The University of Manchester

Performance Understanding Tools for GraalVM using the *extended Berkley Packet Filter* (eBPF)

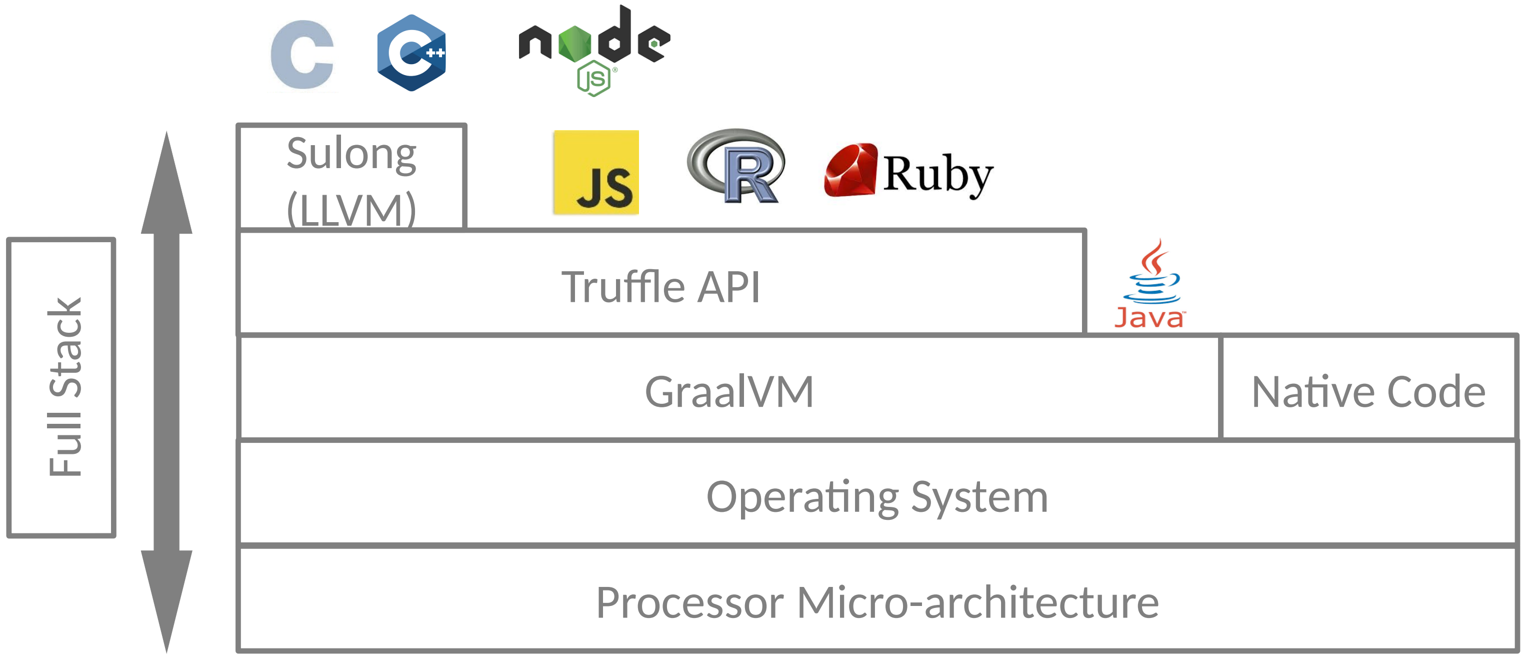
CGO GraalVM Workshop,
27th February 2021

Andy Nisbet, Salim Salim, Swapnil Gaikwad, Mikel Luján
Andy.Nisbet@manchester.ac.uk

https://github.com/beehive-lab

Beehive lab

Objective – Understanding the full stack for fair comparisons



- Where is time spent?
- How well is the underlying micro-architecture being used (performance counters)?
  - What are the reasons (bottlenecks) for poor utilization?
  - Investigate dynamic execution behaviour/program phases?

The University of Manchester

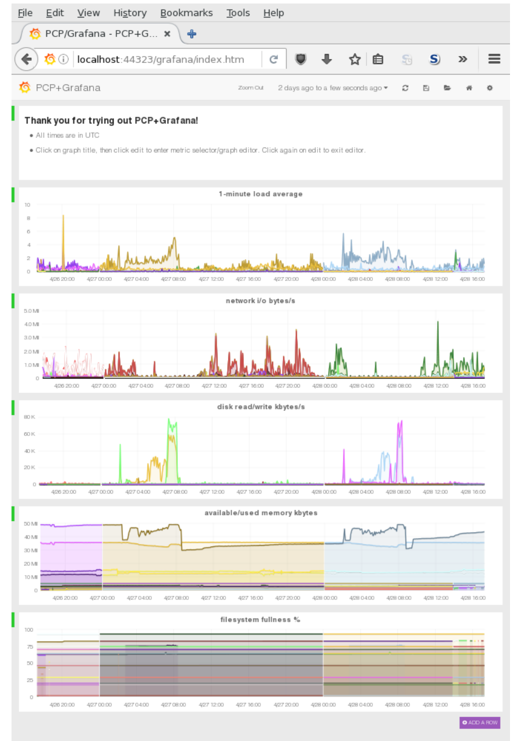
Process of Performance Analysis

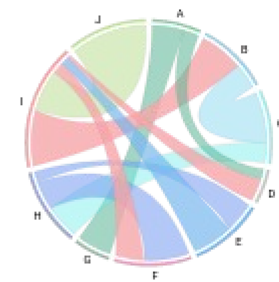
Data sources
“things we can measure, sample or instrument”

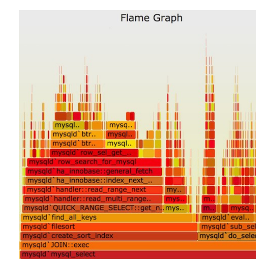
Observations of “data sources” are generated from application execution

Analyse/process logged observations to generate metrics and/or visualizations to aid the detection of issues

Many different online/offline visualizations & analysis tools







The University of Manchester

Outline

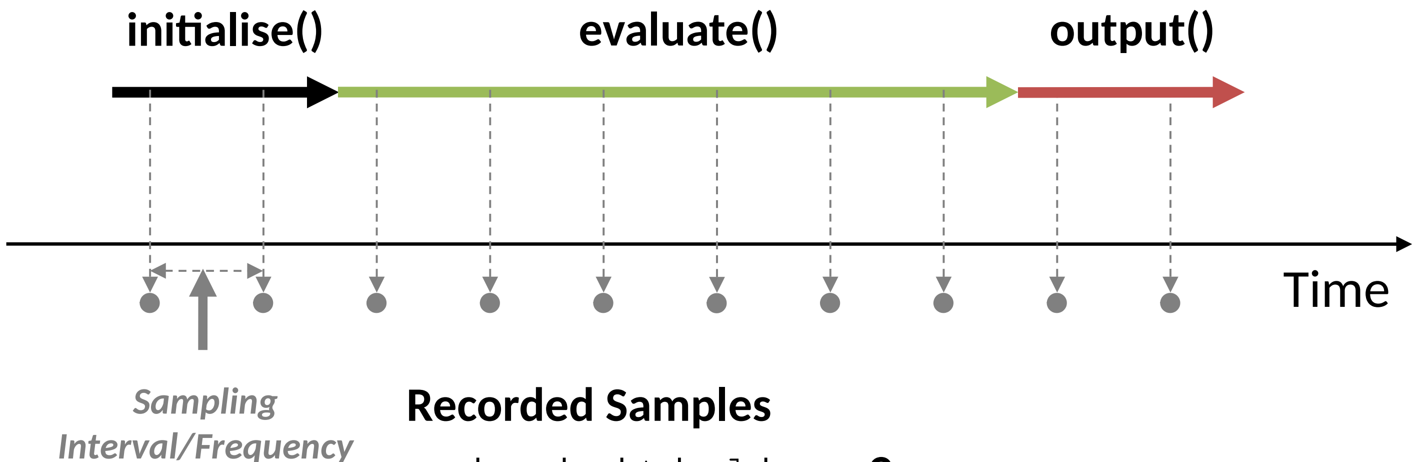
- Flamegraph Profile Visualizations – where is time spent?
- Sampling Profiler Shortcomings (JVM versus OS-perf)
- Truffle-based Language Performance (visualizing guest methods)
- Fullstack Tracing Instrumentation via (OS-eBPF)
  - Deoptimization case study
- Full-stack (micro-architecture) Performance analysis
  - Novel bcc-java tool for (full-stack) analysis
- Conclusions /discussion / acknowledgments

Simplified Flamegraph Example

```
void evaluate()   {  /* something expensive */ }
void initialise() {  /* initialise data */ }
void compute()    {  evaluate(); }
void output()     {  /* output results */ }

int main()
{
    initialise();   // 20% of the time
    compute();   // 60% of the time
    output();    // 20% of the time
    return 0;
}
```

Example: CPU Sampling Profiling



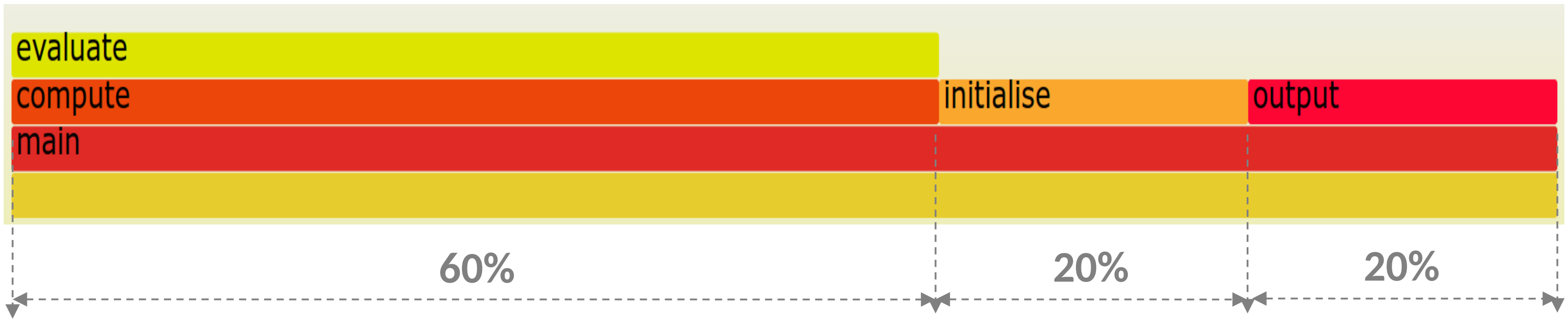
Recorded Samples

`main;initialise` **2**

`main;compute;evaluate` **6**

`main;output` **2**

Example: Flamegraph



Recorded Samples

`main;initialise` **2**

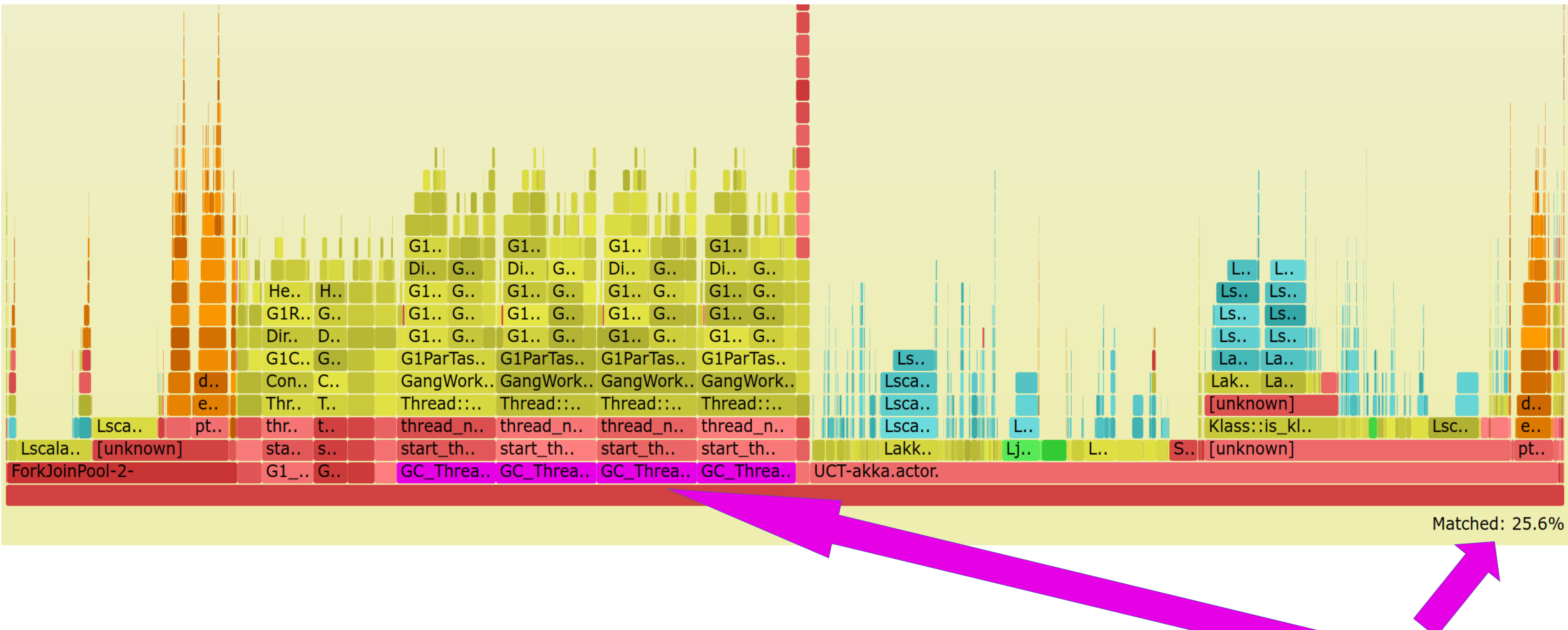
`main;compute;evaluate` **6**

`main;output` **2**

MANCHESTER 1824

The University of Manchester

CPU Profiling Flamegraph (perf)



Java (JIT-ed) | Inlined Java | C++ | Kernel Mode | Other/Rest | Searched

999Hz, 200s perf generated flamegraph on a 4 core laptop akka-uct from the Rennaisance benchmarks suite GraalVMCE21.1.0-dev

The University of Manchester

Main Findings on Profiling

- Production sampling rates f: 99Hz ≤ f ≤ 999Hz
- CPU Flamegraphs are just one visualization:
  - **Intermittent performance issues can be hidden in narrow columns**
- ICPE19 Nisbet et al, https://doi.org/10.1145/3297663.3309677

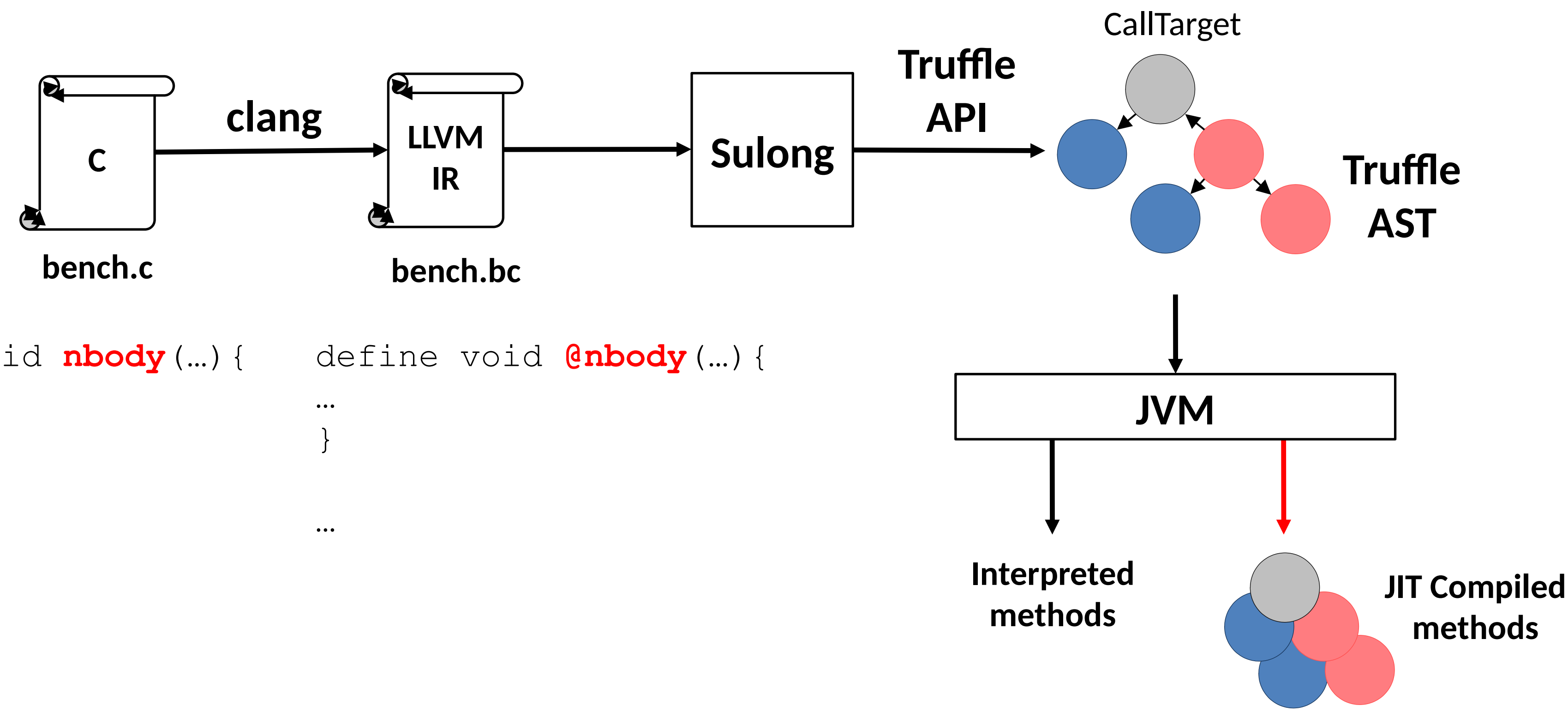
OS stack capture (perf)	JVM stack capture
Interpreter methods appear only as (**Interpreter**)	**Incomplete code coverage** of intrinsics/stubs & no view of OS
Full code coverage, but **need to dump JIT-ted code addresses**	**May suffer from safepoint bias Identify wrong hot-methods**
Hybrid profilers (**higher overheads**), no sampling bias, both Interpreted methods and OS are seen, but **some incomplete code coverage issues remain**	

The University of Manchester

Outline

- Flamegraph Profile Visualizations – where is time spent?
- Sampling Profiler Shortcomings (JVM versus OS-perf)
- **Truffle-based Language Performance (visualizing guest methods)**
- Tracing Instrumentation via (OS-eBPF)
  - Deoptimization case study
- Full-stack (micro-architecture) Performance analysis
  - Novel tool for comparing/evaluating performance (full-stack)
- Conclusions & Future Work

Sulong GraalVM based Execution

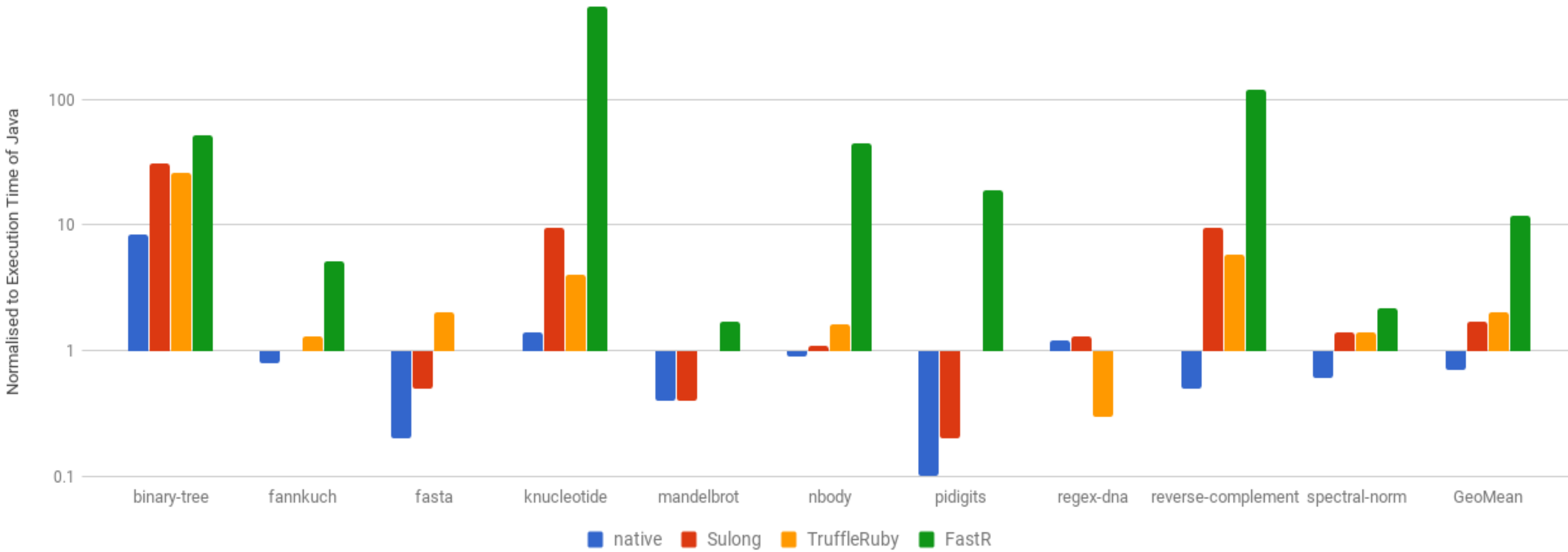



```
void nbody(…){
…
}
…
```

```
define void @nbody(…){
…
}


…
```

Performance Comparison for Truffle-based languages



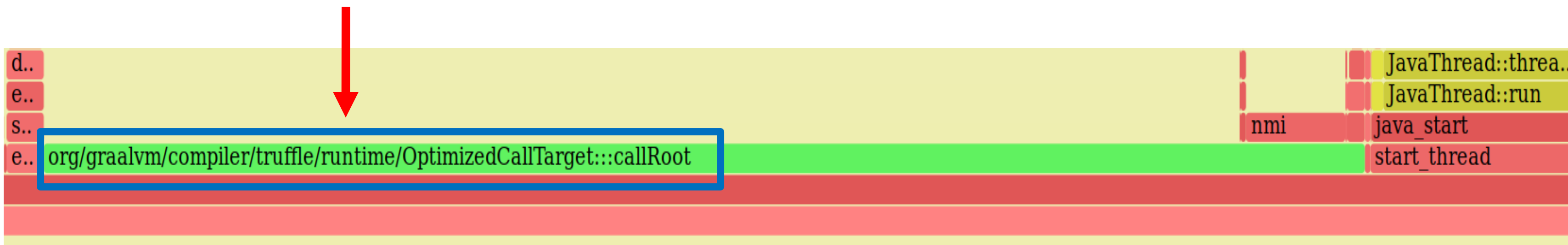
Shootout: Computer Language Benchmarks Game

Why is Truffle Language A is faster than B on a benchmark?

ManLang18: Gaikwad, Nisbet, Luján: https://dl.acm.org/doi/10.1145/3237009.3237019

Problem: LLVM IR Function name is Invisible in flamegraph

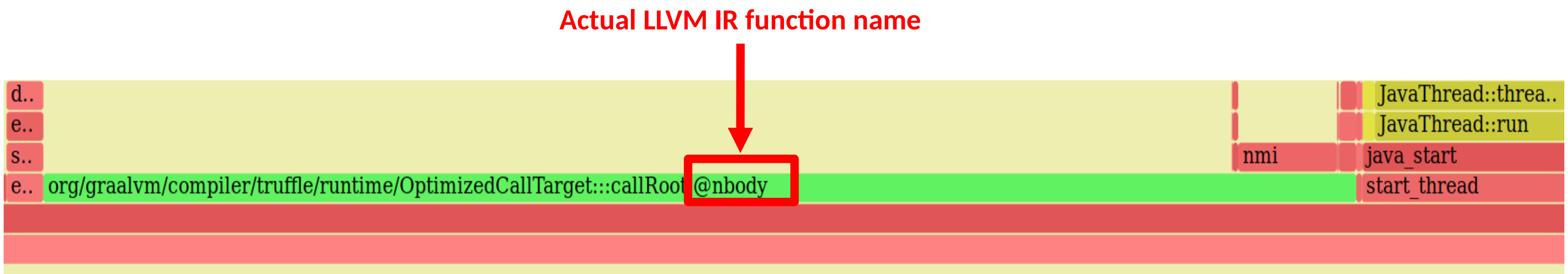
Hot Compiled Method from Truffle API (callRoot) – in general LLVM IR function name is invisible!



Width of the frame is proportional to the time spent in the associated function

- Profile of Sulong nbody (shootout benchmark suite)
- It has a single source method – **@nbody** not seen
- **callRoots represent a guest language compiled method**
- **Need a mechanism to relate callRoots to guest methods**

Truffle Profiling: Making Truffle guest language methods visible in flamegraphs

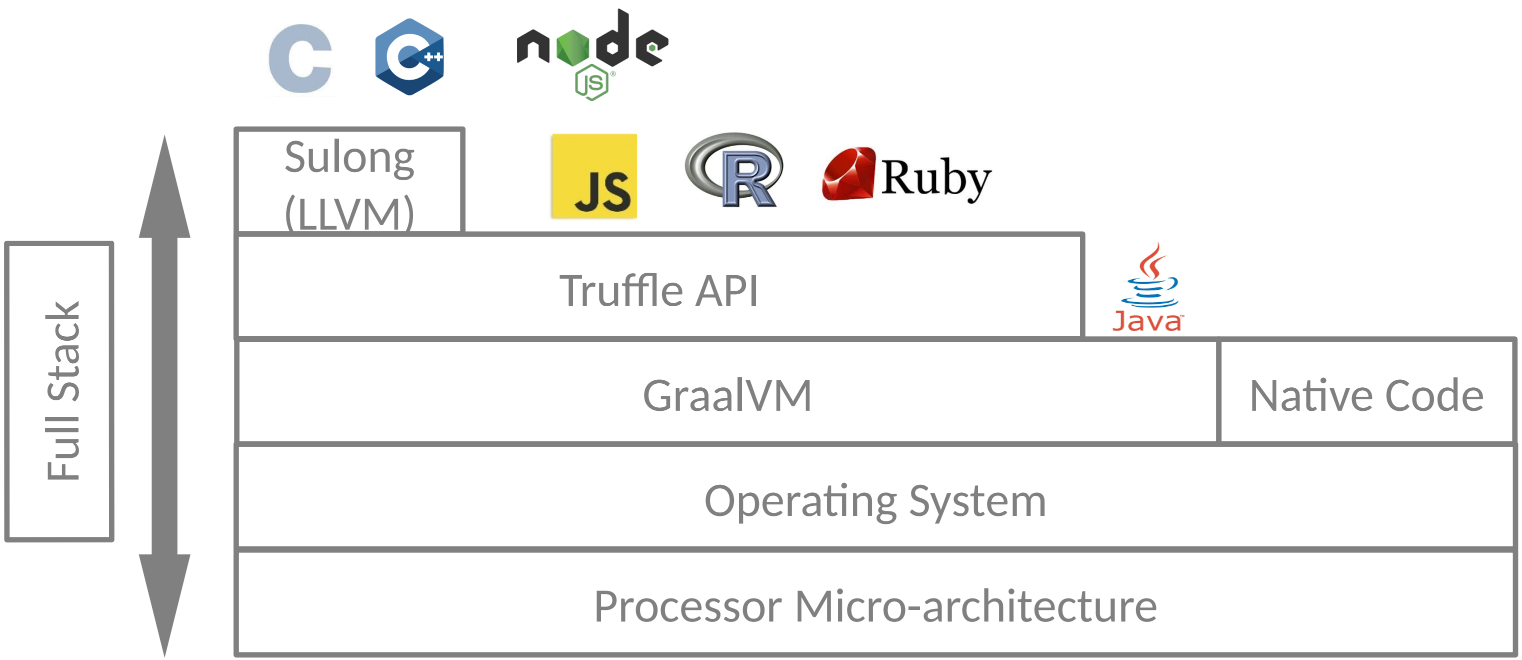


- Manlang18 modified Graal JIT – log information to resolve different callRoot code addresses to guest language source code
  - Flamegraph colors can be used to highlight different guest languages in a polyglot application

Outline

- Flamegraph Profile Visualizations – where is time spent?
- Sampling Profiler Shortcomings (JVM versus OS-perf)
- Truffle-based Language Performance (instrumenting guest methods)
- **Fullstack Tracing Instrumentation (OS-eBPF)**
  - Deoptimization case study
- Full-stack (micro-architecture) Performance analysis
  - Novel tool for comparing/evaluating performance (full-stack)
- Conclusions & Future Work

Towards Fullstack Tracing Instrumentation



Towards Fullstack Tracing Instrumentation

Fullstack tracepoints

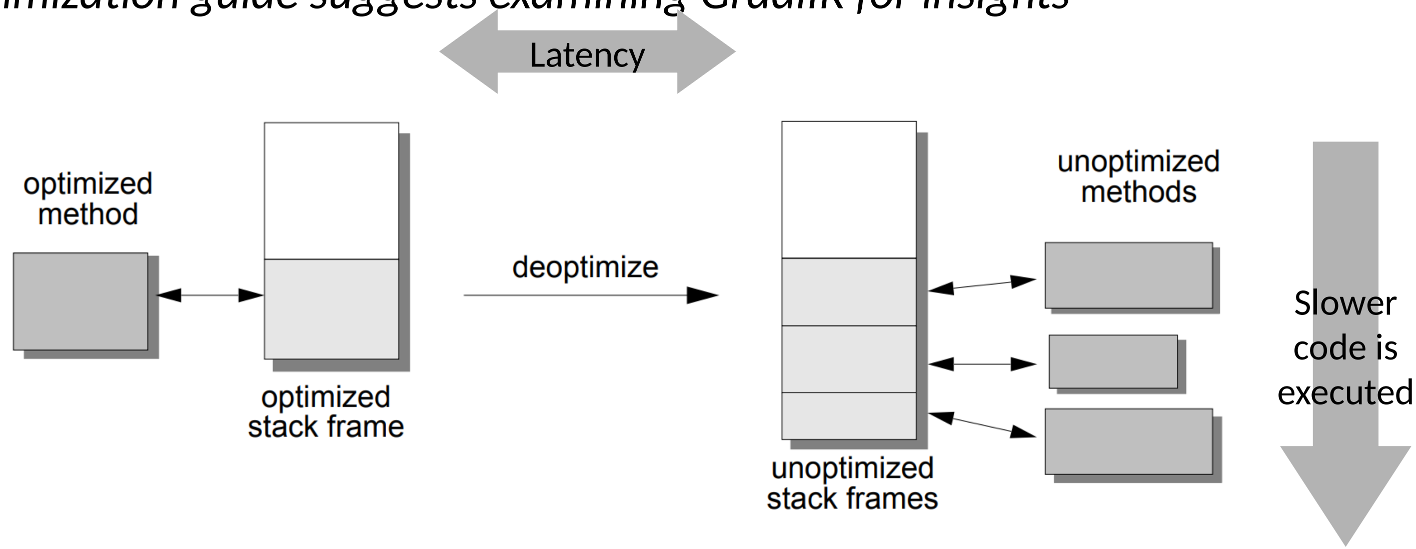
GraalVM libjvm.so 521 "hotspot"
Math library libm **9**
Standard C library **25**
pthread library **25**
Dynamic linker library **12**
Operating System **2653**
Performance counters measure behaviour on Micro-architecture

- eBPF insert/attach instrumentation to user and OS-kernel code tracepoints & probes
- **Measure – rather than sample**
- **Selectively capture/sample information at points of interest**
- Can also instrument any known address or text symbol (probe)

Deoptimization: Use-case for eBPF Tracing

- Speculative optimization leads to deoptimization if assumptions are violated
- *Which GraalVM methods do we need to trace? (instrument function entry/exit)*
- *Capture information using eBPF instrumentation*
  - *Selectively take a call-stack to find out what triggered deoptimization*
  - *Measure performance counters TLB/L3/cache-misses with instrumentation*
  - *Oracle optimization guide suggests examining GraalIR for insights*

Deoptimization must transform stack layout & leads to execution of less optimized code



MANCHESTER
1824
The University of Manchester

Count GraalVM deoptimizations - funccount

Count the executions of all Deoptimization related methods – print out every 5s
funccount libjvm.so:*Deopt* -i 5

High frequency

```
drandynisbet@drandynisbet-XPS-13-9360:~/CGO/flamegraphs/perf-map-agent/bin$ sudo /usr/share/bcc/tools/funcco
m.so:*Deopt* -i 10
Tracing 94 functions for "b'/home/drandynisbet/CGO/graalvm-6214be1be2-java11-21.1.0-dev-with-lli/lib/server/

FUNC                                    COUNT
b'_ZN14Deoptimization17last_frame_adjustEii'      1126

FUNC                                    COUNT
b'_ZN14Deoptimization17last_frame_adjustEii'       965

FUNC                                    COUNT
b'_ZN14Deoptimization19uncommon_trap_innerEP10JavaThreadi'        1
b'_ZN14Deoptimization13unpack_framesEP10JavaThreadi'        1
b'_ZN14Deoptimization24query_update_method_dataEP10MethodDataiNS_11DeoptReasonEbbP6MethodRjRbS6_'        1
b'_ZN14Deoptimization25unwind_callee_save_valuesEP5frameP11vframeArray'        1
b'_ZN14Deoptimization13uncommon_trapEP10JavaThreadii'        1
b'_ZN14Deoptimization24fetch_unroll_info_helperEP10JavaThreadi'        1
b'_ZN14Deoptimization25revoke_biases_of_monitorsEP10JavaThread5frameP11RegisterMap'        1
b'_ZN14Deoptimization17last_frame_adjustEii'       957

FUNC                                    COUNT
b'_ZN14Deoptimization19uncommon_trap_innerEP10JavaThreadi'        1
b'_ZN14Deoptimization13unpack_framesEP10JavaThreadi'        1
b'_ZN14Deoptimization24query_update_method_dataEP10MethodDataiNS_11DeoptReasonEbbP6MethodRjRbS6_'        1
b'_ZN14Deoptimization25unwind_callee_save_valuesEP5frameP11vframeArray'        1
b'_ZN14Deoptimization13uncommon_trapEP10JavaThreadii'        1
b'_ZN14Deoptimization24fetch_unroll_info_helperEP10JavaThreadi'        1
b'_ZN14Deoptimization25revoke_biases_of_monitorsEP10JavaThread5frameP11RegisterM
b'_ZN14Deoptimization17last_frame_adjustEii'       874
```

Determining deoptimization latency in GraalVM (libjvm.so) funclatency

Collect histograms of latency for a specific Deoptimization related method
funclatency -t -U -u 5 libjvm.so:_ZN14Deoptimization17last_frame_adjustEii

```
Function = b'Deoptimization::last_frame_adjust(int, int)' [16865]
     nsecs               : count     distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 0        |                                        |
         4 -> 7          : 0        |                                        |
         8 -> 15         : 0        |                                        |
        16 -> 31         : 0        |                                        |
        32 -> 63         : 0        |                                        |
        64 -> 127        : 0        |                                        |
       128 -> 255        : 0        |                                        |
       256 -> 511        : 0        |                                        |
       512 -> 1023       : 0        |                                        |
      1024 -> 2047       : 978      |****************************************|
      2048 -> 4095       : 34       |*                                       |
      4096 -> 8191       : 31       |*                                       |
      8192 -> 16383      : 7        |                                        |
     16384 -> 32767      : 1        |                                        |
     32768 -> 65535      : 0        |                                        |
     65536 -> 131071     : 0        |                                        |
    131072 -> 262143     : 1        |                                        |
    262144 -> 524287     : 0        |                                        |
    524288 -> 1048575    : 0        |                                        |
   1048576 -> 2097151    : 0        |                                        |
   2097152 -> 4194303    : 0        |                                        |
   4194304 -> 8388607    : 0        |                                        |
   8388608 -> 16777215   : 1        |                                        |
```

Long latency

Call-stack context for long Deoptimizations in GraalVM (libjvm.so) funcslower

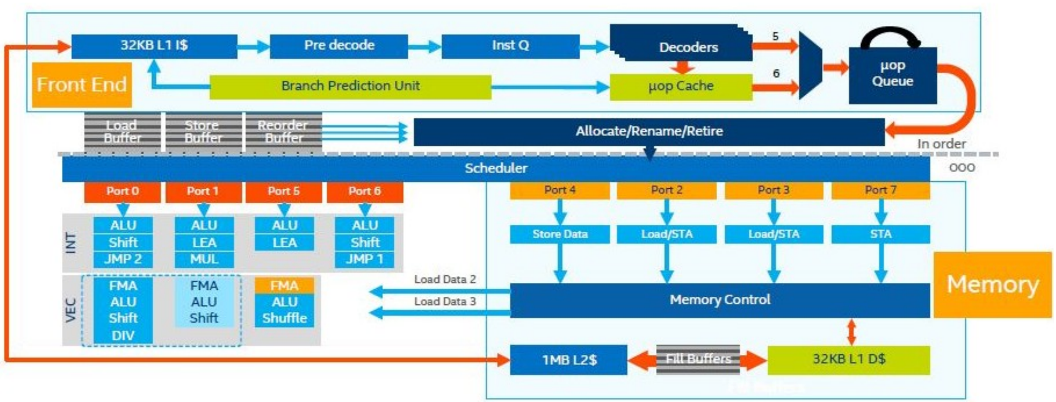
Timestamp collect user call-stacks greater than 5 micro-second latency
Deoptimization::last_frame_adjust(int, int)

funcslower -t -U -u 5 libjvm.so:_ZN14Deoptimization17last_frame_adjustEii

```
    b'start_thread'
48242.137819 C1 CompilerThr 700902    5.99               3 /home/drandynisbet/CGO/graalvm-6214be1be2-java11-2
adjustEii
    b'CodeEmitInfo::interpreter_frame_size() const'
    b'LinearScan::compute_oop_map(IntervalWalker*, LIR_OpVisitState const&, LIR_Op*)'
    b'LinearScan::assign_reg_num(GrowableArray<LIR_Op*>*, IntervalWalker*)'
    b'LinearScan::assign_reg_num()'
    b'LinearScan::do_linear_scan()'
    b'Compilation::emit_lir()'
    b'Compilation::compile_java_method()'
    b'Compilation::compile_method()'
    b'Compilation::Compilation(AbstractCompiler*, ciEnv*, ciMethod*, int, BufferBlob*, DirectiveSet*)'
    b'Compiler::compile_method(ciEnv*, ciMethod*, int, DirectiveSet*)'
    b'CompileBroker::invoke_compiler_on_method(CompileTask*)'
    b'CompileBroker::compiler_thread_loop()'
    b'JavaThread::thread_main_inner()'
    b'Thread::call_run()'
    b'thread_native_entry(Thread*)'
    b'start_thread'
48243.936765 C1 CompilerThr 700902    6.56               1 /home/drandynisbet/CGO/graalvm-6214be1be2-java11-2
adjustEii
```

Understanding Full Stack Execution Behaviour with Top-down Analysis

- Performance counter metrics – give reasons for code execution efficiency (IPC)
- Structured methodology is needed to understand out-of-order execution in modern Intel/ARM processors
- Many instructions are typically in-flight awaiting resources/results to become available
- Inefficiences at front end, back end, and due to incorrect speculations



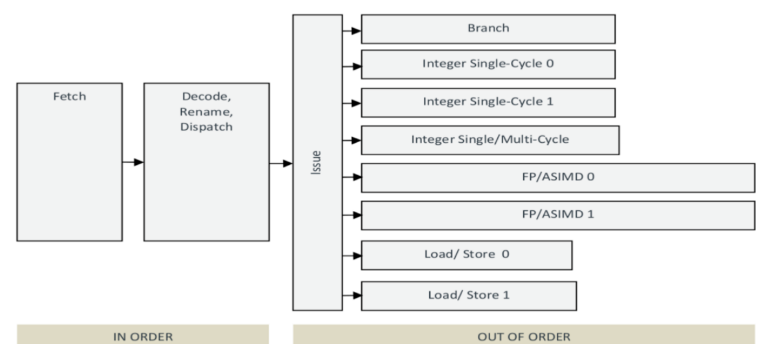
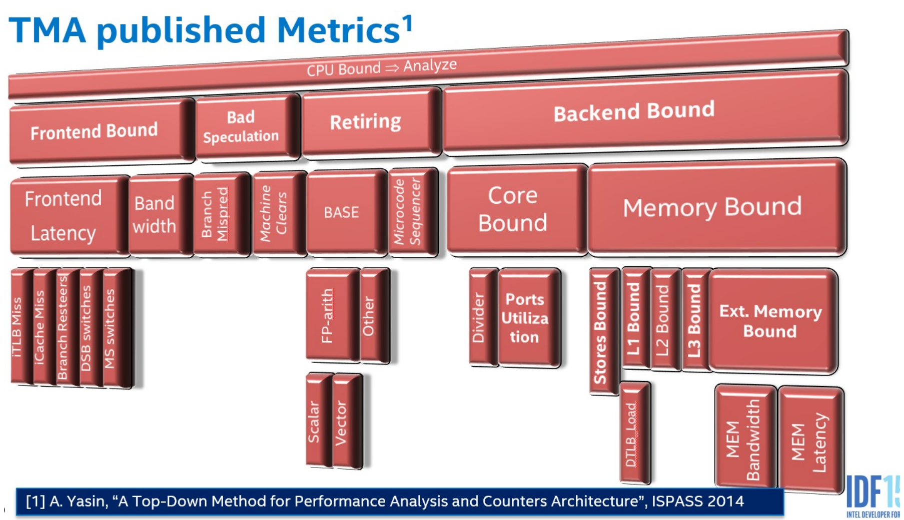


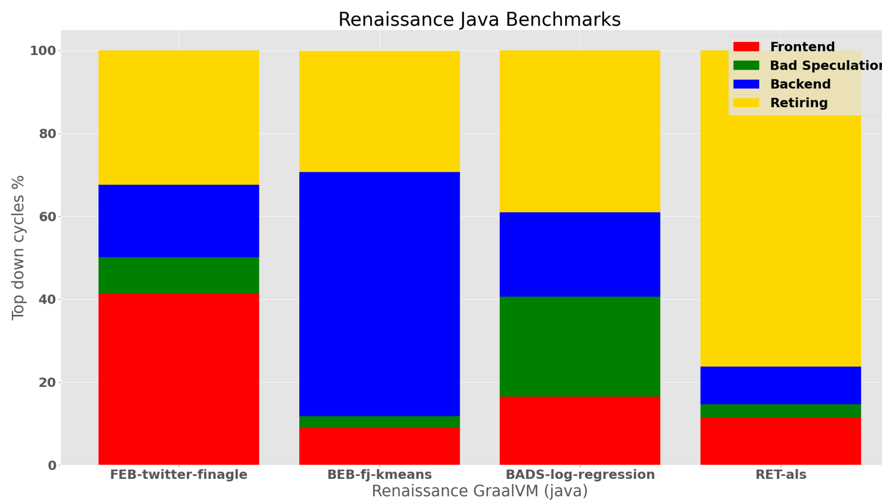
Image snipped from wikipedia

Understanding Full Stack Execution Behaviour with Top-down Analysis

- Top-down – structured way to analyze performance
- Use different sets of performance counters to identify issues
- Metrics – classify the percentage of cycles limited by a microarchitectural issue
- Maximise useful work by increasing the Retiring percentage

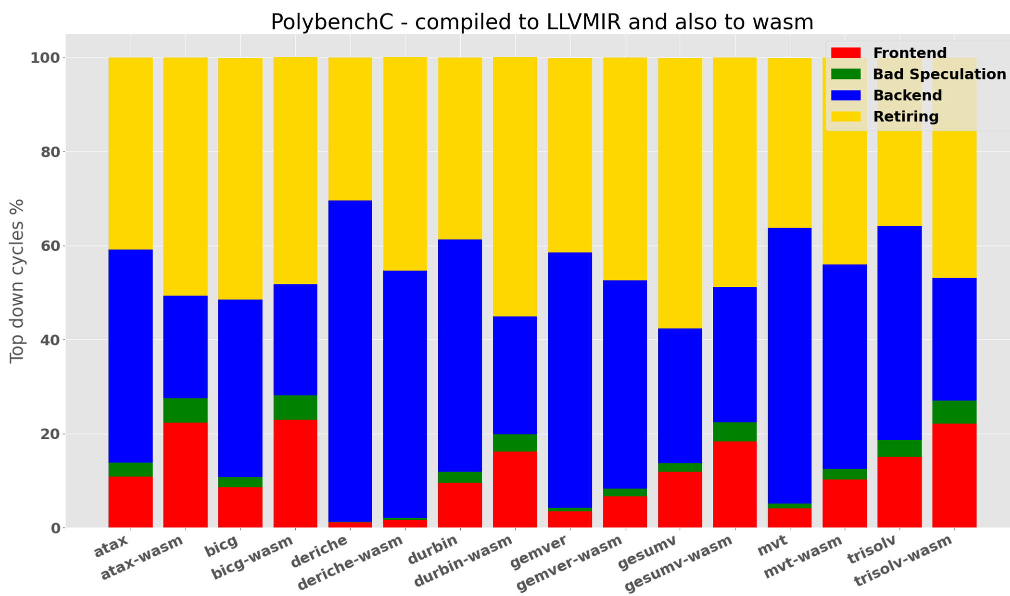


[1] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture", ISPASS 2014



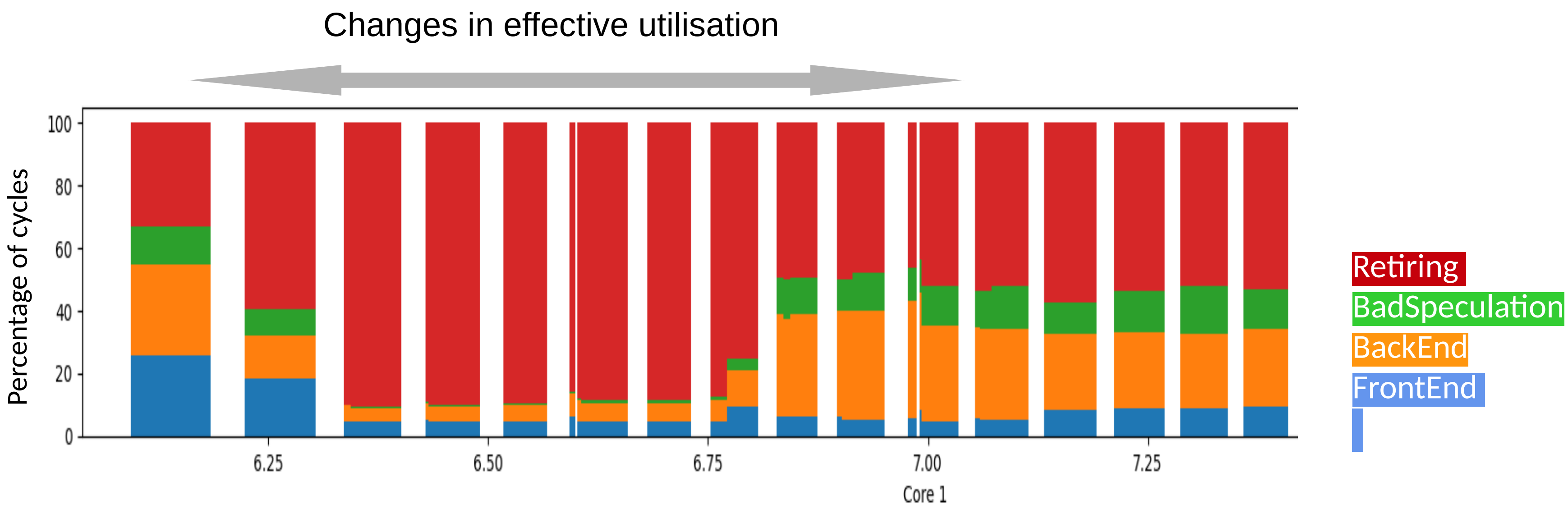
Understanding Full Stack Execution Behaviour with Top-down Analysis

- C benchmarks compiled to LLVMIR and also to WebAssembly
- Different top-down behaviour exhibited by the same benchmark executed using different Truffle languages
- Aggregated information only hints at different behaviour



Fullstack concept with bcc-java

- Top-down tracing of every thread execution time-slice on a CPU
- Less than 5% overhead
- Dynamic per-thread top-down execution behaviour is exposed
- Retiring Indicates how well the microarchitecture is utilised

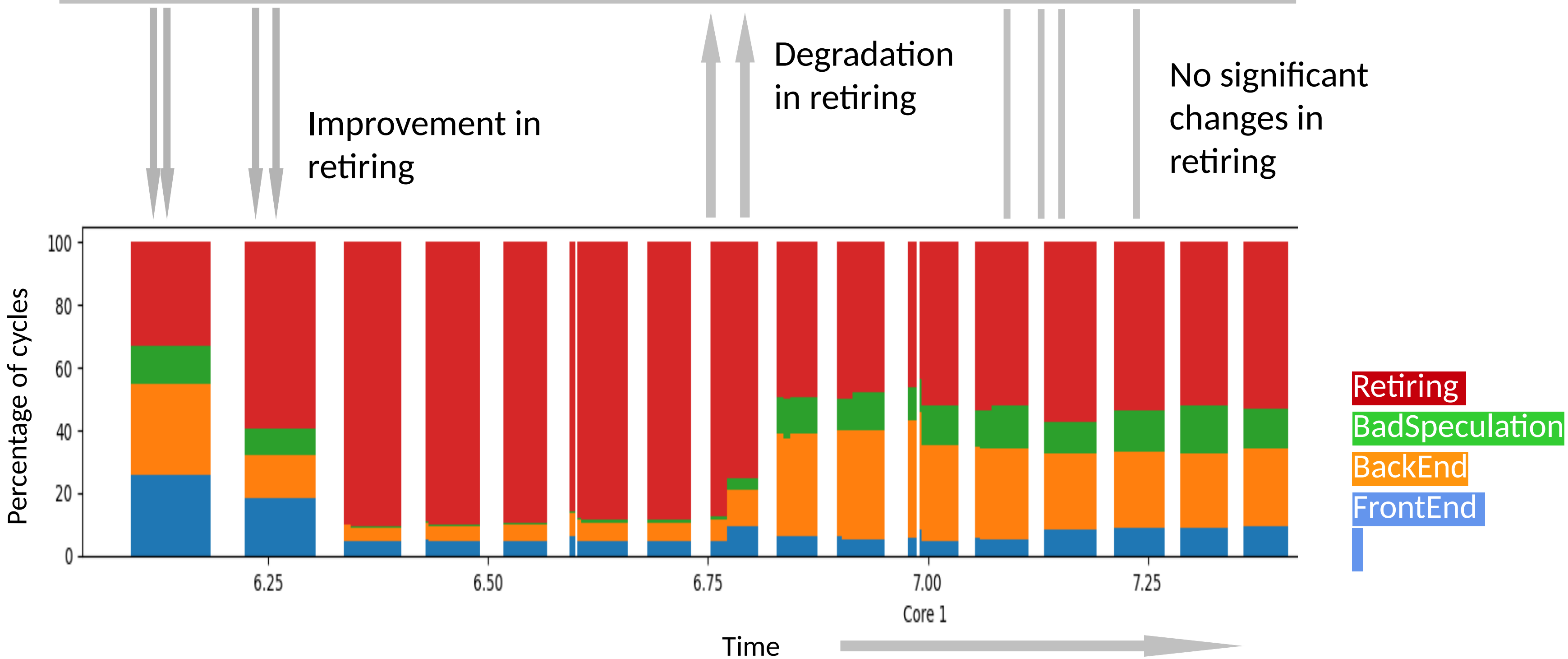


MANCHESTER 1824
The University of Manchester

Fullstack concept with bcc-java

- Insights/correlations can be drawn concerning changes to top-down behaviour

Instrumented events: timestamp, event, core, optional call-stack



MANCHESTER 1824
The University of Manchester

Conclusions

- Better tooling is needed to make it easier to instrument GraalVM/JVMs using perf/eBPF
  - For example improved support to identify/instrument JIT-ted code addresses
- Even standard eBPF tools can extract useful information - instrumenting libjvm.so
  - funccount/funclatency/funcslower
- Flamegraphs can visualise where time is spent, at reasonably low overhead
  - Needle in a haystack: performance issues can be obscured!
- Fullstack tracing, performance counters, and selective call-stack capture can act like a magnifying glass for performance analysis
- Novel aspects of our tool – dynamic thread level behaviour is overlayed with event traces

The University of Manchester

Discussion Questions

- Does the community have any important performance problems/use-cases they can share?
  - Information on what GraalVM code/events to trace for a given use-case such as Deoptimization?
  - How to implement tooling to selectively dump the GraalIR for a compilation unit?
  - How to identify performance impact of deoptimizations? Can we identify the impact of executing less optimized code?
- Recommendations for performance optimization/GraalVM internals tutorial examples/information sources?

Acknowledgements

EPSRC
Engineering and Physical Sciences Research Council

European Commission

ROYAL ACADEMY OF ENGINEERING

THE ROYAL SOCIETY

arm

ORACLE