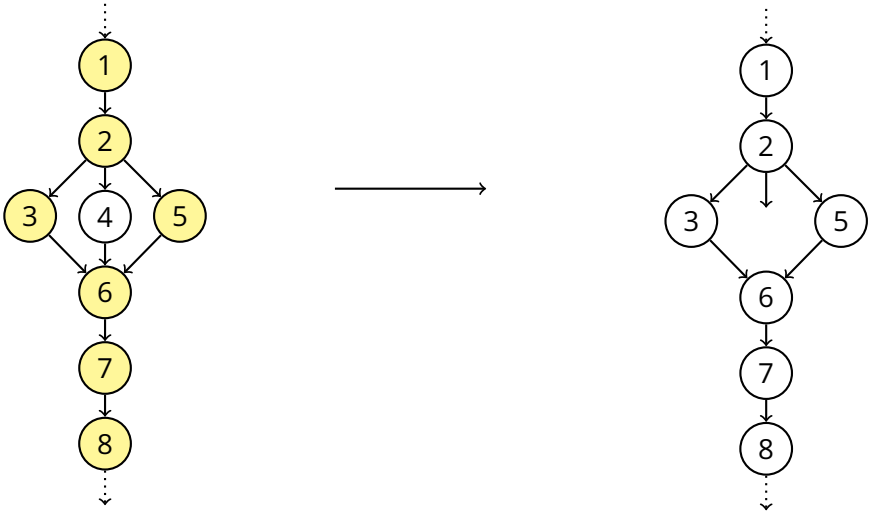
Avoiding trace tree explosions with trace sampling



Avoiding trace tree explosions with trace sampling



Avoiding trace tree explosions with trace sampling



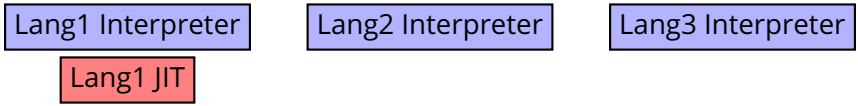
Language Implementation

Language Implementation

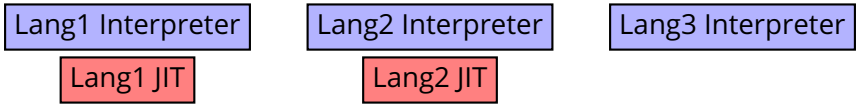
Language Implementation

Language Implementation

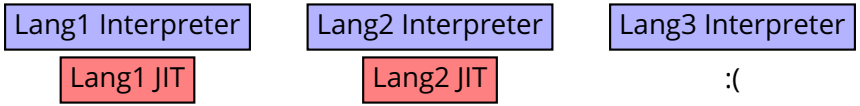
Language Implementation



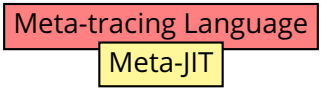
Language Implementation



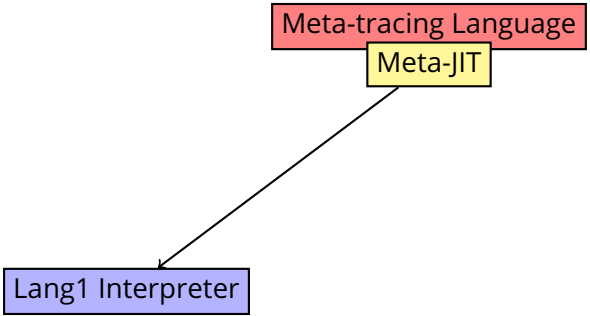
Language Implementation



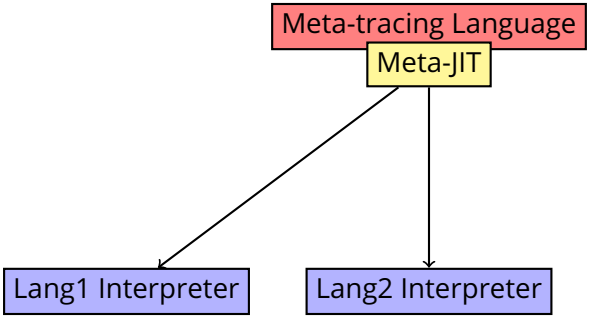
Language Implementation with Meta-Tracing



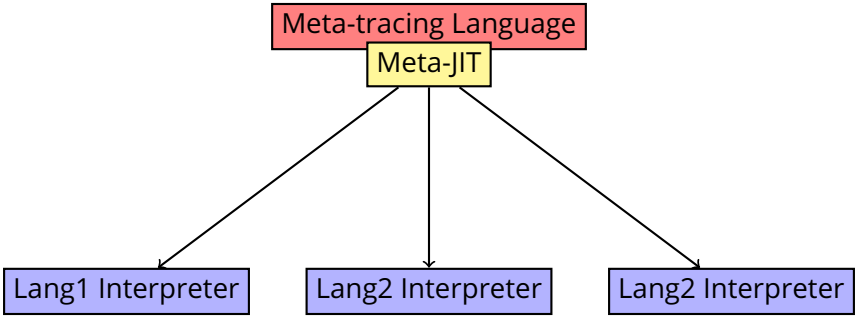
Language Implementation with Meta-Tracing



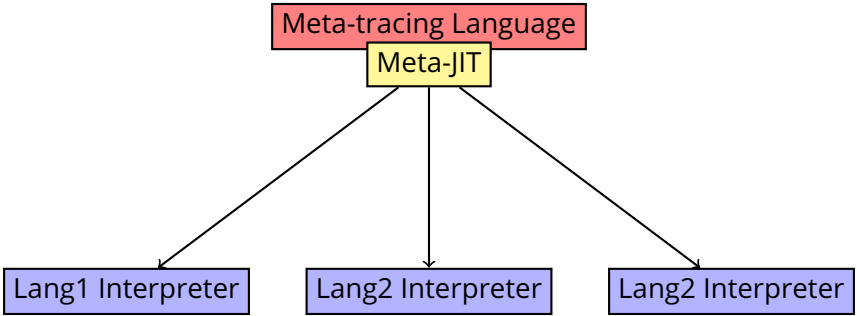
Language Implementation with Meta-Tracing



Language Implementation with Meta-Tracing



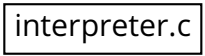
Language Implementation with Meta-Tracing



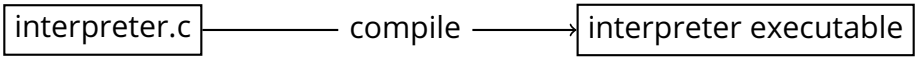
Free JITs!

yk

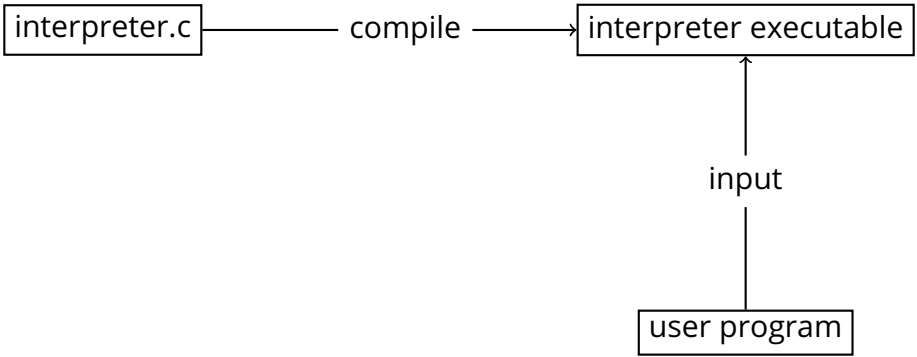
yk: high-level view



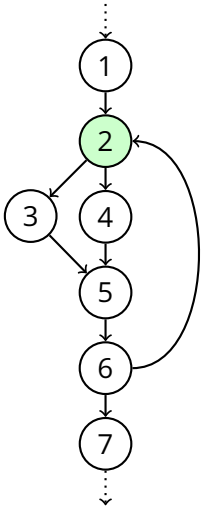
yk: high-level view



yk: high-level view

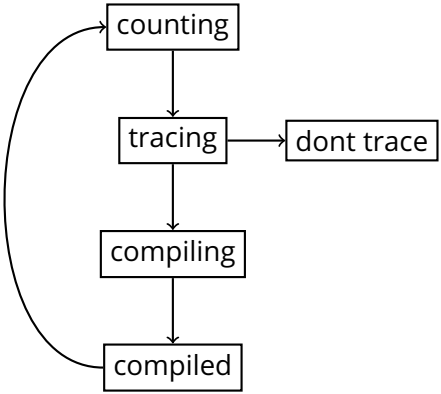


Feeding-back to the JIT

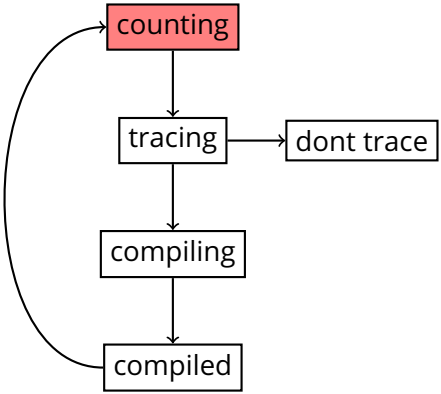



```
YkLocation *locs = calloc(
    len(prog), sizeof(YkLocation *));
locs[2] = yk_location_new();

...

yk_control_point(locs[pc])
```

Location: state machine



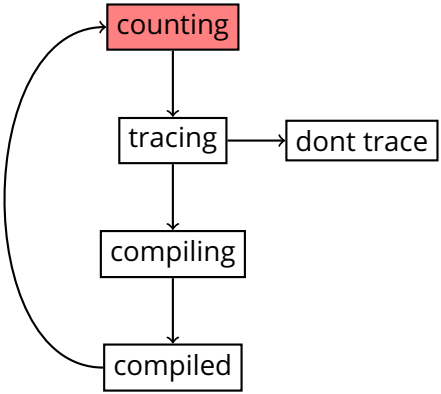
Location: state machine



Counting

- Atomic increment of hot count.

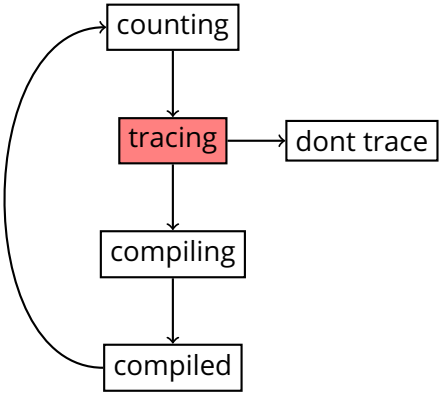
Location: state machine



Counting

- Atomic increment of hot count.
- Transition once hot threshold met.

Location: state machine

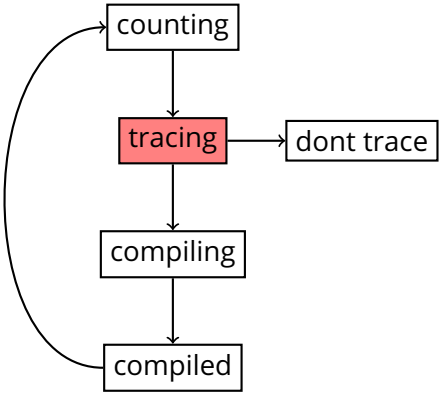


Tracing

As the location enters the tracing state:

- Turn on PT.

Location: state machine

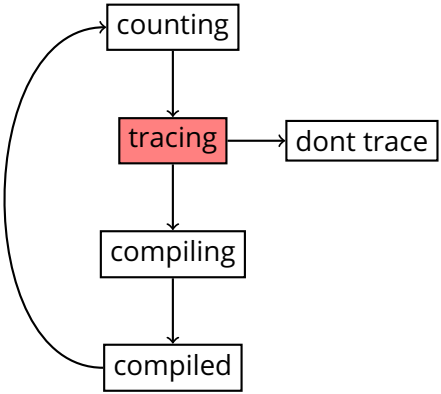


Tracing

As the location enters the tracing state:

- Turn on PT.
- Carry on interpreting.

Location: state machine



Tracing

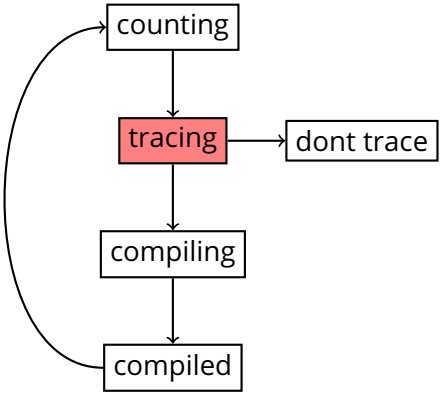
As the location enters the tracing state:

- Turn on PT.
- Carry on interpreting.

When we see this location again:

- Stop PT.

Location: state machine



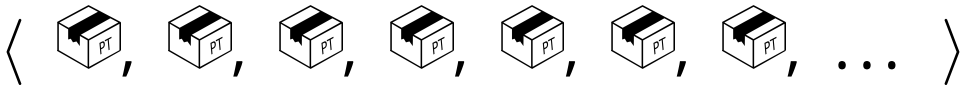
Tracing

As the location enters the tracing state:

- Turn on PT.
- Carry on interpreting.

When we see this location again:

- Stop PT.
- PT payload is decoded.
- Binary trace is “mapped”.



IPT block decoder

`⟨ 0x7ffff7fd6090, 0x7ffff7fd609c, 0x7ffff7fd60ba, ... ⟩`

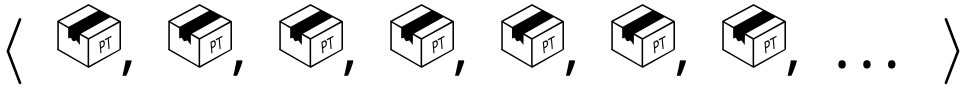
```
clang -fuse-ld=lld \
    -flto \
    -Wl,--plugin-opt=-lto-embed-bitcode=optimized \
    -Wl,--lto-basic-block-sections=labels \
    ...
    interpreter.c
```

SHT_LLVM_BB_ADDR_MAP Section (basic block address map)

This section stores the binary address of basic blocks along with other related metadata. This information can be used to map binary profiles (like perf profiles) directly to machine basic blocks. This section is emitted with `-basic-block-sections=labels` and will contain a BB address map table for every function which may be constructed as follows:

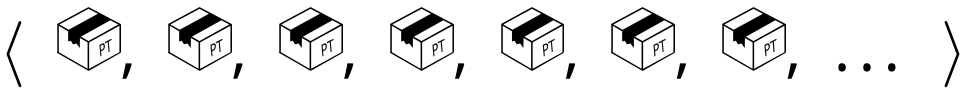
```
.section  ".llvm_bb_addr_map","",@llvm_bb_addr_map
.quad     .Lfunc_begin0                # address of the function
.byte     2                            # number of basic blocks
# BB record for BB_0
 .uleb128  .Lfunc_beign0-.Lfunc_begin0  # BB_0 offset relative to function entry (always zero)
 .uleb128  .LBB_END0_0-.Lfunc_begin0    # BB_0 size
 .byte     x                            # BB_0 metadata
# BB record for BB_1
 .uleb128  .LBB0_1-.Lfunc_begin0        # BB_1 offset relative to function entry
 .uleb128  .LBB_END0_1-.Lfunc_begin0    # BB_1 size
 .byte     y                            # BB_1 metadata
```

This creates a BB address map table for a function with two basic blocks.



IPT block decoder

⟨ 0x7ffff7fd6090, 0x7ffff7fd609c, 0x7ffff7fd60ba, ... ⟩



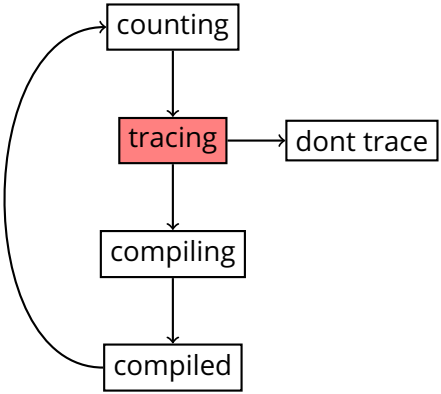
↓ IPT block decoder

⟨ 0x7ffff7fd6090, 0x7ffff7fd609c, 0x7ffff7fd60ba, ... ⟩

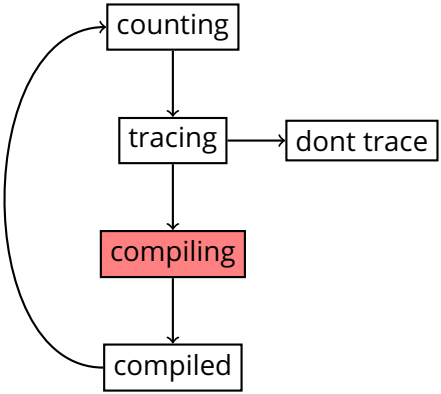
↓ map to LLVM block IDs

⟨ main<bb6>, do_stuff<bb0>, do_stuff<bb1>, ... ⟩

Location: state machine



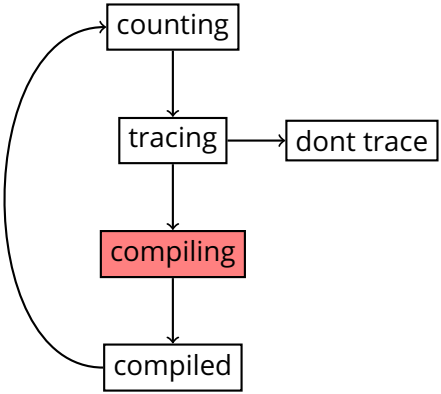
Location: state machine



Compiling

- Extract blocks from IR.

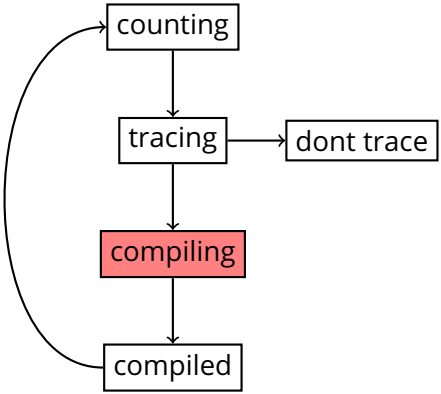
Location: state machine



Compiling

- Extract blocks from IR.
- Copy them into a new LLVM module.

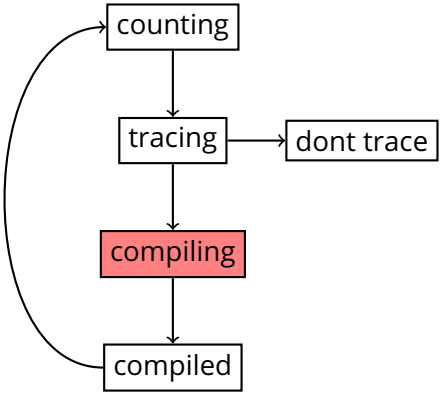
Location: state machine



Compiling

- Extract blocks from IR.
- Copy them into a new LLVM module.
- Rewrite terminators to guards.

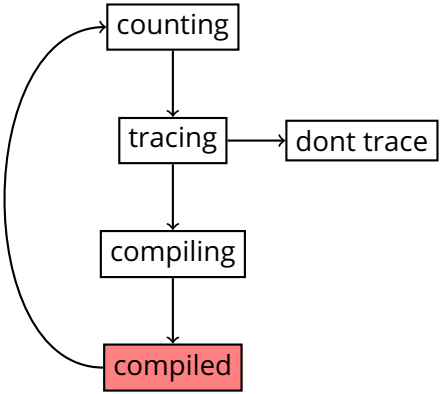
Location: state machine



Compiling

- Extract blocks from IR.
- Copy them into a new LLVM module.
- Rewrite terminators to guards.
- Compile to executable code.

Location: state machine



Compiled

- Jump to compiled code instead of interpreting.

- We can use this for interpreters written in any LLVM language.
  - Rust, C, C++, Julia, D, ...
- You get a tracing JIT for free!
- We can adapt existing interpreters.
  - CPython, Lua, Ruby MRI, ...
- A lot of the trace compilation work can be done by LLVM.
  - At least initially...

No performance numbers yet!

Plan of action

- Get some C interpreters JITting on our system.
  - Lua and CPython.

- Implement and evaluate additional ideas to improve warmup.

Thanks for Listening