UNDERSTANDING THE PERFORMANCE OF MANAGED RUNTIME ENVIRONMENTS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2022

Timothy Hartley

Department of Computer Science

Contents

List of Tables

List of Figures

Abstract

Managed runtime environments (MRE) have become commonplace across the spectrum of computing devices and are nowadays found on portable devices such as mobile phones, personal computers and data-centre servers. Optimising MRE execution requires insight into the performance of the MRE components themselves and their interactions with the workloads they host. This thesis, in two parts, is concerned with aspects of performance engineering and understanding in the context of managed runtime environments.

The first part, covers an investigation arising from porting MaxineVM, a research virtual machine to the ARMv8 architecture. During this work aspects of the original design were encountered, that cut across constraints imposed by the ARMv8 architecture affecting elements of the just-in-time compilation system, specifically the construction and subsequent treatment of call-sites at runtime, that must be patched to redirect control. We investigate functionally equivalent implementations of call-sites, evaluating the performance and tradeoffs using a microbenchmark, three JVM benchmark suites and statistical profiles derived from microarchitecture performance counters, on three diverse ARMv8 platforms. The experiments show the variation in performance between the alternate strategies of up to 12%, and also variation across the different implementations of the architecture. We find the potential opportunity to explore optimisation relevant to all instruction set architectures with limited direct call ranges using code cache management to encourage local direct branches.

The second part of the thesis presents two fine-grained studies into managed runtime performance. Firstly, the Top-Down Microarchitecture Analysis methodology is extended to individual managed runtime threads, demonstrating dynamic microarchitectural utilisation behaviours of individual threads at OS-scheduling quantum granularity. These behaviours reveal which threads are effectively utilising the processors microarchitecture, and which are not, identifying opportunities for optimisation and motivation for further investigation. The second study explores and refines warm-up

and steady-state behaviour analysis of a MRE. Benchmarking experiments are a common technique used to gain insight into the performance of MREs, and methodologies typically rely on timing iterations in order to ascertain when peak performance has been achieved. This thesis proposes a new approach, including microarchitecture performance counters, specifically using counts of retired micro-operations as a measure of work done. It is argued that this approach offers a more reliable metric than elapsed time, that is less susceptible to interference from the OS, and microarchitectural effects.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I thank my supervisor Mikel Luján for all his help, encouragement and guidance. Thanks also to Christos Kotselidis my co-supervisor, Andy Nisbet and Foivos Zakkak. Thanks to my family for their love, support and welcome distractions. This research was generously funded by an EPSRC iCASE award with Arm Ltd.

Chapter 1

Introduction

In order to increase performance in the post Dennard scaling era, the microprocessor industry adopted multicore architectures, scaling the number and configuration of cores for targeted applications or to meet a specific price/performance target. Exploiting the potential of multicore parallel execution has required a shift in the approach to software design and implementation in order to efficiently map parallel tasks onto hardware. Several paradigms and parallel programming frameworks have been developed in order to help realise this objective.

The demand for software applications in the current era has ensured that many are nowadays implemented in managed languages that require a Managed Runtime Environment (MRE) to execute them. The safety features of managed languages coupled with their feature rich runtime libraries lower the inherent cost of software engineering, enabling higher developer productivity. MREs have subsequently become ubiquitous across the computing spectrum and can be found deployed in datacentres, on personal computers and mobile phones. Gaining insight into the performance of a workload running on a MRE is compounded by the nature of the runtime itself, where parallel execution includes not only the application payload, but also components of the MRE itself, such as just-in-time (JIT) compilers and garbage collection (GC). Extensible tools are needed to aid performance understanding and expose optimisation opportunities for both MREs and the applications they host.

This thesis investigates performance understanding for managed runtimes. Specifically, it investigates the performance impact of architectural constraints on just-in-time compilation issues, and it presents two studies of the performance of managed language application and runtime threads at operating system thread scheduling quantum

granularity. Our tools can identify the execution phases of an MRE executing an application, and can determine if application and MRE service threads are efficiently utilising the underlying processor architecture.

1.1 Thesis Outline

The remainder of the thesis is based on two independent but related pieces of work in which we investigate the performance of two MREs: MaxineVM and the OpenJDK JVM.

Chapter 2

The work described here arose from porting MaxineVM [WHVDV+13] to the ARMv8 architecture, and the need to support method calls that must reach beyond the limit imposed by the branch and link instruction `BL`. The complication of constructing such long range call-sites on ARMv8 platforms resides along three vectors:

- The ISA limits direct branch targets to within 128MB of the program counter.
- The coherency protocol does not include the instruction caches, delaying updates to instructions accessible to the CPU from being observable until a cache maintenance sequence of instructions has been completed.
- The architecture excludes all but 8 instructions from being safe to both execute and patch concurrently. Out of that set only the two branch instructions: `B` and `BL`, are particularly useful for constructing callsites.

At the time the port was done, there was no clear picture of an optimum canonical implementation of a concurrently and executable and patchable call-site for JIT compiled code for ARMv8, including the port of OpenJDK. The OpenJDK strategy was to trap into the interpreter and patch code sequentially with all other threads paused [HD14]. This approach was not available to MaxineVM being a compile only MRE, where patching must be done to compiled code that may be concurrently executed by another core. Such runtime updates to shared code without explicit synchronisation is difficult problem.

Contributions

This chapter conducts a thorough evaluation of alternate call-sites and associated runtime patching strategies that are resilient to concurrent execution and modification. The purpose of the investigation was to determine whether there was a performance margin between the different approaches that would favour a particular implementation. The performance analysis conducted was based on timed benchmark experiments, and analysis of performance monitoring unit (PMU) event counts collected using perf during the course of the executions. Some further program level profiling was done using a JVMTI agent and bytecode instrumentation. These measurements enabled us to not only differentiate between the relative performance of the different approaches but also understand the scope of the experiments and to elicit some behaviour of MaxineVM when executing the different workloads. An early version of the work described in chapter 3 was also trialled for collecting PMU events.

A prior version of this work has been published [HZKL19]; the version presented in chapter 2 has been submitted to the Transactions on Architecture and Code Optimization journal. All of the implementation, experiments, analysis and text were completed by the author. Co-authors provided useful suggestions, editing and alternative implementation ideas some of which were trialled but not preserved or presented in the paper. Foivos Zakkak produced figure 2.1.

Chapter 3

This work is concerned with investigating and improving the performance understanding of managed runtimes. The contributions comprise two studies into the performance of OpenJDK 11: firstly the Top-Down Microarchitecture Analysis [Yas14] methodology is extended, and secondly the current understanding of warm-up and steady-state behaviour of managed runtimes is investigated and improved.

Top-down microarchitecture analysis (TMA) [Yas14] is a well established methodology for identifying performance issues in modern processors supporting out-of-order execution, and so help identify optimisation opportunities. Current tools such as *perf* and toplev [Kle22] typically perform such analysis and measurement over timed intervals, at full system, process, and processor core granularity. In this study a finer-grained analysis is performed, in which individual threads are investigated to determine whether they are efficiently using a processors microarchitecture, using measurements taken at operating system thread scheduling quantum granularity.

In the second study, warm-up and steady-behaviour of managed runtimes is investigated. MRE vendors and developers frequently use benchmark experiments, two common use cases include: verifying the performance of new optimisations, or in regression tests to identify any unwanted side effects that may have been inadvertently introduced. Such assessments frequently rely on identifying a phase of peak, steady-state performance, when the performance of the MRE and workload has been optimised and stabilised, and has been the subject of a number of studies [KJ13, GBE07, BBTK+17, CR16]. Some have questioned the conventional wisdom that MREs exhibit a defined warm-up phase with subsequent steady-state execution [BBTK+17]; and drawn attention to the pitfalls of using elapsed wall-clock time as a reliable measure [CR16]. In this work, the utility of microarchitecture performance counters is investigated, to help identify program execution phases that indicate if warm-up has occurred and steady-state execution behaviour has been achieved.

Contributions — TMA Analysis

A detailed analysis of the performance of individual managed runtime threads using top-down metrics derived from performance counters at OS scheduling quantum granularity. Results are shown that demonstrate the different top-down behaviours of VM and application threads executing the Renaissance benchmark suite using OpenJDK. Dynamic behaviours are shown in both violin plots and traces visualised in the Chrome profiler, enabling identification of threads suffering from overheads related to a specific microarchitectural characteristic, and revealing features that would otherwise go unresolved using the conventional current techniques.

Contributions — Warm-up and Steady-state Analysis

This study investigates the applicability of using microarchitectural performance counter measurements to investigate warm-up and steady-state behaviour in MREs and workloads. Based on the experimental method of [BBTK+17], changepoint analysis is used across benchmark iterations comparing wall clock execution time and accumulated retired micro-operations. This work argues that such quantities are a more reliable measure of effective work done than elapsed time and less susceptible to measurement error in a typical system, where resource contention and microarchitectural and OS interaction can affect timed experiments.

Contributions — BPFxVM

To achieve these objectives BPFxVM is developed and described: a suite of low-overhead, flexible tools for investigating performance using the BPF Compiler Collection (BCC) toolkit [IOV22]. BCC provides a python based frontend to the *extended Berkeley Packet Filter* (eBPF) subsystem in the Linux kernel. BPFxVM leverages the features of eBPF along with sampling microarchitectural PMU event counters to produce execution profiles and traces. This approach is both MRE and platform agnostic and is demonstrated on both x86 and ARMv8 machines and can be used to investigate any application that executes on a Linux platform.

Chapter 2

Just In Time Compilation on Arm – A Closer look at Call Site Code Consistency

2.1 Abstract

The increase in computational capability of low-power Arm architectures has seen them diversify from their more traditional domain of portable battery powered devices into data center servers, personal computers, and even Supercomputers. Thus, managed languages (Java, Javascript, etc.) that require a managed runtime environment (MRE) need to be ported to the Arm architecture, requiring an understanding of different design trade-offs. This paper studies how the lack of strong hardware support for Self Modifying Code (SMC) in low-power architectures (e.g. absence of cache coherence between instruction cache and data caches), affects Just-In-Time (JIT) compilation and runtime behavior in MREs. Specifically, we focus on the implementation and treatment of call-sites, that must maintain code consistency in the face of concurrent execution and modification to redirect control (patching) by the MRE. The lack of coherence, is compounded with the maximum distance (reach of) a call-site can jump to as the reach is more constrained (smaller distance) in Arm when compared with Intel/AMD. We present four different robust implementations for call-sites and discuss their advantages and disadvantages in the absence of strong hardware support for SMC. Finally, we evaluate each approach using a microbenchmark, further evaluating the best three techniques using three JVM benchmark suites and the open source

MaxineVM showcasing performance differences up to 12%. Based on these observations, we propose extending code-cache partitioning strategies for JIT compiled code to encourage more efficient local branching for architectures with limited direct branch ranges.

2.2 Introduction

The onset of sustained increases in computational capability and market share of low power processors can be traced back to the early days of mobile computing, in particular smartphones. Subsequent innovation in architecture and process node have enabled low power designs to achieve ever higher performance and so enter new markets. Low power processors can now be found, amongst others, in Internet of Things (IoT) devices, autonomous vehicles, edge nodes of distributed systems, data center servers [ARM19c], and personal computers. This trajectory of processor capability has enabled a shift in the languages used to program them. In addition to traditional languages such as C and C++, many applications are nowadays developed in managed languages. Although managed languages have their own well known benefits such as higher developer productivity, better memory safety and application portability, they in turn rely on a managed runtime environment (MRE) to efficiently execute them [Mic19, Jam19, Azu19]. The MRE itself inevitably adds to the computational workload and the possibility arises when porting such a system to a new architecture, to introduce unforeseen overheads and design trade-offs.

From our experience of porting MaxineVM to ARMv7 and ARMv8, one such example arises from the variance of hardware support for self modifying code (SMC) across different architectures. For example, the x86 family of processors provide strong hardware support for SMC offering a total store order memory consistency model [SSO+10], and instruction/data cache coherence, based upon techniques such as snooping, cache invalidation and processor pipeline flushing. On the other hand, lower power processors typically employ weaker memory consistency models, and provide less hardware support for SMC, instead requiring programmers and compilers to ensure that the appropriate combinations of memory barriers and other synchronization measures required by the architectures are used correctly [RO16, SFP+20].

Managed runtime environments (MRE) typically rely on a phase of interpreted execution of platform agnostic bytecode during which profiling information is collected. These data profiles are used to guide the runtime to perform the most profitable

Just-in-Time (JIT) compilations and dynamic optimizations, generating native code to where control must be directed. Typically, code is optimized at the method or block granularity and execution is redirected to the optimized code by calling, i.e. branching between the interpreted code and the compiled methods or blocks. Furthermore, there are cases where MREs may need to re-compile some code segments. Such cases include: *de-optimization* [HCU92], where a speculative assumption made during compilation is found to be invalid; and *tiered compilation* [PVC01], where code that is on the "hottest" path is selected to be further optimized using more aggressive compiler optimizations.

In order to include newly JIT compiled code onto the execution path, MREs employ runtime code patching: i.e. they modify existing code *on the fly* in order to redirect control to the new compilations. Two possible strategies include: either eagerly patch all currently compiled call-sites to the affected target, or patch the entry points of the old target such that it will lazily patch referring call-sites the next time it is executed.

The implementation of call-sites and the associated strategies for their runtime code patching have not been thoroughly studied, in part because the most commonly targeted execution platforms for managed languages feature strong hardware support for SMC. On such platforms simply overwriting the target address of a call instruction is sufficient in most cases to redirect a call-site to a different target without further concern.

In this work, which extends our previous study [HZKL19], we examine how the absence of hardware support for SMC, typical of some low-power architectures, affects the implementation and performance of MREs. More specifically this paper contributes the following:

- It discusses and examines the constraints imposed by different architectures, such as x86-64 and AArch64 that affect their support for SMC in the context of JIT compiled code and MREs (see §2.4).

- It presents (see §2.5) and evaluates (see §2.7) four different safe call-site implementations for MREs and architectures without hardware support for SMC, showing their performance and corresponding patching overheads. The evaluation is performed on three diverse 64–bit ARMv8 platforms against microbenchmarks and three standard JVM benchmark suites (for the best three call-site implementations) using MaxineVM [KCR+17, WHVDV+13].

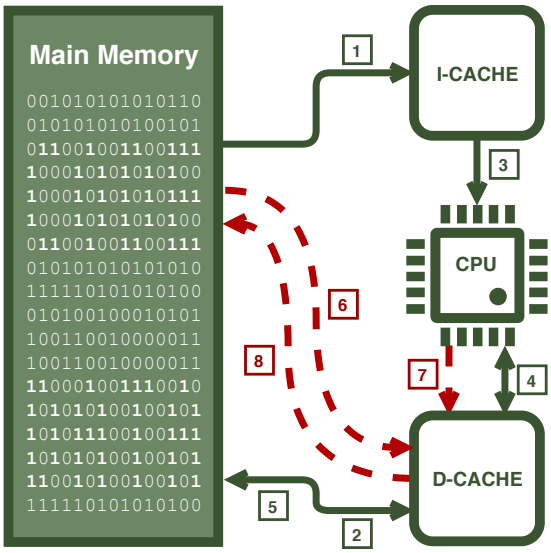

Figure 2.1: Code-patching illustration.

- It shows that different call-site implementations exhibit not only different capabilities and implementation complexity, but also reveals performance variations of up to 12%.

2.3 Background

Self-modifying code is a natural consequence of JIT compilation in MREs. During the course of executing its workload the runtime compiles new native code, replaces existing code and re-steers execution as required. Together, the MRE and its workload operate as a single operating system process that generates and modifies its own code.

2.3.1 Harvard Architectures and SMC

Contemporary processors are typically based on modified Harvard architectures [HP12] where separate instruction and data caches are used to optimize performance by reducing the latency required to access main memory, both for data and instructions. In such architectures, the instruction cache and pre-fetch unit are utilised to try and fetch ahead of time the instructions that are likely to be executed in the near future.

Figure 2.1 illustrates the data-flow of a modified Harvard architecture in the context of code patching. Solid green arrows indicate the typical data-flow, whilst dashed red arrows indicate SMC data-flow. Main memory contains both code and data. Code

from main memory is placed in the instruction cache (I-Cache) after a fetch by the I-Cache pre-fetcher or on resolution of I-Cache misses 1. Similarly the data cache (D-Cache) caches data from main memory 2. Based on the current value of the Program Counter (PC), the processor reads instructions from the I-Cache and executes them 3. Some of these instructions might modify data, resulting in writes to the D-Cache 4 that are written-back to main memory 5 at a point in time that is determined by the processor's memory consistency model [Gha96]. With SMC, read/write accesses for code modifications may require the processor to fetch code into the data cache first 6. Consequently, the modified code is written to the data cache 7 that is then written-back to the main memory 8 and re-fetched to the instruction cache 1 in order to become visible to the processor, and eventually be executed.

The involvement of two kinds of caches (I-Cache and D-Cache), means that maintaining code coherency across the system is more complicated than maintaining data coherency. As a result, different architectures provide different hardware support for SMC [Int15] in a similar manner as they implement different memory models. Consequently, self-modifying-code must comply with the underlying architecture's coherency constraints to ensure that the code updates become visible for execution.

2.3.2 Categorization of Processors

The hardware support that an architecture provides for SMC includes elements of the ISA, memory consistency model and the cache coherence protocol. The stronger the guarantees the architecture provides, the more intuitive it is to program, but it also becomes more complex to implement and scale to a high number of cores. Contemporary processors that provide strong guarantees regarding both data and code coherence typically belong to the x86 family and target high performance. Conversely, architectures that are designed to prioritize a low power envelope, tend to offer weaker guarantees regarding data and code coherence and provide a reduced instruction set. Examples of such architectures are ARMv7, ARMv8 (AArch32/AArch64) and RISC-V. For the discussion in the remainder of this paper we categorize processors with the terms high-end and low-power.

In the next section we examine how the architectural features that have been outlined so far, affect SMC on both high-end and low-power processors. We focus the discussion on the implementation details of call-sites that enable them to be safe to both concurrently execute and patch in MREs such as the Java Virtual Machine (JVM).

2.4 Architectural Constraints on SMC

General purpose MREs were initially/primarily designed and optimized for high-end architectures with sufficient hardware resources to support the extra overheads of MREs whilst still meeting acceptable performance criteria. As a result, the implementation of some design decisions on low-power architectures may introduce unexpected performance overheads when compared to corresponding high-end architecture implementations. However, we note that there are examples of runtimes such as JavaME and Android that have been developed specifically to target resource constrained devices. Each type of architecture or processor design has different performance targets in a predefined power envelope. This includes different versions of ARM and x86 architecture families and asymmetric multicore processors [CBGM12] such as ARM's big.LITTLE. By properly sizing the different hardware units of a chip (e.g., caches, branch predictors, etc.) along with additional implemented hardware features (e.g., vector instructions, specific hardware accelerators, etc.) microprocessors can scale up or down depending upon their targeted performance and power.

In addition to these design decisions there are other factors that influence the performance and behavior of different architectures when considering the implementation of SMC. These include: *i)* limitations imposed by the instruction set architecture (ISA), *ii)* explicit synchronization required by both the memory model and cache coherency protocol, *iii)* limits to the class of instructions that can be safely executed and modified concurrently.

In the remainder of this section we elaborate these points further.

2.4.1 Call-Site Size, Patch Size, and Atomicity — ISA

Call-sites comprise an instruction, or sequence of instructions, that perform a procedure call by directing control from the current PC to a new target address. ISAs with fixed length instruction encoding have a hard limit to the size of their operands, and this leads to constraints on the maximum range of PC-relative single instruction calls. The A64 ISA for AArch64 defines a branch with link instruction, `BL <target>`, that branches to a $\pm$128MiB PC-relative target, setting the link register to the return address. Consequently, we can see that for fixed length ISAs it is necessary to use a sequence of instructions to reach targets that are beyond the range of the single instruction common case. For example in AArch64 it is possible to use three instructions (for

```
1 ADRP X16, CALL_TARGET
2 ADD  X16, X16, :lo12:CALL_TARGET
3 BLR  X16
```

Listing 2.1: AArch64 call-site example for a long-range target.

long-range targets up to $\pm 4GiB$) as shown in Listing 2.1. The code in Listing 2.1 essentially initializes a register (`X16`) with the 4KiB memory page containing the target address (line 1), adds the low 12 bits to form the address (line 2) and branches to it (line 3).

In comparison, the x86-64 ISA [Int19] has a variable length encoding that supports wider operands, an example of which is the `CALL <target>` instruction. One variant: `CALL rel32` is 5 bytes wide and supports a 32-bit displacement operand that is added to the program counter (PC) to form a $\pm 2GiB$ PC relative target address, sufficient for the majority of calls. For calls required to go further than $\pm 2GiB$ from the program counter, `CALL` *`r/m64`* can be used to load the absolute 64-bit target address from *`r/m64`* into the program counter.

The ISA therefore defines the sizes of both the call-site and its potential patch. The patch-site can be as small as a part of an instruction (patching an operand) or up to a few instructions. Additionally, specific attention must be given to ensure that the patching operation appears to be atomic to the executing application code. Specifically, if the ISA does not provide a wide enough atomic write instruction to ensure the patching is atomic, then it is up to the managed runtime system to provide that guarantee.

2.4.2 Visibility and Timeliness — Memory Consistency and Coherence

Having discussed the implications imposed by the ISA, we now consider those related to the memory model of the architecture. In order to modify instructions, a managed runtime system will write to the affected memory addresses via the data caches of the processor, as outlined in section 2.3.1. If the coherency protocol of the processor does not include the instruction caches, then it becomes the responsibility of the managed runtime to ensure that any modified instructions written to the data caches are visible to the instruction fetch stream, at the appropriate time. Furthermore, in a multiprocessor system, any shared memory locations containing instructions currently being modified could be in-flight along the execution path on another core.

For AArch64 both of these conditions require mitigating action in order to fully

```
1 DC CVAU, Xn ; Clean data cache by VA to point of unification (PoU
    )
2 DSB ISH     ; Ensure visibility of the data cleaned from cache
3 IC IVAU, Xn ; Invalidate instruction cache by VA to PoU
4 DSB ISH     ; Ensure completion of the invalidations
5 ISB         ; Sync. fetched instr. stream
```

Listing 2.2: AArch64 JIT compiler side synchronisation [ARM19a].

```
1 ISB ; Sync. fetched instr. stream
```

Listing 2.3: AArch64 call-site side synchronisation [ARM19a].

synchronize the instruction and data streams. Listing 2.2 presents the instruction sequence that should be executed by the processor that has performed any code modifications; and listing 2.3 shows the code that should be executed by all cores that could observe the same set of modified addresses. Line 1, listing 2.2 cleans the data cache line containing the modified code address to the *point of unification* (PoU) [ARM19a]. Line 3 then invalidates the cache line in the instruction cache. Data synchronization barriers at Lines 2 and 4 are used to ensure that any prior data read/write instructions have completed before the barrier completes. Finally, at line 5 is an instruction synchronization barrier that causes the executing processor core to discard any instructions from its pipeline and prefetch buffers, ensuring subsequent instructions will then be fetched from cache or memory. The same `ISB` instruction must then be executed by the remaining cores running the current process if they may subsequently execute the modified instructions.

The programmer/compiler inserted synchronization inevitably incurs a performance overhead. Consequently, the placement of such barriers creates a trade-off between the time-delay for patched code to become accessible, and how much overhead we pay per call-site patching. Skewing the time that the patched code becomes visible may result in different cores running different versions of the code having a potential impact on both application performance and its correctness. In the case of tiered compilation, performance could be negatively affected by failing to invoke the latest and highest-performing version of a compiled method. In the case of de-optimization, correctness may be affected by invoking an optimized version of the code that was based on an assumption that is no longer true.

In contrast to AArch64, the x86-64 architecture provides hardware coherency between the instruction and data caches, and differentiates between self and

cross-modifying code [Int16]. Self-modifying code, where instructions are modified and executed by the same processor core, are handled transparently to the runtime, and are automatically flushed from the processor's pipeline. Although cross-modifying code does not provide this guarantee, the 4-byte displacement operand of a `CALL` instruction can be safely patched provided it is suitably aligned [Dev20, Ros01].

Although ARMv8 processors typically rely on software to maintain coherency between instruction and data caches, as just described, version 8.5-A of the architecture introduced the option for implementations to include instruction caches in the hardware cache coherency. The capability is reported in the cache type register `CTR_EL0`, where the bit fields `DIC` and `IDC` indicate whether explicit cache maintenance instructions are required. Specifically, if the `DIC` flag is set, then instruction cache invalidation is not required (line 3, listing 2.2). If the `IDC` flag is set, then the data cache clean to PoU is not required (line 1, listing 2.2). We will return to this feature which is relevant to our evaluation platforms in section 2.7.1.

2.4.3 Patchable Instructions — SMC Support

The previous two measures ensure that patching operations are both atomic, and any subsequently altered code is available to the instruction fetch stages of all processors. There is a further dimension, namely, any restrictions imposed by the architecture as to the set of instructions that can be concurrently modified and executed. Some architectures impose no restrictions including x86-64, SPARC, and RISC-V. POWER permits only two instructions to be concurrently modified and executed (`B` and `ORI 0,0,0` – direct branches and no-op). ARMv8 limits the instructions to eight (`B`, `BL`, `BRK`, `HVC`, `ISB`, `NOP`, `SMC`, and `SVC` [ARM19a]). In any other case, *"Concurrent modification and execution of instructions can lead to the resulting instruction performing any behavior that can be achieved by executing any sequence of instructions that can be executed from the same Exception level"* [ARM19a].

In order to modify any other instructions outside of this set, the MRE needs to ensure exclusive access to the corresponding addresses by suspending execution of concurrent threads, for example by performing a "Stop-the-World" (StW) operation. StW is essentially a pause of all application threads in order to allow for the managed runtime to perform operations that cannot be safely performed while the application is running. StW pauses are mainly used by garbage collection algorithms to ensure that while collection happens, the state of the heap does not change by the action of mutator (application) threads. In the case of call-site patching, a StW pause ensures that the

application is not running the code being patched, making the patching operation safe (no matter the instructions used in the call-site implementation).

2.5 Call-Site Implementations

Having outlined the design constraints that must be considered for SMC, we present four different viable call-site implementations. Each scheme can be safely patched and executed concurrently. For each case we present the implications for both hardware and software. We show implementations that target AArch64, however the strategy and discussion are also relevant to other architectures.

Together with the constraints outlined in the previous section, the specification for AArch64 limits the implementation of patchable call-sites by:

1. Limiting the range of direct calls. The range of a `BL` instruction is limited to $\pm 128MiB$ PC relative displacements.

2. Supporting up to 128–bit atomic writes. Stores must be naturally aligned in order to be atomic. Note that 128-bit atomic writes are supported using Load-Exclusive Pair (`LDXP`) and Store-Exclusive Pair (`STXP`) instructions.

3. Requiring explicit cache maintenance operations and memory barriers. The memory model and cache coherency protocol require software to schedule all appropriate cache maintenance operations and memory barriers to ensure that any patched instructions are available to the instruction fetch stage.

4. Limiting the set of instructions that can be concurrently patched and executed.

2.5.1 Direct Calls

In common with other ISAs, A64 provides a single instruction to facilitate procedure calls to a PC-relative literal offset. The branch with link instruction enables calls within $\pm$128MiB of the branch instruction. As the instruction resides in the set that is safe to concurrently modify and execute, the old branch instruction can simply be overwritten with another without exclusive access. The new branch can be made visible to instruction fetch stages by the cache maintenance sequence, if necessary. The benefits of this approach are simplicity and performance and its disadvantage is the limited range of the call.

Although the ±128MiB range would enable a code cache large enough for many applications, it is nonetheless a hard limit. Some techniques employed by managed runtimes may increase the expected amount of code, or decrease the effective utilization of the code cache. For example, tiered compilation may result in multiple cached versions of the same code compiled at different optimization levels, or with different speculative assumptions. In addition, managing partitions of the code cache using techniques common to GC algorithms, for example, *mark-sweep* or a *semi-space* scheme, creates temporary empty space within the addressable range of the code cache. Both OpenJDK and MaxineVM make use of these two approaches respectively for managing the more expendable *profiled* (OpenJDK) [HNG14] and *baseline* (MaxineVM) [WHVDV+13] code.

We also note that other contemporary RISC ISAs have more constrained direct calls. For example RISC-V supports only ±1 *MiB* PC-relative displacements (`JAL`), and POWER ±32 *MiB*. For these reasons we explore alternatives that avoid the limited range of immediate branches.

2.5.2 Absolute-load Indirect Calls

The `Absolute-load Indirect` scheme loads the address of a call target from a fixed address and branches to it. Listing 2.4 shows an example where lines 1–3 initialize register `X16` with the location that holds the target address, line 4 loads the address, and line 5 performs the call. Note that the example in listing 2.4 assumes the standard AArch64 48–bit address space.

With this approach the managed runtime can maintain the mapping of call target addresses anywhere in the address space. In order to patch a call, the target address is simply overwritten, and since no code is patched there is no need for synchronization or cache maintenance. In order to ensure atomic updates, the address holding the target must be 64-bit aligned. When using this scheme, any location within the virtual address space of the runtime can be reached from the call-site. The other advantage of this approach is the patching simplicity, since each callable procedure has the address of its entry point stored at a single location, it needs only to be updated once, rather than at each call-site. The downside is that all call-sites comprise five instructions, and are on the fast-path.

```
1 MOVZ X16, #0xABCD ;Craft the address
2 MOVK X16, #0xEF89, lsl #16 ; holding
3 MOVK X16, #0x7650, lsl #32 ; the target
4 LDR  X16, [X16]
5 BLR  X16
```

Listing 2.4: AArch64 absolute-load indirect branching. Note that this code assumes the common 48-bit address space, implementations with a 52-bit address space will require an additional `movk` instruction to set the uppermost 16-bits

```
1 LDR  X16, CALLEE_1
2 BLR  X16
3 ...
4 RET
5 CALLEE_1: .quad 0x0123456789ABCDEF
6           ...
7 CALLEE_N: .quad 0x01234ABCDEF56789
```

Listing 2.5: AArch64 relative-load indirect branching.

2.5.3 Relative-load Indirect Calls

Making use of PC-relative addressing we can improve on the `Absolute-load Indirect` scheme and store the target address of the call close by, for example, after the callers compiled code. In order to perform the call, shown in listing 2.5, the target address is simply loaded and then branched to. As in the previous example, no code is altered, and so no explicit synchronization or cache maintenance is required; however, the location holding the target addresses must be 64-bit aligned in order for updates to be atomic. This approach improves on the previous example by only requiring two instructions to perform the call, although a further 64-bit quantity is required to accommodate the target address. This approach is limited by the range of the PC-relative load, $\pm 1MiB$ for AArch64, so in order to be feasible the compiled method and storage for its call target addresses must be within this range.

Non-AArch64 compatible variation: We note that for architectures that do not restrict the set of instructions that can be concurrently modified and executed, it would be possible to combine this technique with the direct PC-literal call as follows. To transition from a `Relative-load Indirect` call to `Direct`, the branch register in listing 2.5 could be replaced with a direct branch and the load instruction, now redundant, replaced with a no-op. Reversing these steps would facilitate the transition from `Direct` to `Relative-load` indirect.

```
1 BL SHORT_TARGET ; or TRAMPOLINE
2 ...
3 RET
4 TRAMPOLINE:
5     LDR X16, CALLEE1
6     BR  X16 ; Don't link
7     CALLEE1: .quad 0x0123456789ABCDEF
```

Listing 2.6: Trampoline implementation with Relative-load Indirect call-sites.

2.5.4 Trampolines

Another approach for safe patching of call-sites is to use trampolines. A trampoline is a bimodal call scheme that completes a call in one or two legs. At the call-site, the scheme is implemented using a single `BL` instruction, that can be safely patched without exclusive access. The `BL` instruction can either branch directly to the target method (if it is within the range of a direct branch), or to an out-of-line trampoline which is a long-range call-site, see Listing 2.6. Note that at the trampoline there is a plain branch without linking (line 6), this way when the callee returns it does not return to the out-of-line trampoline but right after initial `BL` (line 1). This approach is currently being used by the OpenJDK AArch64 port. A limitation of this approach is that the out-of-line trampoline still needs to be within the range of the direct branch at the call-site. This can be achieved by reserving space for a trampoline per callee after the compiled code, in a similar manner to the way we allocate space for target addresses for `Relative-load Indirect` branching.

2.5.5 Patching only at Safe-points

Managed runtimes are able to suspend their hosted applications at *safe-points* in order to perform critical operations, for example, elements of garbage collection or deoptimization. Safe-points are usually implemented as a poll to an address on a special memory page and are injected by JIT compilers at or just after calls, or other points where application thread state is well known. At a safe-point the stack and heap can be safely manipulated whilst ensuring that a thread's view of the world remains consistent when it leaves the safe-point. To enable the safe-point, the runtime will protect the memory page causing application threads to trap and cooperatively suspend execution until resumed by the runtime.

This mechanism could also be employed for runtime code patching, thereby avoiding the complexities of concurrent modification and execution of code. Under such a

mechanism, a thread patching a call would have exclusive access to the code and would therefore be able to use any instructions to generate the patch. One example would be to use the scheme in listing 2.1. This approach requires three instructions and the address generation sequence `ADRP`, `ADD` is typically optimised by the microarchitecture [ARM20a, ARM17a, ARM17b, ARM20b]. Note also that the otherwise incompatible approach described in section 2.5.3 would also be permissible at a safepoint . Application threads upon resumption would synchronise their instruction streams, for example with the barrier in listing 2.3. An early version of the AArch64 OpenJDK port [HD14] employed this technique to avoid the complexities of call-site patching imposed by the architecture. The benefit of this approach is the ability of the managed runtime to use any instruction sequence to both implement and patch a call-site. The drawback is the penalty of bringing the application to a standstill during patching which may be prohibitive, particularly for latency sensitive applications.

2.6 Comparison of Call-Site Implementations

In this section we show a comparison of the different call-site implementations that we presented in § 2.5. Table 2.1 presents a summary of this comparison based on the following criteria:

- Whether code-patching or data-patching is required.
- Whether SMC support is required.
- The code size of the call-site implementation.
- The complexity of the patching code.
- The supported target range.
- Whether patching requires a “Stop-the-World” pause.

2.6.1 Code vs Data Patching and SMC Support

We characterize each implementation based on whether patching needs to alter code, data or both. Implementations that require code patching, e.g. Direct (§ 2.5.1), and Trampolines (§ 2.5.4) cannot be implemented in the absence of SMC support whether by hardware of software. On the other hand, implementations requiring only data

patching, e.g. Absolute-load Indirect (§ 2.5.2), and Relative-load Indirect (§ 2.5.3) do not require SMC support since they do not alter the code itself, rather a memory address containing the actual target address.

2.6.2 Code Size

Code size is an important metric not only because larger code size is associated with more instructions and thus more cycles, but also because it impacts cache performance, especially in embedded systems. For each implementation we report the number of 32-bit instructions. In addition to the instructions required for a call, both `Relative-load Indirect` and `Trampolines` require a further 64-bit quantity that specifies the target address of each unique call target in a compiled method. These 64-bits reside with the code and contribute to the overall code size and may impact code cache performance. For `Relative-load Indirect` this is marked as +2 in Table 2.1. `Trampolines`, depending on the call-range, require execution of a single instruction (for short-range calls) or up to three instructions (for long-range calls). For short-range, the reserved space is four instructions wide, while for long-range the reserved space is two instructions wide (marked as +4 and +2 respectively in Table 2.1)

2.6.3 Patching Complexity

The complexity of the patching code, although not on the common path, is another interesting metric when it comes to call-site implementations. The patching complexity relies heavily on whether the call-site requires code or data patching. When data-patching, an overwrite of the old address is sufficient to make future calls jump to the new target. However, different implementations have different complexity. In the `Absolute-load Indirect` branching (§ 2.5.2) case where we keep a single central 64-bit value for each compiled method, a single 64-bit write is enough to effectively patch all the call-sites targeting the corresponding method. On the contrary, in the `Relative-load Indirect` (§ 2.5.3) case we need to visit each caller method and overwrite the corresponding 64-bit inline segment, increasing the complexity from $O(1)$ to $O(n)$, where n is the number of methods that invoke the corresponding method at least once.

In implementations requiring code-patching, the complexity increases even more. In code-patching we need to patch each call-site separately, even if multiple call-sites with the same target reside in the same method, resulting in a complexity of $O(m)$,

Table 2.1: Comparison of call-site implementation approaches.

	Direct	Absolute-load Indirect	Relative-load Indirect	Trampolines	Any Safe-point
Characterization	Code	Data	Data	Code + Data	Any
Requires SMC support	Yes	No	No	Yes	Maybe
Size (in instructions)	1	5	2 +2	1 +4 to 3 +2	Any (≥ 1)
Patching Complexity	Medium	Low	Medium	High	Any
Supported Call Range	Limited	Any	Any	Any	Any
Stop-the-world pause	No	No	No	No	Yes

where m is the number of call-sites that invoke the corresponding method and $m \geq n$. To effectively patch all call-sites invoking a method, we need to either go over all the compiled methods and eagerly patch the corresponding call-sites, or patch the current compiled version of the callee such that it lazily patches call-sites when next executed. The latter allows patching to run in constant time, but penalizes the first execution of each unpatched call-site.

2.7 Evaluation

The evaluation consists of two parts: firstly we use our own microbenchmark, and secondly three well known JVM benchmark suites that we run with the open source MaxineVM [WHVDV+13]. The microbenchmark experiments enable us to perform a fine grained microarchitectural and performance analysis of implementations of each of the four call site schemes outlined in section 2.5, on three diverse 64-bit ARMv8 platforms. Based on the observations from the microbenchmark results we then implement three call-site schemes in MaxineVM, and evaluate the performance using results from the Renaissance, DaCapo and SPECjvm2008 benchmark suites. In the following sections we describe the experimental platforms followed by the experiments and results.

2.7.1 Experimental Platforms

We used three hardware platforms for our experiments: AppliedMicro X-Gene 1, Graviton2, and Odroid C2. All three platforms were used for the microbenchmark experiments. However, only the first two were used for the JVM benchmarks, and the third platform (Odroid C2) was omitted because it has insufficient memory to run all benchmarks.

APM X-Gene 1: The Applied Micro X-Gene X-C1 Evaluation Kit is equipped with a APM883208-X1 8 core processor [Wik18], clocked at 2.4GHz and 32GiB of DRAM. The processor has a 4-wide out-of-order superscalar microarchitecture [GSF12]. Each core has 64KiB L1 cache (32KiB instruction + 32KiB data), each pair of cores shares 256KiB L2 cache and all cores share 8MiB of L3 cache. The X-Gene has Debian 9u5 installed and runs Linux 4.9.0-8-arm64. The X-Gene 1 was one of the first ARMv8 system-on-chips to be manufactured, and the first server on chip of that architecture.

Graviton2: The second system, an Amazon AWS Graviton2, is based on the ARM Neoverse N1 processor [ARM19b]. The processor is clocked at 2.5GHz, has a 4-wide front end, an 11 stage accordion pipeline and can dispatch and commit up to 8 instructions per cycle. Each core has 128KiB L1 cache (64KiB instruction + 64KiB data), 1MiB L2 cache, and all cores share 32MiB of L3 cache (the die comprises 64 cores). We used two separate instances of the Graviton2: m6g.2xlarge and m6g.metal. The m6g.2xlarge has 8 cores and 32GiB of DRAM and the m6g.metal has 64 cores and 256GiB of DRAM. The Graviton2 has Ubuntu 18.04 LTS installed and runs Linux 5.3.0-135-aws. On virtualized instances such as the m6g.2xlarge, the performance monitor unit (PMU) registers cannot be accessed directly from userspace programs. As our microbenchmark experiments require access to the PMU registers, discussed further in the next section, we used the m6g.metal instance for the microbenchmarks and m6g.2xlarge instance for the JVM benchmarks. The Neoverse N1 has an option for the instruction cache to be coherent with other caches within the coherency domain including the data cache [ARM20c], as discussed briefly in section 2.4.2. We found that the Graviton2 is susceptible to errata #1542419 [ARM20d], and the CPU reports itself as not instruction cache coherent via the cache type register (CTR_EL0.DIC == 0). Instruction cache maintenance instructions are therefore required at EL0, following stores to addresses that contain instructions. The N1 platform represents the current state of the art of ARMv8 server processors.

Odroid C2: To complement the two out–of–order CPUs we also ran the microbenchmarks on a dual issue, in–order, quad–core Cortex–A53 Odroid C2. Each core has 64KiB L1 cache (32KiB instruction + 32KiB data), all cores share a unified L2 cache of 512KiB. The processor is clocked at 1.54GHz and the board has 2GiB of DDR3 DRAM. Due to memory constraints it is not possible to run all Java benchmarks on the Odroid and only the microbenchmarks results are presented. The Odroid has Ubuntu 18.04.6 installed with Linux kernel version 3.16.72-46. The Cortex–A53 is the

most widely deployed ARMv8 CPU [Ltd20].

2.7.2 Microbenchmark

The microbenchmark performs a fixed number of invocations of a target callee function using each of the four schemes. We used a total of two callees: one was used as the default target and the other as the patching target. For each call–site implementation we generated the corresponding sequence of assembly instructions. We timed the experimental runs, and collected event counter values from the PMU of each platform, enabling us to measure the performance in a cycle accurate manner. Furthermore, we were also able to estimate the cost of the patching code per call-site implementation by isolating its execution.

In order to avoid introducing front–end stalls, we separated each inline call–site from its neighbor with a fixed integer computation comprising 7 instructions. This formula was determined empirically by observing the effect of the padding on the front end stall count. The number of instructions is also inline with our observations from categorizing the benchmark workloads discussed in section 2.7.5.1, that show approximately 10% distribution of branch instructions in code, also observed by others [LTCC+16]. We aligned all branch targets to 8 byte boundaries. The Neoverse N1 software optimization guide [ARM20b] recommends aligning branch targets to 16 byte boundaries, however we found that there was no measurable difference in the microbenchmark results when compared to 8 byte alignment.

2.7.2.1 Methodology

We configured the microbenchmark to perform 1000 calls to a target function for each call–site scheme on each platform. For the `Trampoline` scheme we used the long range variant only, since at short range it is the same as `Direct`. On the Odroid-C2 we employed the userspace *CPUfreq* scaling governor and set the frequency to maximum to avoid perturbations arising from dynamic voltage and frequency scaling. On all platforms the process affinity was set using using the `taskset` command to pin the process to a single core, and we used a kernel module to enable access to the PMU registers by userspace programs running at exception level 0 (EL0). We ran four iterations of the experiment to populate the memory system, before timing and counting events around the critical section of code.

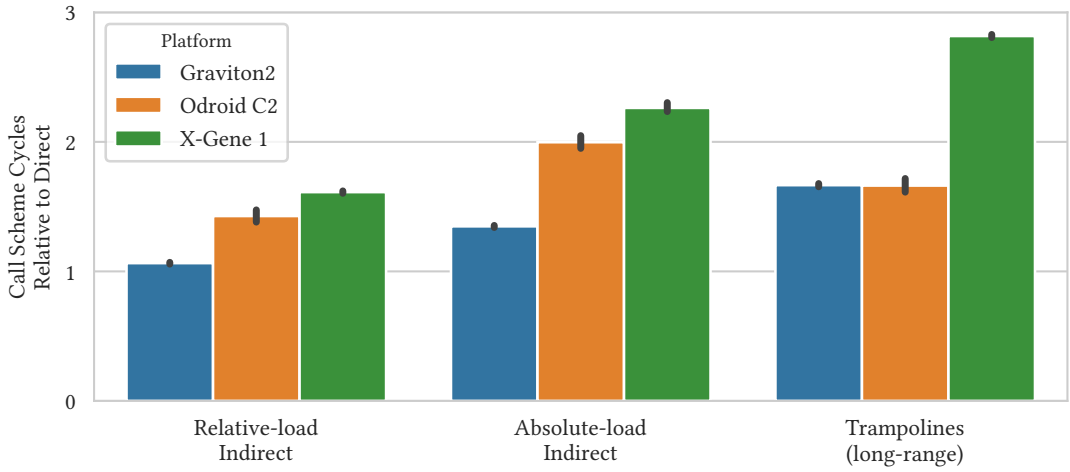

Figure 2.2: Microbenchmark results showing the ratio of cycles of the alternate schemes from section 2.5 when compared to `Direct`. Error bars show the 95% confidence interval.

2.7.3 Microbenchmark Results

Figure 2.2 shows the relative performance of a call for each scheme when compared to the `Direct` scheme on the same platform. The y–axis shows the number of cycles per call for each scheme divided by the number of cycles taken for the `Direct` scheme; the error bars show the 95% confidence interval. Table 2.2 contains the following statistics collected from the PMU during the experiment: the instruction throughput measured in instructions retired per cycle (IPC), the L1 instruction cache refill rate, and the branch mispredicton rate for each microbenchmark. The L1 instruction cache refill rate is shown as the percentage of L1 instruction cache accesses that cause a refill. The branch misprediction rate is the percentage of predictable branches that were mispredicted or not predicted.

The `Direct` scheme, being one instruction long, always consumes the fewest cycles. Furthermore, since the branch target is encoded in the instruction, it can be discovered earlier in the pipeline and subsequently used to generate the next fetch addresses by the prefetch unit. This shorter loop, when compared to the other schemes that require input from a register to determine the call target, ensures that the `Direct` scheme also obtains the highest instruction throughput, and the branch misprediction rate is 0% for all platforms. As expected, the Graviton2 achieves the highest instruction throughput for all schemes and also sustains a higher throughput relative to the Direct scheme. The microprocessor has a well provisioned prefetch unit containing

Table 2.2: Statistics collected from the performance monitoring units whilst executing the microbenchmark. Note that the abbreviations G2, C2 and XG refer to the Graviton2, Odroid-C2 and X-Gene 1 respectively. IPC is instructions retired per cycle.

	IPC			L1 I-Cache Refill Rate %			Branch Mispredict %		
	G2	**C2**	**XG**	**G2**	**C2**	**XG**	**G2**	**C2**	**XG**
Direct	3.2	1.1	1.4	0.8	1.9	10.1	0.0	0.0	0.0
Relative-load Indirect	3.2	0.8	0.9	1.3	3.6	10.2	0.3	0.2	7.1
Absolute-load Indirect	3.1	0.8	0.8	4.0	8.6	19.6	0.3	0.2	8.9
Trampolines (long-range)	2.2	0.8	0.6	10.3	4.7	9.2	0.2	0.2	17.6

a decoupled branch predictor able to run independently from, and ahead of the instruction fetch stage [PSB+20], and is possibly a variant of *Fetch Directed Prefetching* [RCA99].

The `Relative-load Indirect` scheme is the most efficient of the register indirect schemes presented on all platforms. The scheme incurs an overhead of approximately 1.4x per call for the Odroid and 1.6x for the X-Gene when compared to `Direct`. The observed overhead for the Graviton2 is almost negligible.

`Absolute-load Indirect` shows a 2.2x overhead compared to `Direct` for the X-Gene, 2x for the Odroid-C2, and 1.3x for the Graviton2. All platforms record a significantly higher L1 instruction cache miss rate with this scheme when compared to `Relative-load Indirect`, shown in table 2.2.

With the long–range variant of `Trampolines`, the Graviton2 and the Odroid C2 show a similar 1.6x penalty, and the X-Gene 2.7x. The Graviton2 shows the highest L1 instruction cache misprediction rate for this benchmark.

The X-Gene shows a low IPC relative to its four instruction issue width in comparison to the Graviton2 (quad-issue) and Odroid C2 (dual-issue). The high branch misprediction rates and high L1 instruction cache refill rate coupled to low IPC for all of the indirect schemes suggest that part of the reason is due to a failure to fetch instructions from the correct execution path. These results most be viewed historically as the X-Gene was an early implementation of the 64–bit ARMv8 architecture.

2.7.3.1 Patching

In addition to the execution of calls we also present the patching overheads. The majority of patching in a JVM takes place during the warm-up phase when newly compiled methods are linked. As such the patching overheads do not contribute significantly

Table 2.3: Number of cycles per patch operation.

	Direct	Absolute-load Indirect	Relative-load Indirect	Trampolines (long-range)
Graviton2	65	12	13	79
X-Gene 1	129	14	14	144
Odroid-C2	77	20	18	96

to peak performance. Table 2.3 presents the number of cycles for each platform and benchmark for a single call–site patch including all required cache maintenance for the target platform. For `Trampolines` we evaluate the case where we patch in a trampoline, as would be the case when going from a short-range single branch call to an otherwise out-of-range target. The schemes that modify instructions incur much higher overheads than those that store addresses for indirect-calls as a direct result of the cache maintenance operations and associated barriers and pipeling flushing. The cost of the barriers is smaller on the Graviton2 which has some hardware synchronization between the data and instruction caches, and does not require the modified data cache lines to be cleaned to the PoU unlike the other two platforms.

2.7.3.2 Limitations of the Microbenchmark

The microbenchmark is implemented as a linear sequence of unconditional branches to a target function. As such each call-site is executed once with no prior execution history in the branch prediction resources of the processor. The microbenchmark results represent the worst case, as might be expected when a call is encountered for the first time. The integer performance of the CPU will also affect the IPC due to the integer computation padding discussed in 2.7.2. The Graviton2 has four integer execution units, the X-Gene 1 has two and the Odroid C2 has one.

2.7.4 JVM Benchmarks

Based on the results of the microbenchmark we implement three viable schemes in MaxineVM: `Direct`, `Trampolines` and a version of `Relative-load Indirect`, discussed further in section 2.7.4.2. We evaluate the relative performance of the schemes using three well known JVM benchmark suites: Renaissance 0.11.0, DaCapo 9.12-MR1-bach and SPECjvm2008. We also conduct a further analysis using the DaCapo benchmarks in section 2.7.5.1 in order to extract further insights into some of the points that arose during experiments, as discussed below.

2.7.4.1 MaxineVM

MaxineVM is based on version 2.9 and also includes some bug-fixes not yet available upstream, the image is built using OpenJDK 1.8.0_265. MaxineVM's code cache is partitioned into three adjacent regions: boot, baseline and optimized (figure 2.8). The boot cache is populated during the image build phase and is a fixed size. The baseline and optimized code caches are configured and populated at runtime. The baseline cache occupies a managed semi-space heap scheme and the optimized cache is unmanaged, i.e. newly compiled optimized code can only be allocated into the optimized cache until it is full. If this condition is met MaxineVM will currently terminate. The default sizes for baseline and optimized caches are 256MB and 16MB respectively. In order to run MaxineVM using the `Direct` scheme it is necessary to constrain the size of the baseline code cache in order to remain within the 128MiB limit of a single `BL` instruction across the entire code cache. Typically the boot region is around 14MB for an AArch64 MaxineVM image, so a baseline code region of 98MB allows for the default 16MB optimized cache. This option is specified on the command line when running MaxineVM for all benchmarks using the `Direct` scheme with the option: `-XX:ReservedBaselineCodeCacheSize=98MB`. The option is also used for the `Relative-load Indirect` scheme discussed in section 2.7.4.2. Note that using the same setting for the `Trampoline` scheme would reduce it to the `Direct` scheme, since anywhere in the code cache would be reachable from the branch and link at the `Trampoline` call-site. For the `Trampoline` scheme we therefore use the default 256MB baseline code cache size to ensure that a fraction of calls require the long-range variant of the `Trampoline` to be utilized. All schemes therefore use single instruction direct calls within ahead-of-time (AOT) compiled boot image code, and at runtime the two alternate schemes transition to their implementation specific form for JIT compiled code. An implication arising from this constraint is that some benchmarks generate too much baseline code for the restricted code cache size and so fail to run. The benchmarks that can be run are enumerated in the sections that describe the suites below.

2.7.4.2 Relative-load Indirect, MaxineVM

In MaxineVM, the boot code cache contains position independent code only, since at the time the image is built, absolute addresses of locations within the code are not known. Calls in this region rely on relative addressing using the PC only. In order

to implement this scheme in MaxineVM without significant re-engineering we employ the `Direct` scheme in boot image code and transition to the `Relative-load Indirect` scheme at runtime. Effectively the core components necessary to start running MaxineVM will employ the `Direct` scheme, however new workload code compiled at runtime will use the `Relative-load Indirect` call scheme. This also means that calls from the boot code region need to be able to reach the entire code cache so it is necessary to constrain the code cache size as discussed in section 2.7.4.1. To distinguish this implementation of the scheme in MaxineVM, we will refer to it as `Indirect-Maxine` for the remainder of this chapter.

2.7.4.3 Affected calls

Maxine uses a conventional virtual dispatch scheme for all virtual and interface method calls. Hence for the experiments, the three alternate call-site schemes affect only calls to targets that can be resolved at JIT compilation time and can therefore be statically bound. This includes both static calls (those originating at an `INVOKE_STATIC` bytecode), and calls that can be devirtualized either speculatively, or calls to target methods with *final* or *private* attributes. The remaining virtual and interface invokes that are not devirtualized use the conventional virtual dispatch scheme and are unaffected in the experiments. We investigate this further in section 2.7.5.1.

2.7.4.4 Methodology

For all benchmarks a heap size of 16GB was set (-Xms16G -Xmx16G). The heap size was chosen to accommodate the largest working set of the benchmarks (Renaissance reactors) and to minimize any interference in the measurements due to garbage collection.

For both Renaissance and DaCapo we omit the early runs where the time is greater than 3 standard deviations of the mean of the remainder of the set. For SPECjvm2008 we used the default of one iteration running for 240s after an initial warmup of 120s. All of the SPECjvm2008 benchmarks were run except xml.transform which MaxineVM fails. From DaCapo we run: *avrora, fop, h2, jython, luindex, lusearch, lusearch-fix, pmd, sunflow, xalan*. Of the remaining DaCapo benchmarks, some are problematical on both MaxineVM and OpenJDK AArch64: *batik* (fails due to missing JPEG library), *eclipse* (fails due to raised Exceptions), *tomcat* (fails validation); or fail to start on Maxine: *tradebeans* and *tradesoap* (which are also now problematical on some

OpenJDK versions). From the Renaissance benchmark suite we run: *fj-kmeans, future-genetic, mnemonics, par-mnemonics, philosophers, reactors, rx-scrabble, scala-doku, scala-kmeans, scala-stm-bench7*. Of the remainder, five require further work to run reliably under MaxineVM: *finagle-chirper, finagle-http, db-shootout, neo4j-analytics*; *scrabble* exhibits non-deterministic behavior, and the rest generate too much baseline code to run given the constrained baseline code cache size: *akka-uct, als, chi-square, dec-tree, gauss-mix, log-regression, movie-lens, naive-bayes, page-rank, dotty*. For the Renaissance benchmarks, the default number of iterations of each benchmark was used. For DaCapo we run 30 iterations of each benchmark except fop, luindex and lusearch-fix where we run 100. Since these three benchmarks have short runtimes we used the larger number of iterations to reduce uncertainty in the plots.

2.7.5 Benchmark Results

The plots for the benchmarks are shown in figures 2.3, 2.4, 2.5 and 2.6. As the `Direct` scheme is the optimal solution, in the plots we show the results of `Indirect-Maxine` and `Trampolines` relative to `Direct`, where a y-axis value greater than one indicates a slowdown (higher is worse). For both Renaissance and DaCapo the error bars show the 95% confidence interval according to Fieller's theorem for the quotient of two means [Fie54]. The SPECjvm2008 results are for a single iteration and so there is no confidence interval. All of the plots include a *geomean* bar showing the geometric mean of all the benchmarks run from each suite.

From the plots we make two general observations. Firstly, for each benchmark the slowdown introduced by both Trampolines and Indirect-Maxine schemes is less on the Graviton2, where the bars tend more towards parity with the Direct scheme, than is observed for the X-Gene 1. Secondly, whilst the difference in performance between the two schemes is almost negligible for the Graviton2: a fraction of 1% in the geomean; for the X-Gene 1 there is 2% difference in the geomean for DaCapo and Renaissance, slightly less for SPECjvm2008, varying up to 5% (SPECJVM2008 xml.validation). Both of these observations are inline with the microbenchmark results.

Looking more specifically at the individual benchmark suites, the difference in performance between the two schemes is more significant with DaCapo, less so for Renaissance and some of the SPECjvm2008 benchmarks. From the SPECjvm2008 benchmarks the scimark [PM20] and crypto families reveal little difference in the performance between the two schemes. These two benchmark groups feature loop-based numerical computations and a small code footprint and are not revealing in this work.

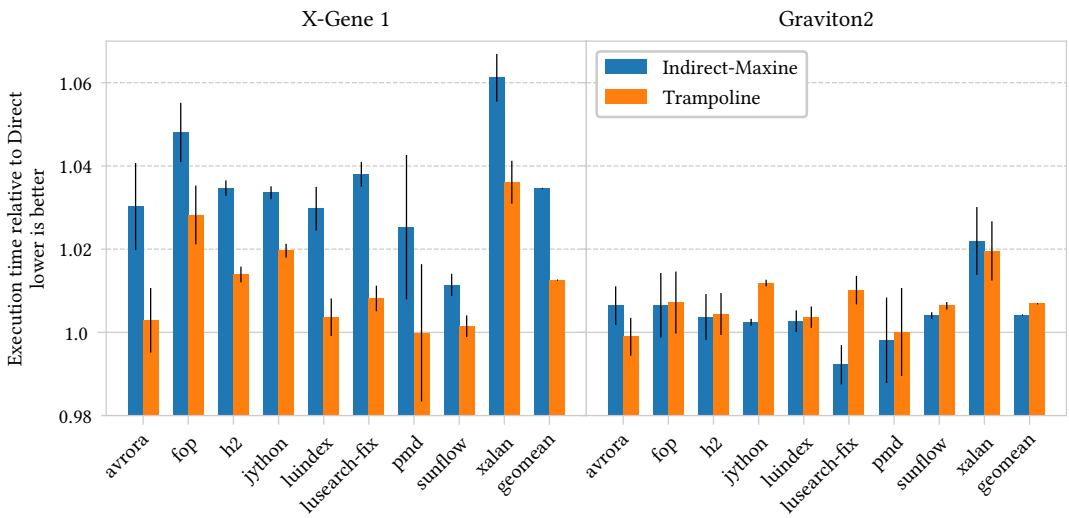

Figure 2.3: DaCapo benchmark results showing the performance of the call-site schemes, relative to the `Direct` scheme implemented on MaxineVM.

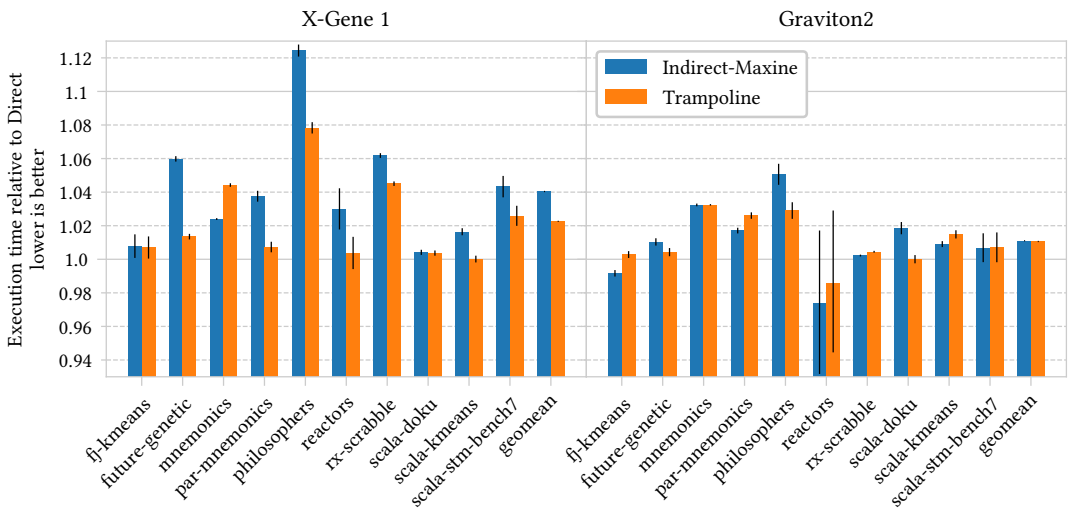

Figure 2.4: Renaissance benchmark results showing the performance of the call-site schemes, relative to the `Direct` scheme implemented on MaxineVM.

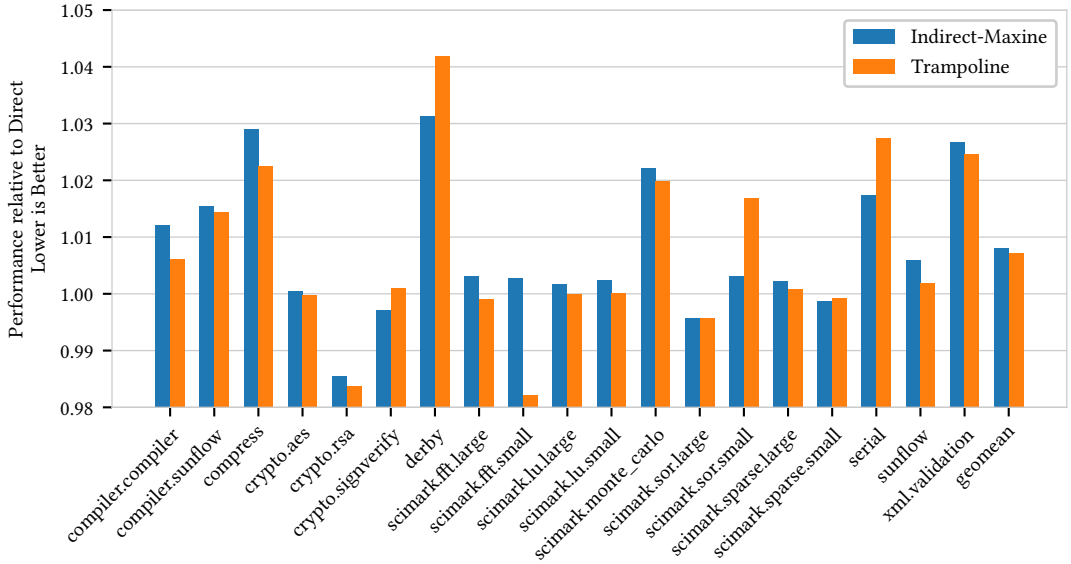

Figure 2.5: SPECjvm2008 benchmark results showing the performance of the alternate call-sites relative to the `Direct` scheme implemented in MaxineVM and run on Graviton2.

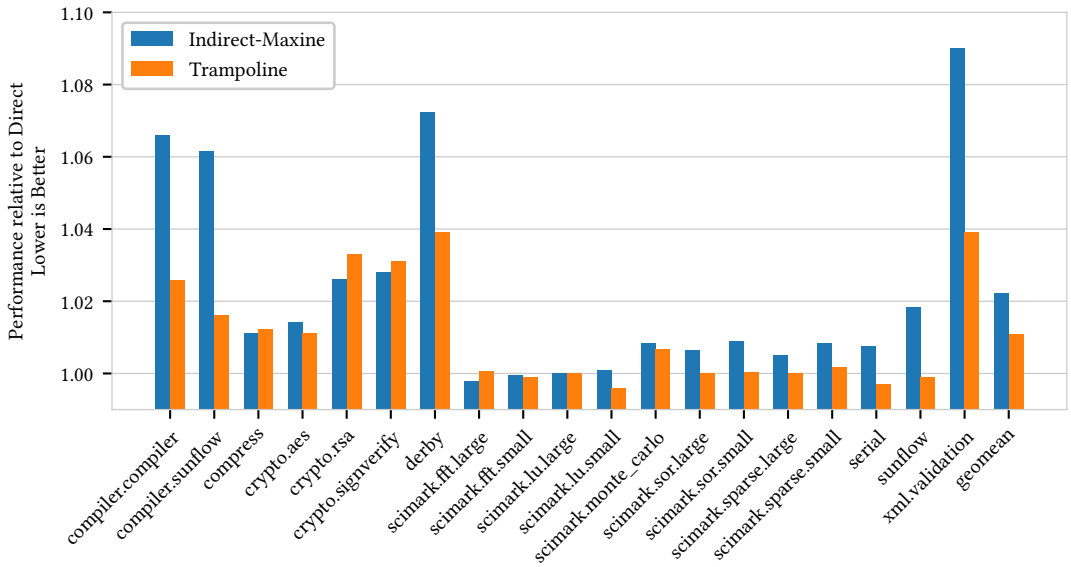

Figure 2.6: SPECjvm2008 benchmark results showing the performance of the alternate call-sites relative to the `Direct` scheme implemented in MaxineVM and run on X-Gene 1.

Other studies have shown that these benchmarks incur orders of magnitudes fewer procedure calls than the other benchmarks in the suite [PRL+19b] and are resistant to program-flow optimizations [HM11].

A number of the results (e.g. lusearch-fix / Graviton2 in figure 2.3) appear to suggest a better performance than `Direct` for one of the alternate schemes, indicated by a bar height of less than one. Some noise in timed benchmark experiments is to be expected with managed runtimes [BBTK+17], where execution includes VM components, such as JIT compilation and garbage collection, as well as the workload. Given that the results presented are for a single run of the experiment, multiple independent runs would help to eliminate some of the measurement uncertainty and allow a better confidence in the mean to be produced.

2.7.5.1 DaCapo, Further Analysis

In order to gain a further understanding of the variation in the results, we conduct and present a further analysis of the DaCapo benchmark workloads. Part of the motivation for this work is to understand how the distribution of calls in the workload might affect the results.

JVMTI agent: Firstly we implement a JVM Tool Interface (JVMTI) agent to instrument all call-sites from each benchmark that collects the properties of calls and their targets during a sequenced run. The results are shown in figure 2.7(a) and indicate the percentage of static calls, and calls to *final* or *private* targets. Collectively, this group defines the set of calls that can be statically bound by a JIT compiler with no side-effects. In practice, compiler optimizations such as inlining will reduce this set; and speculative devirtualization may add to it.

Perf Analysis: Secondly, during a run of DaCapo we collect PMU event counters by hooking into the callback mechanism of the benchmark suite and attaching to the process with the Linux *perf* utility. From the counts of register indirect, return and immediate branches we can estimate the percentage of calls from statically bound call–sites: this is shown by the total combined height of the bars in figure 2.7(b). This quantity identifies the percentage of calls affected in the experiment for the differing schemes. Furthermore, for the `Trampoline` scheme we can also estimate the percentage of statically bound calls that are out of range of an immediate branch and hence take the trampoline to reach their target, also shown in figure 2.7(b). We say a trampoline is *taken* if the indirect leg of the `Trampoline` has been used to reach the target. If the target is reached directly from the `BL` instruction at the `Trampoline` call-site, then

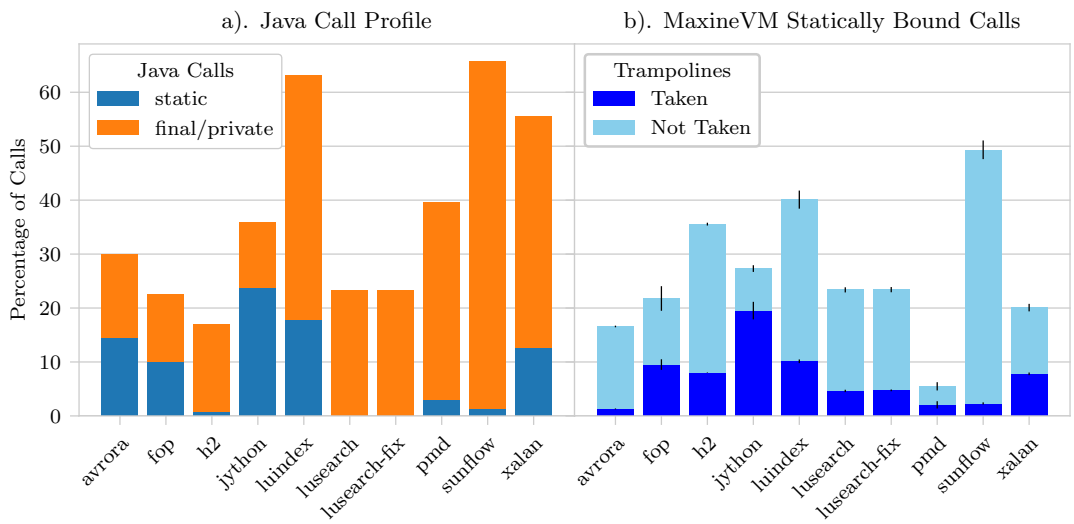

Figure 2.7: Showing the DaCapo call profile for Java and MaxineVM. For the DaCapo benchmarks we show: a). the profile of Java calls in the benchmarks; and b). the percentage of calls in the MaxineVM that are statically bound by the runtime (total bar height), as well as taken/not taken trampolines. We include lusearch in the results, this is the same workload as lusearch-fix albeit without a bug fix and it serves as validation.

we say that the trampoline is *not taken*.

Figure 2.7b shows that overall the number of trampolines taken as a percentage of all calls is fairly small, varying from about 1.5% for *avrora* and up to 20% for *jython*. One further observation from figure 2.7(b), shows that for *pmd* only 5% of all invokes were affected by the experiment. This observation is consistent with the results of the DaCapo benchmarks in figure 2.3 that show little statistically significant variation for *pmd* with the three alternate schemes.

The MaxineVM code cache has been described briefly in section 2.7.4.1. As a consequence of its organization, optimized code is located in a small partition, only 16MB by default. Intuitively, it would seem that as execution proceeds and hot code is optimized and hence co-located, then procedure calls ought to cluster more in the optimized region, where displacements are small, illustrated in figure 2.8. If so, calls made using the `Trampoline` scheme reduce to a single instruction branch and link, the same as the `Direct` scheme.

We were curious to find out if it were possible to observe an increasing bias towards calls taking only the direct leg in the `Trampoline` scheme as execution proceeds along the iterations. Figure 2.9 shows a plot of the percentage of statically bound calls, where the trampoline leg is taken per benchmark iteration. Most of the benchmarks show a

Figure 2.8: An illustration of the partitioned code cache in MaxineVM. The arrows indicate calls revealing the short-range displacements in the optimized partition.

general downward trend of trampolines taken, with *fop* showing the greatest reduction. The main exception is *pmd* that shows no clear trend – as already noted, there are very few statically-bound calls observed when running this benchmark. Although the percentage change is in general small for several of the benchmarks, there are reasons that might prevent a larger reduction. MaxineVM boot code cache contains many ahead-of-time compiled artefacts from the image build phase, such as intrinsic implementations, and other essential compiler stubs. Any calls from the optimized code cache to these compilations in the boot code cache region will be out of range of a single branch, and so would require the long-range leg with the `Trampoline` scheme. Organizing the code cache so that the boot and optimized regions are adjacent and so still within range of a single `BL` for AArch64 may yield a larger reduction in the number of trampolines taken, however we have not yet verified this.

2.7.6 Impact on Code Size

Table 2.4 shows the effects on code size contributed by the three schemes used in the benchmark results. We report the size of the code compiled by both baseline and optimizing compiler for the `Direct` scheme and the percentage increase for each of `Indirect-Maxine` and `Trampolines`. We use the same nine benchmarks from the DaCapo suite we used for the results in section 2.7.5, each run for one iteration only. We observe a fairly consistent additional code footprint for each scheme and compiler between 10% - 20%, with the exception of *fop* and *jython*. These two anomalies are caused in each case by methods that the baseline compiler generates more than 1MB of code for. This exceeds the range limit of a PC-relative load instruction making the

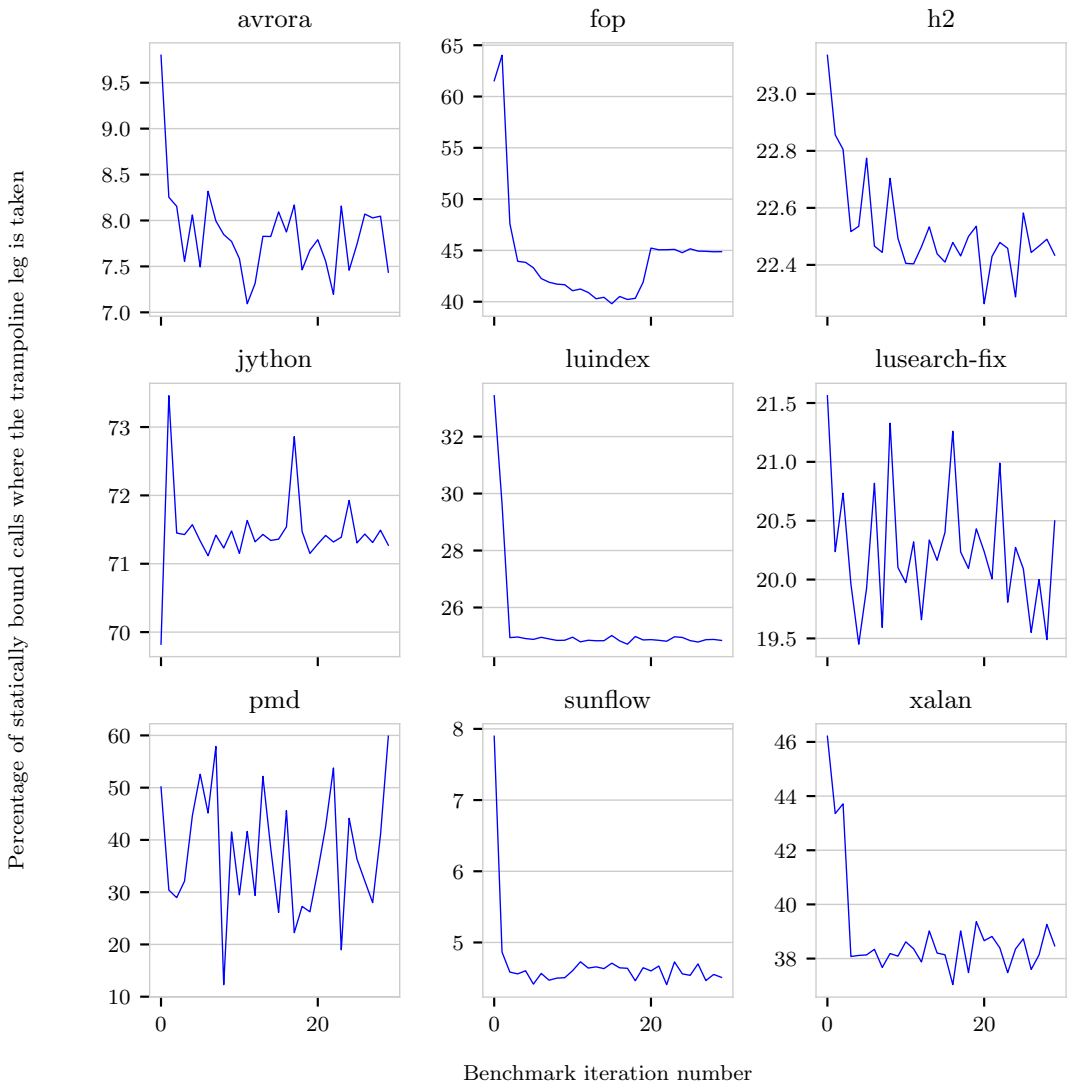

Figure 2.9: The change in trampolines taken as a percentage of statically bound calls, when using the Trampoline scheme for the DaCapo benchmarks.

Table 2.4: The impact on code size of `Indirect-Maxine` and `Trampolines` compared to `Direct`.

	Baseline Code Size			Optimized Code Size		
		Indirect			Indirect	
	Direct (kB)	Maxine	Trampolines	Direct (kB)	Maxine	Trampolines
avrora	5,306	+11%	+14%	134	+13%	+18%
fop	19,950	-3%	+16%	332	+320%	+19%
h2	7,187	+10%	+13%	514	+14%	+19%
jython	28,707	+4%	+13%	1,880	+26%	+9%
luindex	5,416	+11%	+14%	103	+13%	+18%
lusearch	3,824	+11%	+14%	170	+15%	+20%
lusearch-fix	3,826	+11%	+14%	170	+15%	+20%
pmd	8,477	+10%	+13%	458	+13%	+17%
sunflow	6,335	+11%	+14%	158	+11%	+15%
xalan	7,813	+10%	+13%	470	+14%	+19%

`Indirect-Maxine` scheme non-viable for these compilations. In each case the baseline compilation for these methods fails over to the optimizing compiler which generates better, more compact code that is within the range of a load instruction. Consequently both fop and jython generate less code than the trend for the baseline compiler and the `Indirect-Maxine` scheme combination; and more code than expected for the optimizing compiler / `Indirect-Maxine` combination. This illustrates the limitation of this scheme which we introduced in section 2.5.3. (For the timed benchmarks experiments in section 2.7.4 we explicitly compiled the three methods with the optimizing compiler for each scheme and platform to rule out any potential interference.)

2.8 Summary

Following our experiences of porting a managed runtime system, MaxineVM, to ARMv7 and ARMv8, we have discussed in this work the implementation of call-sites and associated safe code patching in JIT compiled code. We have discussed how the constraints imposed by the ISA, memory consistency model, cache coherence and an architectures direct support for SMC affect the implementation scope. To support our investigation, we have implemented and analyzed four different call-site schemes using a microbenchmark on three diverse ARMv8 processors. We have further evaluated three of those schemes using three well known JVM benchmark suites and the open source MaxineVM.

Our analysis has shown a variation of up to 12% in performance between the different strategies and highlighted different trends between different implementations

of the ARMv8 architecture. In particular, for advanced CPUs such as the Neoverse N1, with sophisticated branch prediction resources the difference in performance between the alternate schemes is small. Given the optimal `Direct` call scheme has a limited range, this highlights the virtue of optimizing code for size, and attempt to keep the code-cache size within the range of single branch of the ISA. Where this is not possible, we have shown that the `Trampoline` scheme decays to a direct call for managed runtimes that employ a partitioned code cache, where optimized code resides in a partition that can be spanned by a single branch instruction of the target ISA. From this observation we propose that for other architectures with fixed size ISAs and limited range procedure call instructions there is potential scope for for exploring optimizations for managed runtimes in terms of code cache organization. Specifically, by employing a `Trampoline` style procedure call scheme and partitioning the code cache such that code from the hottest paths with the highest level of compiler optimizations is co-located in an appropriately sized partition for the target ISA.

2.9 Related Work

The majority of prior studies have focused on detecting [DDZ18], modelling [AMDB07a], analysing [AMDB07b], and verifying [CSV07, Myr10] SMC. SMC is of particular interest in the area of security where it can be employed to either improve the security of systems [VCMKS12, AMN+11, HBLF13], or to obfuscate code in order to trick malware detectors [PVA+14, DCN+19] or to prevent reverse-engineering [MKP11]. In managed languages, SMC occurs as a consequence of tiered JIT compilation and runtime patching of call-sites to re-steer execution between the different compiled versions. Call-site implementations and the implications for replacing old methods with newly compiled ones, have not been thoroughly studied mainly because the dominant execution platforms of managed languages provide strong hardware support for SMC. Hence, implementations of call-sites were resulting in memory coherent results completely transparently to the users.

In architectures without hardware support for SMC, such as the AArch64, although studies have been conducted analysing their performance on various workloads [LTZ+15, RVV+13, TT13], to the best of our knowledge, no prior work exists on characterising the performance implications of SMC in the context of MREs. Our work aims at providing the first in-depth characterisation of alternative memory-safe call-site implementations.

Code-cache partitioning has been implemented in OpenJDK from version 9 onwards [HNG14], locating profiled, non-profiled and non-method code in three separate partitions. The motivation for this work was to improve iTLB and instruction cache performance, and reduce GC overheads.

Chapter 3

Managed Runtime Performance Understanding based on eBPF

3.1 Abstract

The *extended Berkeley Packet Filter* (eBPF) is a Linux subsystem that allows for flexible and dynamic full-stack instrumentation at user, library, and kernel levels without the need to change and recompile source code, and without changing kernel modules. We present *BPF-xVM*, an eBPF-based collection of tools that can be used to analyse the performance and execution behaviour of managed language runtimes. In this paper, we focus on the OpenJDK11 Java Virtual Machine with concurrent garbage collectors using G1 and Shenandoah with the Renaissance benchmarks. Our techniques are general and can also be applied to non-managed runtime executables.

BPF-xVM can capture, for the first time at operating system scheduling quantum granularity, the performance counters necessary to calculate top-down microarchitectural performance analysis that can help to identify processor bound performance bottlenecks attributed to the execution of threads. Offline python scripted analyses calculate and visualize statistical change-point, and timeline based views of top-down and execution time for display within the Chrome Profiler.

The main contributions of this paper are the production of a tool that can measure top-down performance of individual threads within a process at operating system scheduling granularity, with the ability to align these measurements against traced events. We demonstrate for the first time, the different dynamic and aggregate microarchitectural top-down behaviours of compiler, garbage collection and application threads in OpenJDK. We present an investigation into determining the steady-state

behaviour of applications that goes beyond using execution time, to include per-thread performance counter measurements. The geomean overhead of our tool for the Renaissance benchmarks using warmed up execution and a fixed processor frequency on a 4-core Intel Xeon workstation and using 4-cores on an AArch64 Ampere server is less than 2%.

3.2 Introduction

In this work we introduce a set of eBPF and python based tools, *BPFxVM*, that provide accurate performance analysis for managed language runtimes such as Java, Javascript and Python. We provide for the first time, a mechanism to identify which threads are inefficiently using a processor's microarchitecture at operating system thread scheduling quantum granularity. Top-down microarchitecture analysis is a well established methodology for identifying potential issues in modern processors supporting out-of-order execution, but current tools such as `Perf` and `toplev` are typically only able to perform such analysis and measurement over timed intervals, at full system, process, and processor core granularity. Such analysis is complicated by the very nature of managed runtimes where multithreaded execution encompasses not only the application, but also virtual machine *(VM)* services associated with just-in-time*(JIT)* compilation, and garbage collection *(GC)*. In this work, we demonstrate the use of offline statistical analysis and Chrome profiler visualizations of trace files that capture dynamic variation in top-down behaviour across all VM and application threads for the Renaissance benchmarks executed with OpenJDK11. Offline statistical analysis is focused on determining which data should be used to identify program execution phases that indicate if steady-state execution behaviour is achieved.

The research contributions presented in this work are:–

- RC1 the production of an eBPF tool to trace performance counter measurements for top-down analysis at thread-level for both X86 and ARMv8 instruction set architectures. We believe that we provide the first tool that can perform such fine grained measurements. This provides greater accuracy, and fidelity for a single application than is possible using measurements with perf and toplev from pmu-tools that perform process, core or system wide measurements. The tool will work for any executable, but it does not currently follow or trace threads that are created following additional process address space forks. We provide a set of python scripts to analyze and visualize the logs of performance counter traces,

making it possible to distinguish between different application and VM threads, whilst providing aggregated statistical and dynamic time-line views of logged information and top-down behaviors using the Chrome Profiler trace format.

- RC2 the ability to correlate and align top-down performance of threads with traced events from the OS, managed runtime, and application threads. Function hooking can record the event/operation type, argument values, start/stop time, and thread-id performing the event. This enables performance counter changes to be attributed to the traced events under investigation. For example, we provide functionality to trace DVFS changes that can be aligned with our top-down measurements. However, correlation and alignment is especially important for managed runtimes such as Node.js where the virtual machine threads do not have specific roles, and they are merely named *node*, in contrast to many Java virtual machines where thread roles are clearly identifiable. For runtimes such as Node.js, it is important to be able correlate measurements that identify the start/stop of managed runtime activities such as GC, deoptimization, and compilation against the top-down information. Node.js currently provides tracepoint support for instrumenting activities such as GC start/finish, and http client/server requests and responses.

- RC3 We investigate how to determine if an application demonstrates steady-state behaviour, rather than steady-state execution time [BBTK+17] for its performance. We apply statistical change point analysis to identify program execution phases and regions of interest for an application. We consider using i), univariate signals for changepoint analysis across applications iterations, firstly with wall clock execution time measurement, and secondly with the accumulated retired micro-operations attributed to all threads, and ii), multivariate signals where the accumulated retired micro-operations of threads are associated with specific thread roles such as application, compiler and GC. We use this to demonstrate the ability to find program phases that are significant for the behaviour of an application, and also for the behaviour of specific services within an application. We argue that using these techniques provides a better measure of whether an application has reached a steady state, than using measured execution time that is a function of the behaviour of non-deterministic out-of-order execution from the operating system and whatever other applications are running.

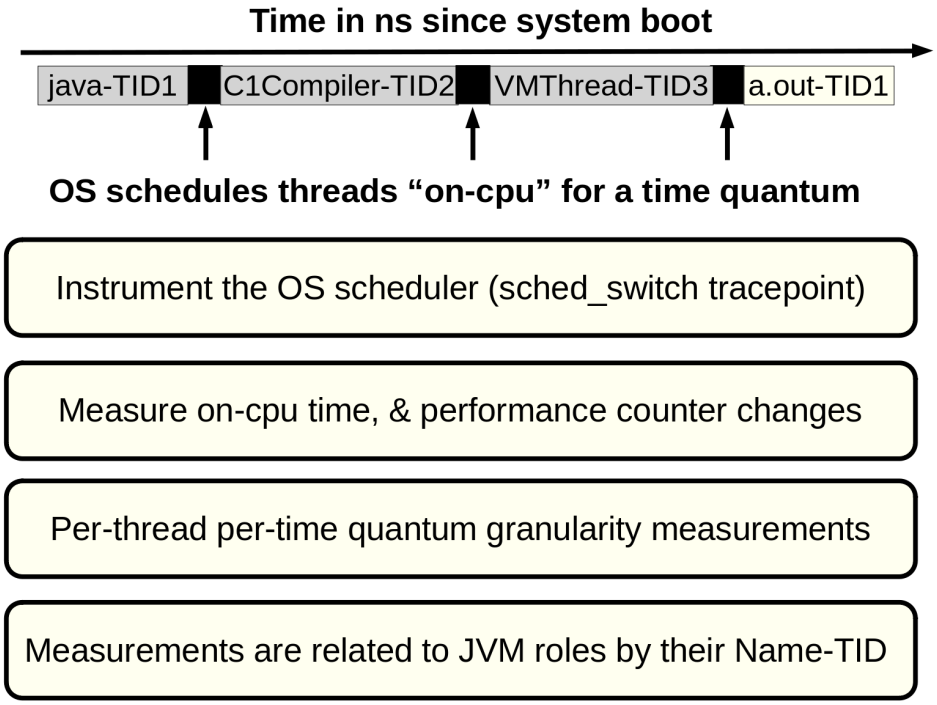

Figure 3.1: BPFxVM instrumentation tooling concepts. A Java process PID can be identified by scanning the filesystem for /proc/PID/java, or be passed as a command line argument to filter the process to be investigated. The unique 64-bit id of a thread is composed of the PID (upper 32 bits) and a Thread-ID TID in the lower 32 bits. Fine granularity per-thread, per OS scheduling quantum measurements are related to individual application and JVM service threads beyond what is possible with Perf and toplev. The Linux kernel maintains a 15 character *comm* string for each and every thread that we exploit to infer the roles of JVM service and application threads.

Figure 3.1 presents the high-level instrumentation concepts that provide the basic foundations for the tools and analysis presented in this work. Fundamentally the ability to record, for a specific process of interest, information such as the time instants when a thread executes *on-cpu*, and the accumulated changes in a set of performance counters for each OS scheduling quantum. This information can be stored along with the thread name and its unique Thread ID *(TID)* making it feasible to relate measurements to different threads that have specific roles with the Java Virtual Machine, for example those related to application processing, the main Java thread, JIT compilation, GC, and VM housekeeping. Additionally, it is feasible to record when a thread transitions between runnable and sleeping states, and to align measurements in time across different cores using the system time since boot. This makes it feasible to observe not only which threads are executing on-cpu and when, but also the thread sleep/wake dependences that have occurred. Here, we focus on initial attempts to exploit these fundamental instrumentation concepts and capabilities for problems and use-cases related to performance analysis and understanding for OpenJDK11 using the Renaissance benchmark suite [PRL+19a], and selected benchmarks from the Computer Language Benchmark Game [Gou04] for our experiments relating to steady-state behavior. Section 3.3 discusses related prior work concerning profiling and tracing tools, program phase detection in native applications, and also in managed runtimes with virtual machines. Section 3.4 discusses the concepts behind the top-down microarchitectural performance analysis methodology. Section 3.5 introduces the extended Berkeley Packet Filter and the Berkeley Compiler Collection that allow easier development with python frontends to eBPF. Section 3.6 describes the design and capabilities of the BPFxVM tool. Section 3.7 presents the experiments. Section 3.8 presents conclusions and opportunities for future work.

3.3 Background and Motivation

Here we describe related work concerning program execution phase detection, and the analysis of virtual machine performance for managed runtime systems that has motivated this work. Table 3.1 present a summary of the main features of selected related work that we describe in this section. It is important to understand the difference between counting or tracing and sampling based analysis. Counting/tracing based analysis accurately records the changes in a quantity such as a performance monitoring unit (PMU) register assigned to count an observable effect such as the total L3 cache

Tool Name	Target Functionality	PMU counters	Overheads/granularity
POP	Online execution phase analysis	Yes (tracing)	1.35% @10M instruction window
ScarPhase	Online execution phase analysis	Yes (tracing)	16.12% @10M instruction window
[BBTK+17]	Offline warmup analysis via PELT changepoints	No	Minimal (highly constrained target machine used for experiments)
Java Flight Recorder	JVM event based instrumentation	No	Typically < 2% (events < 20ms not recorded)
SHIM	Jikes/JVM instrumentation	Yes (sampling)	1.02x @1213cycle sample period
Perf/toplev	Performance + Topdown analysis	Yes (tracing)	System/core/process per-core 100ms granularity 2.5%, 10ms 7%
BPFxVM	Performance analysis Full-stack event based instrumentation	Yes (tracing)	OS thread quantum granularity (Renaissance geomean all benchmarks < 3%, max for single benchmark < 15%)

Table 3.1: Selected related background work summary of capabilities and overheads.

misses, whereas sampling seeks to statistically relate periodic changes in a quantity to a specific thread or a call-stack. Sampling can suffer from many issues due to imprecision in determining the actual PC (for example due to skid and shadowing effects) related to a change in a measured quantity, and selection/sampling bias.

3.3.1 Managed Runtime Performance Understanding

[EDE12, SBEE17] introduced techniques to analyze the scalability of managed language applications using speedup stack based visualizations. Speedup stacks are comprehensive bar graphs that break down an application's execution to explain the main causes of sublinear speedup, where the overheads associated with parallel execution do not allow perfect scaling, for example due to synchronization between threads for correct operation of application and VM services. [SBEE17] implemented OS kernel modules to monitor the application and service threads' scheduling behavior with less than 1% overhead, and without requiring modifications to Java code. Speedup stacks compare the achieved speedup of a multi-threaded application to ideal speedup and attribute the gap in performance due to a set of performance delimiters. The OS modules collected information concerning the number and IDs of active threads belonging to a JVM process. They accumulate the execution time, and five performance counters for the JikesRVM MainThread, garbage collection, and any application threads. Their experiments evaluated two garbage collection implementations: a concurrent, and a stop-the-world, garbage collector. During a stop-the-world collection application threads must yield the CPU and cease execution. The concurrent collector allows application and GC threads execution to proceed in parallel apart from brief stop-the-world pauses in application thread execution to identify a consistent root set, and later to actually free memory. The performance counters track L1/LLC loads and misses, and the count of retired instructions. Their approach did not relate time intervals in an

application's execution to the recorded measurements from kernel modules.

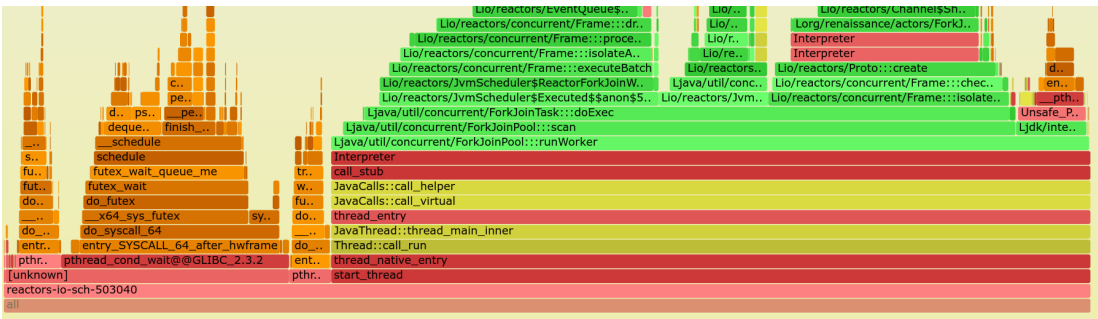

Figure 3.2: Renaissance reactors benchmark, zoomed in on-cpu flamegraph

Flamegraphs have been popularised as a means to visualise performance analysis data where it is appropriate to relate measured quantities to call stacks. Typically the call-stacks are ordered lexicographically, and the width of a specific call-stack is proportionally related to the measurement attributed or related to the specific call stack divided by the total measurements over all call-stacks. It is important to understand that the call-stack ordering in the flamegraph does not indicate a time-line. For a sampling profiler without any selection bias [MDHS10a], where call-stacks are measured repeatedly to determine which code is executing on-cpu, the width of each call-stack in the visualised flamegraph is expected to be proportional to the on-cpu time. Note that the on-cpu flamegraph cannot determine to what degree computation by different call-stacks is overlapped, or proceeding fully in parallel on different CPU cores, but it can determine where the majority of CPU time is spent. Figure 3.2 illustrates a zoomed in on-cpu flamegraph for a short snapshot of the Renaissance reactors benchmark. Colors are typically attributed to the call-stacks based on the contents of their text strings, and using any annotations that may have been provided by the profiler that was used to collect the stacks, for example to indicate if the stack refers to OS kernel code. For the flamegraph in this figure, where we used `perf` to collect the samples, we used red for *system* (includes library code), yellow for C++, green for Java and orange for OS kernel code.

[GNL18] described a performance analysis methodology based on flame-graphs for guest languages hosted on a JVM using the Truffle framework. They used the Linux `perf` tool and the JVM agent *perf-map-agent* and enhancements to the Graal JIT compiler to map `perf` sampled call-stack code addresses onto guest language source

code. In their work, they focused on evaluating the warmed-up steady state execution behavior of the Computer Language Benchmarks Game [Gou04] when using Sulong [RSM+18] that enables the execution of C/C++ programs compiled to LLVM-IR to be guest hosted on a JVM. Warmed-up execution expects that all hot-code has been JIT compiled and stable steady-state performance has been achieved. Non-JVM profilers such as `perf` suffer from the disadvantage that they must determine the mapping between JIT compiled methods and their code addresses, for example by using *perf-map-agent*, and also OS based stack walking for call-stack capture requires the frame pointer to be available which incurs some overhead (typically less than 3% [Gre]) when using -XX:+PreserveFramePointer. Flame-graphs were used to evaluate polyglot applications composed of multiple guest languages hosted by a single JVM, where the contributions of different guest languages to CPU time was highlighted using different colors.

[NNRL19] presents an overview into current profiling and tracing tools for identifying performance issues in Java applications. It reiterates the problems with safepoint bias [MDHS10b] in call-stack sampling present in some Java-based profiling tools, and illustrates the benefits of flamegraph based approaches for visualizing sampled call-stacks using Perf and async-profiler. Sampling techniques based on `perf` can see the entire stack whereas async-profiler fails to see certain JVM assembler stubs and compiler optimised intrinsics such as `System.arraycopy` due to OpenJDK issues with stack-walking such as [Pan, Maj]. Note however, async-profiler has the ability to identify which methods are interpreted and hence executing as unoptimized methods. An eBPF based profiling tool, *bcc-java*, was presented with similar capabilities to [SBEE17], but without the need to exploit custom kernel modules. The *bcc-java* tool accumulates PMU counter values for threads of a JVM process of interest, specifically: cycles, instructions and counts of calls to the *futex* system call over the lifetime of the thread, or the duration of profiling. Our BPFxVM tool, leverages the same eBPF/BCC infastructure as *bcc-java*, but is designed to capture the dynamic behaviour of individual threads, derived from changes in PMU counters during individual scheduling quanta, rather than cumulative aggregate behaviour.

3.3.2 Execution Phase/Behaviour Analysis

[CA20] contains a moderately recent survey on phase classification techniques for characterising variable application behaviour. One of the potential use-cases for online phase behaviour analysis is to enable system optimisation through the adaptation

of computation scheduling to better exploit heterogeneous processors, and/or to dynamically control the characteristics of system resources such as to optimise cache allocation technology [XWH+18], dynamic voltage and frequency scaling [HM21] or other parameters relating to compute, I/O and memory. Program execution behaviour is classified into application phases that specify the execution intervals delineated by time or the number of instructions executed where similar execution behaviour is exhibited. A phase typically represents stable execution or application characteristics. The problem of phase classification groups time or execution cycle based windows into intervals with similar characteristics to form phases.

The ScarPhase [SEH11] and the *Precise Online Phase (POP)* detector [TGB19] provide online techniques to perform phase classification for single-threaded applications. POP, like ScarPhase, uses a sampling performance counter approach using the Linux `perf` subsystem in a userspace program that builds signatures for the behaviour of the code executed over an instruction window. POP aims for its phase detection to remain invariant to any performance knobs that may be applied to adapt the system. Statistical and machine learning techniques are used to select a core set of eight performance counters that can be accurately used without any multiplexing on their experimental platform when the non-maskable interrupt is disabled. A lightweight online leader-follower approach is used to perform phase classification based on the signature of the counter values that are read after the sampling window of instructions has executed. Their paper claims an overhead of 1.35% and 0.09% at 10M and 100M instruction window sampling intervals for the single-threaded SpecSpeed 2017 benchmarks with a clustering similarity threshold of 20%, this is a 10x overhead reduction in comparison to ScarPhase that has overheads of 16.12% and 3.19% respectively. POP proposed a new metric *Statistical Time Analyzing Baseline (STAB)* to examine the balance between the number of phases, their stability, and the phase interval length. POP and ScarPhase were then evaluated using the STAB and a related prior-art metric concerning a corrected coefficient of variation described in [SEH11]. The POP detector performed well in comparison to ScarPhase across these metrics on their experiments.

Steady state warmup analysis described in [BBTK+17] presents a measurement methodology and results that analyse the execution of applications across a variety of common virtual machines on three different machines. The paper's results strongly suggest that developers should not rely on the commonly held hypothesis that application execution performance/behaviour on VMs can be viewed as consisting of

a warmup phase that determines which parts of a program can benefit most from dynamic compilation followed by a period of steady-state execution behaviour. In their experiments only 30-43.5% of their highly constrained physical machine and OS execution environment combinations used to evaluate their <VM,benchmark> pairs consistently reached a steady-state of peak performance. They deliberately chose to evaluate small deterministic (microbenchmark) programs under highly idealised settings where a heavily controlled execution environment was used to minimise measurement noise over 2000 in-process iterations and repeated across 30 fresh process executions. They applied changepoint analysis [EFK11] to automatically analyse shifts in the nature of time-series data concerning the execution times of iterations. Changepoint analysis was used to automatically classify if each process execution led to no steady state, flat (no detectable change in peformance over the benchmark's iterations), warmup, or slowdown (leading to a decrease in performance with increasing iterations or time). These simple classifications were used to categorise and highlight unexpected performance patterns. They additionally investigated the effect of GC and JIT compilation to see if these two factors could explain some of the effects highlighted by changepoint analysis. However, in their work the chosen benchmarks were small deterministic programs [Gou04] written in a number of different languages to facilitate investigation and comparison across multiple virtual machines and language implementations. In their changepoint analysis they elected to use the PELT algorithm [KFE12] to reduce the computational complexity of finding an unknown number of changepoints in their measurements of iteration execution time for 2000 iterations. The R `changepoint` package was used with the `cpt.meanvar` function with a minimum size of interval set to be 2 adjacent time-series measurements, a penalty argument that needed to exceed 15logn (where n is the time-series length of 2000 minus iteration executions that were excluded for being outliers). In section 3.7.3 we discuss issues concerning the changepoint analysis deployed in our experimental evaluation that goes beyond using wall clock execution time.

3.3.3 Instrumentation Approaches

SHIM [YBM15] is a continuous profiler that samples performance counters and memory locations that store software state at fine-grained cycle resolutions. SHIM has geomean execution time overheads of approximately 1.60x and 1.43x respectively at sample periods of 15 and 1505 cycles when observing method and loop identifiers using an observer thread executing on an unutilised core with symmetric multithreading;

and for remote observer threads in a chip multiprocessor, 2.1x and 1.02x overheads for 30 cycle and 1213 cycle sample periods. Time-varying software and hardware events are regarded as signals that are to be sampled by SHIM. The events are configured using a compiler or other tools and the mechanisms for reading/sampling are then communicated to SHIM. Offline and online analysis is used to address threats to measurement fidelity, such as from skew in the measurement of rate based metrics, observer effects related to the Heisenberg uncertainty principle, and low sample rates. The motivation for the development of SHIM was to identify program execution hotspots (such as frequently executed methods in Java applications), revealing high frequency behaviours of metrics such as instructions per cycle. SHIM is similar to our work in that they observe software and hardware events for a managed Java runtime. In their case, it is the Jikes RVM, and the period of sampling or tracing exhibits large variations in order to remove potential bias from measurements. SHIM inserts extra yield points into JVM code in order to identify methods and loops as a fine-grain software signal. The number of instructions between any two yield points is application dependent. Yield points are typically used by JVMs to allow for the coordination of threads for activities such as garbage collection and locking. BPFxVM tools do not require any modifications to the managed runtime or the compiler to perform their instrumentation other than the inclusion of debug symbols to allow for easy function hooking. This means that SHIM requires modification to the original JVM whereas our technique does not suffer from this restriction. SHIM attempted to show the diagnostic power of fine-grained program observations and comparisons to average results. JikesRVM is now a relatively old JVM technology compared to the features present in OpenJDK, consequently it is not clear if the techniques and the results provided by JikesRVM are representative of the behaviour of modern JIT compilers and GC algorithms. The current JikesRVM released version does not support Java 8, and the last issue was opened in Jun14th 2020 according to [jik].

3.3.4 Discussion Concerning Background Work

Although SHIM has low-overhead it is important to note that its low-overhead and fine-granularity (15 cycles) is for one software signal, and SHIM requires modification of, and is explicitly designed for the JikesRVM. BPFxVM in contrast, is designed to be independent from the JVM as far as possible, although it is necessary to have access to debug symbols in the libjvm.so library to be able to perform function hooking of key VM events. For example, in OpenDJK11 it is possible to trace deoptimization

events that can then be correlated and aligned with top-down information by function hooking symbols in libjvm.so. The ability to use separate tools to extend our tracing is a key aspect of our system that makes it extensible and adaptable to the performance understanding needs of an application. The overheads reported in figures 3.11 and 3.13 relate to BPFxVM when configured to record a total of 5 performance counters, and in addition it records the duration, quantum end time, cpu core and the thread name plus its associated numeric thread-id for each and every OS scheduling quantum belonging to the process under scrutiny. In BPFxVM, the JVM, or VM dependency relates to defining the events of interest to be traced and hooked.

In the context of our work we have sought to develop tools to perform offline phase classification for complex multithreaded workloads. We aim to perform phase classification to identify areas where managed runtimes can benefit from optimization, and also to begin to better understand the impact of different garbage collection algorithms on performance more deeply than just execution time, using an approach that requires minimal or no modification to source code. In this way we seek to avoid perturbing the execution behavior that we seek to measure. Such problems can occur when bytecode based instrumentation is applied, and JIT compilation optimization decisions may be altered [ZBB15], for example concerning method inlining, that can result in large performance variations from the uninstrumented code. We seek to marry approaches to program phase classification and warmup-analysis in order to understand virtual machine performance under different GC algorithms. In contrast to the work of [BBTK+17] and more closely related to the work of ScarPhase and POP we seek to measure the execution behaviours of threads, rather than just application execution time to determine if behaviour has reached a steady state. In particular we focus our analysis on the well-known top-down approach [Yas14] that attributes the percentage of execution cycles utilised or spent to different aspects of processor microarchitecture.

The benchmarks we have chosen, and the machine configuration on which we run experiments are more closely related to a normal execution environment than in [BBTK+17], as in our experiments we do not use clean boots of the OS in between execution time measurements.

3.4 Top-Down Performance Analysis

Top-down performance analysis as originally presented in [Yas14], and depicted in figure 3.3 helps to analyse the relative overheads of different microarchitectural bottlenecks in an abstract model of *out-of-order (OOO)* processors. Traditional methods based on the calculation of stall overheads associated with a specific event are not suitable for OOO processors, because stalls can overlap, speculative execution occurs, and it is difficult to predict the instruction issue slot utilization associated with superscalar execution. Abstracted top-down metrics provide a way to standardize performance analysis across different microarchitectures and PMU capabilities. Potentially, even to analyse how well a given application has been optimised for two different architectures. We do not seek to compare or evaluate the relative merits of X86 and AArch64 microarchitectures, and this is not a goal of our work. The top level of the abstracted OOO architecture model consists of:–

- The *Frontend bound* category: this refers to issues when the frontend of the CPU under supplies its backend with uops for execution.
- The *Bad speculation* category: this reflects the case where micro-operation *µop* issue slots are wasted due to incorrect speculation, including slots issued to µops that do not eventually retire, as well as slots in which the issue pipeline was blocked due to recovery from earlier mis-speculations. Bad speculation determines the fraction of the workload under analysis that is affected by a processor following incorrect execution paths.
- The *Retiring* category: this reflects the percentage of µop slots, utilized by µops that eventually get retired and complete useful work. However, a high retiring value can signify that there are issues associated with microcode sequencing that merit further investigation (such as for non-vectorized code).
- The *Backend bound* category: this reflects the percentage of issue slots wasted due to no µops being delivered at the issue pipeline due to a lack of required resources for acceptance by the backend which is further subdivided into level 2 categories of *memory bound* and *core bound*. The Memory bound category computes the percentage of execution stalls related to the memory subsystem. The Core bound category captures issues representing the usage of the core's functional units. Consequently, it can highlight issues that can often be mitigated by improved code generation, for example by the usage of vectorization,

and/or relieving the dependencies between arithmetic operations with long execution latency (for example divide operations) via the exploitation of improved instruction scheduling.

The top-down method uses designated performance counters in a structured hierarchical approach to quickly identify the dominant performance bottlenecks by estimating the percentage of time/cycles attributed to a specific abstract overhead. The method has been adopted by multiple in-production and research usage tools for Intel such as VTune, pmu-tools/toplev [Kle22] and for ARM architectures using Forge (formerly Allinea DDT/MAP). Processor specific metrics/formulas and performance monitoring counters are used to determine if an abstracted overhead is significant.[1] The specific metrics, and their thresholds are provided for Intel architectures within the pmu-tools repository and in the online spreadsheet [Yas]. The top-down approach seeks to cate-

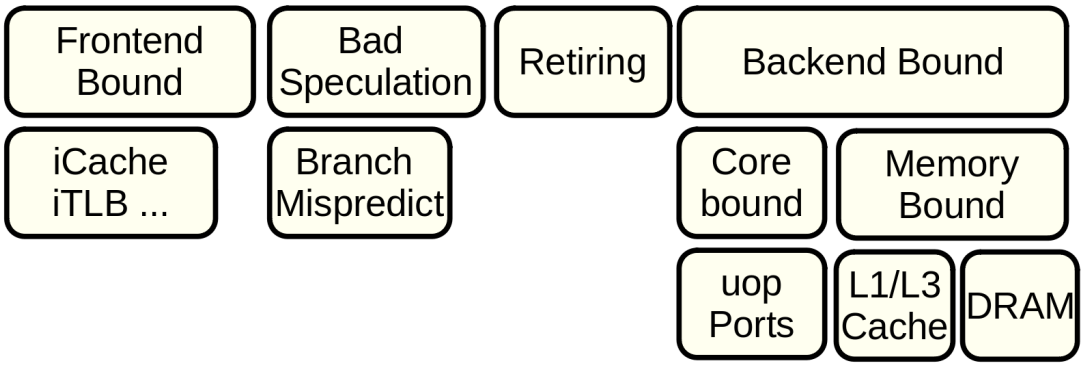

Figure 3.3: Simplified top-down analysis hierarchy for X86 (modified image based on [Yas14]).The top-down classification of an overhead can be refined and mapped onto poor utilisation of specific functional in the microarchitecture. For example, a backend bound computation is typically limited by some combination of the memory system (data cache and main memory bandwidth/latency) and the core itself, such as concerning the utilisation of µop execution ports. Normally the goal is to maximise the percentage of cycles related to retiring µops as this typically indicates good utilisation of the microarchitecture.

gorise CPU execution time at the highest level first, consisting of frontend bound, bad speculation, retiring and backend bound. If the fraction of execution time exceeds a microarchitecture specific threshold, then investigation at lower and more detailed levels is flagged. As one moves down the hierarchy, the overhead classification becomes more directly related to the specific microarchitecture on which measurements were

[1] Closely related work in [AR19] seeks to provide a microbenchmark based methodology to characterize the latency, throughput, and port usage of instructions on all Intel Core microarchitectures.

made. For example, to determine if the causes of a high percentage of backend bound cycles are mostly attributed to poor utilisation of the memory subsystem, or to µop execution port utilisation. A specific class of metrics in the hierarchy should only be investigated if its threshold and all its parent thresholds (at higher levels in the hierarchy) are flagged as exceeded. Further, only the metric values of sibling nodes (children of the same parent node) in the hierarchy should be considered to be directly comparable. This means that it is not appropriate to directly compare the percentages produced by `uops Ports` and `DRAM` as they do not share the same parent node, but it is appropriate to compare `L1/L3 Cache` and `DRAM`. In this work we restrict top-down analysis to the four top-level metric classifications using the metrics depicted in Tables 3.2 and 3.3 for Intel and ARMv8 architectures respectively.

Metric Name	**Formula**
SlotsIssued	uops_issued.any
SlotsRetired	uops_retired.retire_slots
RecoveryBubbles	4 * int_misc.recovery_cycles
FetchBubbles	idq_uops_not_delivered.core
TotalSlots	(4 * cpu_clk_unhalted.thread)
FrontendBound	FetchBubbles / TotalSlots
BadSpeculation	(SlotsIssued - SlotsRetired + RecoveryBubbles) / TotalSlots
Retiring	SlotsRetired/TotalSlots
BackendBound	1 - (FrontendBound + BadSpeculation + Retiring)

Table 3.2: Level 1 Top-down metrics Intel (hyper-threading disabled) – 5 PMU counters required

Metric Name	**Formula**
FrontendBound	stall_frontend / cpu_cycles
Allocated	(cpu_cycles - (stalled_frontend + stalled_backend)) / cpu_cycles
RetiringBound	(inst_retired/inst_spec) * Allocated
BackendBound	stall_backend / cpu_cycles
BadSpeculation	1-(FrontEndBound+BackEndBound+FrontEndBound)

Table 3.3: Level 1 top-down metrics (from [arm]) for Armv8-A architecture – 5 PMU counters required. Note that stall_frontend and stall_backend require at least Armv8.1 PMU.

3.5 eBPF and BCC

The extended Berkeley Packet Filter (eBPF) is a feature that was first added to the Linux kernel version 3.15 (around 2014). It provides an instruction set and an execution environment for modification, and interaction with user and kernel programmability at runtime. eBPF was originally developed to assist the development of efficient network processing code, this has resulted in eBPF providing programming support for the eXpress Data Path (XDP), a kernel network layer that enables network packets to be processed closer to network interface chips/cards. However, since the introduction of eBPF in 2014 it has been adopted by companies such as Facebook, Cloudflare, Netflix, Sysdig, and Isovalent/Cilium. eBPF use-cases have subsequently extended to include network/security monitoring, network traffic manipulation, load-balancing and application/system profiling and performance analysis. In figure 3.4 we illustrate the main principles of operation and interactions between the *BPF Compiler Collection (BCC)* and eBPF that we have used to support implementation of BPVxVM. It is important to note that other front-ends and tools are provided for eBPF beyond BCC, and in this work we do not advocate or discuss their relative merits; more information can be found at the home pages of the IOVisor Project[2], eBPF[3] and associated github repositories[4]. BCC provides a set of tools and examples for creating efficient kernel tracing and manipulation programs using a combination of Python and C. Python is used to simplify compilation, loading, and IO-based interactions with eBPF programs that are written in a subset of C and JIT compiled using LLVM. Before an eBPF program can be executed, it must be compiled to BPF bytecode, and then verified to ensure that its operation is safe, meaning that the program will not crash the OS, such as by performing an invalid access to memory. The bytecode verification performs static code analysis and multiple passes are used to ensure that the program is a directed acyclic graph, and it rejects any programs where the total instructions are larger than a set maximum, have unreachable instructions, out of bounds or malformed jumps, or loops present. Once verified, a program can be loaded and executed by the eBPF virtual machine. eBPF uses an event driven programming model, and data can be shared between instrumentation programs and userspace front-ends to eBPF via BPF maps and arrays that provide associative data structures. One of the strong advantages of eBPF

[2] `https://www.iovisor.org`
[3] `eBPF.io`
[4] `https://github.com/iovisor`

over `perf`, is its ability to selectively perform processing and aggregation of information directly at the point of instrumentation, without the need to dump all information to file. However, eBPF has strong support for the `perf` events subsystem in Linux, consequently it is possible to read performance counters, and to add instrumentation to method entry/return and even to arbitrary code addresses using function hooking implemented with software breakpoints (probes) and using jumps (tracepoints). Tracepoints are advertised stable instrumentation points within application, library and kernel code. The ability to efficiently attach code to tracepoints, and to more arbitrary code locations with probes gives eBPF the capability to be a general purpose tracing tool. Function call arguments can be introspected and traced, and stack traces can be sampled or collected under eBPF program control. Tail-calls are provided to allow a eBPF program to jump into another - this provides a work-around mechanism to avoid issues related to maximum instruction limits, although this was not necessary for our tooling.

eBPF maps enable programs to gather and store information to share with other eBPF programs, and also any userspace programs (for example the Python program used by BCC to compile, load, and execute an eBPF program written in a subset of C) that are granted access to the map data. In the context of our tool we use eBPF hash maps, to store per-thread data concerning the elapsed time and changes in performance counters attributed to individual threads, using the thread identification from the process-id as a hash key in a BPF MAP. Special per-cpu arrays are used to read and access performance counters. After loading an eBPF program, the Python program must initialize each per-cpu array to be associated with a specific PMU counter event. Data can be read from userspace, or pushed to userspace using a variety of mechanisms, but since kernel version 5.8 the preferred mechanism for pushing event data is to use eBPF's ring buffer. The ringbuffer is designed to preserve the ordering of events that happen sequentially in time, even across multiple CPUs and has lower latency than using the standard perf output buffer.

3.6 The Design of BPF-xVM

In order to measure and attribute the changes in performance counters to threads at OS CPU scheduling quantum granularity, we must first identify the process-id of interest to be traced. We follow the mechanisms used in the BCC tool `ustat` that scans the `/proc` directories such as `/proc/2000/java` to determine that a Java process-id of 2000 is

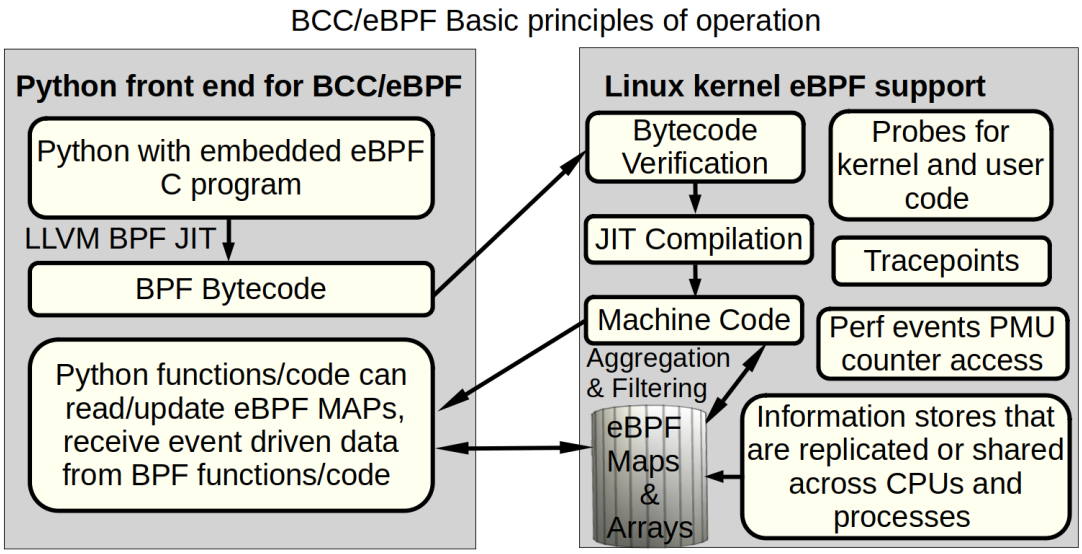

Figure 3.4: (BPF Compiler Collection) for eBPF (extended Berkeley Packet Filter) basic principles of operations

in execution. Alternatively, we also enable a process-id command line argument to be used. An additional command line argument is used to determine how many OS scheduling quanta must elapse for a thread, or if a signal must be delivered before information is pushed to userspace and logged using CSV formats such as: a timestamp (ns), thread name and id, duration (ns), CTR1, CTR2, ... CTRN, CPU.

Figure 3.5 outlines how BPFxVM exploits eBPF tracing capabilites using tracepoints and uprobes, and Figure 3.6 outlines the operation of the *sched_switch* tracepoint instrumentation. BPF_PERF_ARRAY variables for each CPU core are used to access PMU performance counters. The variables are initialised in python code using a call to open_perf_event that configures the counter arrays to be related to a specific PMU event. An `info` BPF_PERCPU_ARRAY variable is used to store the set of performance counters and time in ns since boot of the last *sched_switch* invocation on each CPU. In this way we are able to measure the change in performance counter values and time that is accumulated for each on-cpu quantum of a thread. Note that it would be unfeasible to record this information for each and every process and thread currently in execution. We currently limit what is traced to a single process, the TRACED_PID, that is identified by scanning `/proc` or by using a command line argument. Python string replacement of the TRACED_PID with its actual integer value is performed prior to loading the BCC/eBPF program. The `runtime` BPF_HASH is used to provide a single store of the accumulated information on a per-thread basis.

Monotonic clock `timestamps` enable measurements to be aligned in (ns) of system time since boot. The OS command string name of a thread has its OS thread-id appended, the duration of time that the thread was executing on-cpu (ns), and the accumulated counter values associated with the thread's execution. The cpu/core id that pushed the information is also recorded. For `N=1`, this means we can record each and every thread scheduling quantum, and the CPU that was used, whereas for `N=0` we accumulate information for each and every thread until the instrumentation tool receives a SIGUSR1 signal. On receipt of the signal, all accumulated information is outputted. The linux OS *sched_switch* tracepoint is instrumented to achieve this functionality. We include support for Linux kernel versions to use BPF_PERF_OUTPUT when the more efficient BPF_RINGBUF_OUTPUT is not supported. BPFxVM also instruments the *sched_process_exit* tracepoint ensure information logging for threads where the modulo division of their total scheduling quanta by N (quantum logging granularity) is non-zero. The key aspects of Figure 3.5 are the use of BPF maps and arrays to store information, and the use of a BPF_PERF/RINGBUF_OUTPUT maps to enable data to be pushed to user-space. A BPF_HASH map is used to store per-thread information hashed against the full process-id (including task and thread-id), concerning the accumulated counter values, time on CPU in ns, and the number of quanta. Per-cpu BPF_PERCPU_ARRAYs enable the elapsed on-cpu time, and accumulated changes to performance counter values to be determined from the previous operation of *sched_switch*. Table 3.4 outlines sample CSV file output measurements at per-thread, per OS scheduling quantum granularity and also for DVFS event tracing.

3.6.1 OpenJDK Tracing

The thread names used by OpenJDK11 can be used to identify JVM thread roles. The 15 character command string that is stored by the OS and visible to eBPF instrumentation can easily identify JIT and GC related threads; such as C1_Compiler_Thre, C2_Compiler_Thre whereas GC (G1) related threads include those named: GC_Thread#N. The main thread used to start the JVM is named as java and there are various application and other VM service threads created. We can also perform uprobe based function hooking of function call and return. This can be used to investigate the overhead attributed to specific JVM activity by tracing function activity using code based on the BCC `funclatency` tool. We used this approach to perform some initial investigations related to the Deoptimization class in libjvm.so, but we did not find any significant sources of such overhead in the Renaissance benchmarks. We use

a Renaissance Plugin to output the start/stop times of iterations using the monotonic clock. We have also investigated the use of a Plugin to send a SIGUSR1 signal to the python eBPF process at the end of each iteration, to output aggregated information for: thread quanta performance counters, Deoptimization function latency and any execution count information collected via uprobe instrumentation as outlined in figure 3.5. It is possible to obtain an indicator of total deoptimization latency and activity per iteration using this approach, although it is important to ensure that the Plugin pauses application processing for sufficient time to allow for signal delivery and any IO to complete. Note, that due to the operation of the eBPF techniques used to calculate function latency, we do not account for recursive execution (only leaf calls will be timed), and it is not possible to time a small number (4) of Deoptimization related methods because of issues related to the operation of uretprobes where the return address on the stack is modified to be a trampoline. This can lead to JVM crashes when stack manipulation of the return address is performed by deoptimization, and is related to issues reported in [fun]. We can accurately determine the changes in performance counter values attributed to threads on a per iteration, per quantum basis for the Renaissance benchmarks. We perform offline processing of the logged information using a set of python scripts that can load/analyse and visualise the data.

3.6.2 DVFS Tracing

A command line switch is used to determine if dvfs tracing is enabled, and the cpu_frequency tracepoint is instrumented as outlined in Figure 3.5. The arguments to a specific *TRACEPOINT* can be inspected by examining the appropriate *TRACEPOINT*`/format` entries contained under `/sys/kernel/debug/tracing/events/`. For the `power:cpu_frequency` tracepoint we can see that it takes two unsigned long arguments, that specify i), the new state (operating frequency) and ii), the given cpu. Note that DVFS tracing using this tracepoint is largely only appropriate if hardware support for, and the collaborative processor performance driver is not enabled.

3.6.3 Chrome Profiler Visualizations

Figure 3.7 depicts chrome profiler visualisation of a trace file from BPFxVM. It demonstrates that potential stop-the-world application pauses can be easily identified by determining the intervals where GC related thread names are active, yet no application

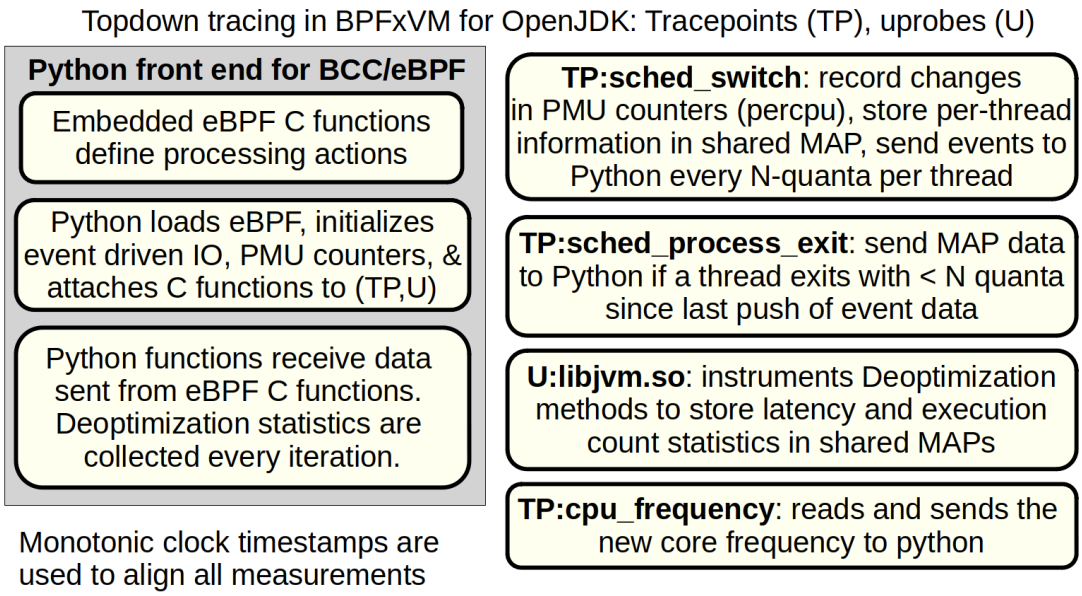

Figure 3.5: Outline of BPFxVM tracing exploitation capabilities for eBPF tracepoints and uprobes in Linux and OpenJDK. BPFxVM only collects event data for the traced process(es).

threads gain access to the available CPUs. Figure 3.8 shows a timeline view of top-down microarchitecture utilisation statistics for the Renaissance akka-uct benchmark.

3.7 Evaluation

In this section we describe the evaluation of BPFxVM. Firstly we outline the experimental environments, then quantify the overheads of tracing compared to untraced execution, followed by our early experiments using the tool to examine the performance of the OpenJDK JVM.

3.7.1 Experimental Environment

We use OpenJDK11 version 11.0.12 running on a Linux desktop machine running Ubuntu Focal Fossa 20.04.2 LTS with kernel 5.10.0-1052-oem and BCC release version v0.18.0 installed. The desktop machine comprised an Intel(R) W-2123 CPU with four physical cores (hyper threading, collaborative processor performance, and turbo-boost were disabled). A userspace governor was used to fixed core clock frequency to 3.3GHz, 32GB main memory, and all disk accesses for the execution of experiments were to a local hard disk drive. Our tool targets both Armv8-A and Intel architectures

```
BPF_PERF_ARRAY(ctr1, NUM_CPUS); // … ctr2, ctr3, ...
BPF_PERF_ARRAY(ctrN,NUM_CPUS); //

BPF_PERCPU_ARRAY(info, struct PerfInfo); // stores time, ctr values
BPF_HASH(runtime, u64, struct ThreadInfo); // per-thread information

RAW_TRACEPOINT_PROBE(sched_switch) {
  // read current time (ns)
  // read new counter values
  ppid = bpf_get_current_pid_tgid(); // THREAD-TID is ppid&0xffffffff
  if ((ppid >> 32)  == TRACED_PID) { // TRACED_PID = Java pid to be traced
      // lookup & update runtime BPF_HASH for thread on-cpu
      // optionally push information to userspace for output
  }
  // update info PERCPU_ARRAY for this CPU
}
```

Figure 3.6: BPFxVM *sched_switch* tracepoint shows a simplified high-level outline of tracing functionality and BPF structures exploited in BCC. Error handling and details of BPF_PERF_OUTPUT and BPF_RINGBUF_OUTPUT omitted for clarity. The set of specific PMU performance counters to be accessed via ctr1, ... , ctrN BPF_PERF_ARRAY variables, are configured using Python after instrumentation is loaded.

using command line switches. We also tested our tool on an 80 ARM core Ampere Altra running CentOS 8.0.1905 and Linux kernel 4.18.0-80 with BCC release version v0.16.0. The less advanced functionality of the 4.18 kernel means that the higher latency perf output buffer must be used instead of a ring buffer that is advocated for 5.8.x kernels and above.

3.7.1.1 The Renaissance Benchmarks

The Renaissance benchmark suite contains a range of modern workloads to investigate JVM behaviour and performance across a range of programming paradigms including concurrent, parallel, functional, and object-oriented programming. We use Renaissance version renaissance-gpl-0.12.0.jar with plugins to collection iteration execution times using a monotonic clock, and also to provide an option to send a SIGUSR1 signal to our tool. This enables the output of per-iteration summary information of events per thread, rather than as traced events occur. We exclusively use BPFxVM to dynamically trace the changes in hardware performance counters to specific threads. All of the Renaissance benchmarks are run within a single JVM process and only rely on the JDK as an external dependency. Some benchmarks use network communication

ENDTIME	THREADNAME-TID	DURATION	PMU C1	...	PMU C5	CPU
7390012629	C2_CompilerThre-13360	10604957			42819126	79
7760014633	java-13351	19795714			75677512	77
8269277193	C1_CompilerThre-13361	3841159			15517694	76
8590009772	java-13351	829981419			31194070	77
9410017915	java-13351	23474033			92790170	78
9450012383	java-13351	22363022			71143586	78
9590008502	Shenandoah_GC_T-13367	10763517			30651032	76
9694505740	Shenandoah_GC_T-13367	4452398			14766818	76
9703387476	C2_CompilerThre-13360	4199319			18660175	76
9792896669	VM_Thread-13355	86000			513896	79

TIMESTAMP	TEXT	CORE	FREQUENCY
17484411810185464	DVFS	76	1000000
17484411910181312	DVFS	77	1020000

Table 3.4: Sample CSV snippet from ARM Ampere system - timestamps simplified to fit pagewidth, taskset used to constrain usage to processor core 76-79. Topdown (multiple counters) and DVFS measurements outlined.

via multiple threads within a single process address space that exercise the network stack using the loopback interface, as described in [PRL+19a]. All garbage collection algorithms were run with `-Xms12G -Xmx12G`. We experimented with runs designed to investigate program execution phases - running for up to 400 iterations, and also timed duration runs: running for 600s, and timing the warmed-up execution for all complete iterations in the last 300s of execution.

3.7.1.2 OpenJDK GC Algorithms

OpenJDK11 provides two parallel and concurrent GC algorithms, the *Garbage First: (G1)* and *Shenandoah*. Both collectors are designed to achieve lower application thread pause times, whilst also maintaining a consistent application throughput, but at the cost of increased GC computation. Shenandoah's key advance over G1 is that it aims to do more of its GC work concurrently with application threads. G1 can only evacuate its heap regions (move objects) when the application is paused, whilst Shenandoah can relocate objects concurrently with the application. To achieve this concurrent relocation, Shenandoah uses what is known as a *Brook's* pointer. Essentially, this is an additional field that each object in the Shenadoah heap has, that points back to the object itself. When Shenandoah moves an object, it needs to fix up all the objects in the heap that have references to that object. On moving an object to a new location it leaves the old Brook's pointer in place, forwarding all references to the new location of the object. When an object is referenced, the application follows the forwarding pointer to

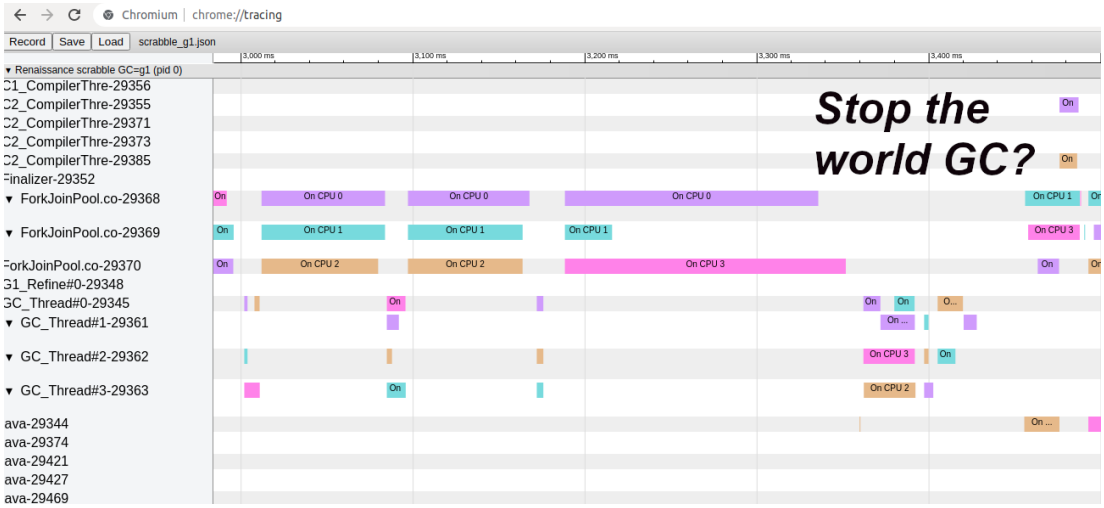

Figure 3.7: Visualising potential GC stop-the-world issues, using trace files. The chrome profiler can be used to visually determine time-intervals where only GC threads are running, potentially identifying stop-the-world-pauses where no application threads can run.

the new location. Eventually, the old object with the forwarding pointer needs to be cleaned up, but by decoupling the cleanup operation from the step of moving the object itself, Shenandoah can achieve greater levels of concurrency than G1.

3.7.2 Tracing Overheads

The relative slowdown of our tool when measuring topdown behaviours of each thread on a per quanta basis was evaluated on runs using a 12GByte heap, with a forced GC between each iteration, and where the iteration time measurements were taken from the last 300s of a 600s duration run. On the X86 machine we fixed the frequency to 3.3GHz and on the Ampere to 2GHz. We used four cores on each machine, and on the X86 we turned off cooperative power management, hyperthreading and turbo boost in the BIOS. The slowdown experiments did not instrument Deoptimization functions from libjvm.so or DVFS activity. The slowdown is calculated in comparison to the geomean of the measured iterations for the uninstrumented application benchmark. The *sched_switch* instrumentation restricted its attention to on-cpu using thread scheduling states that were identified as TASK_RUNNING. Figure 3.10 demonstrates the relative slowdown of our tool on the X86 machine for the Renaissance benchmarks using Shenandoah GC. Note that both reactors and db-shootout benchmarks exhibited

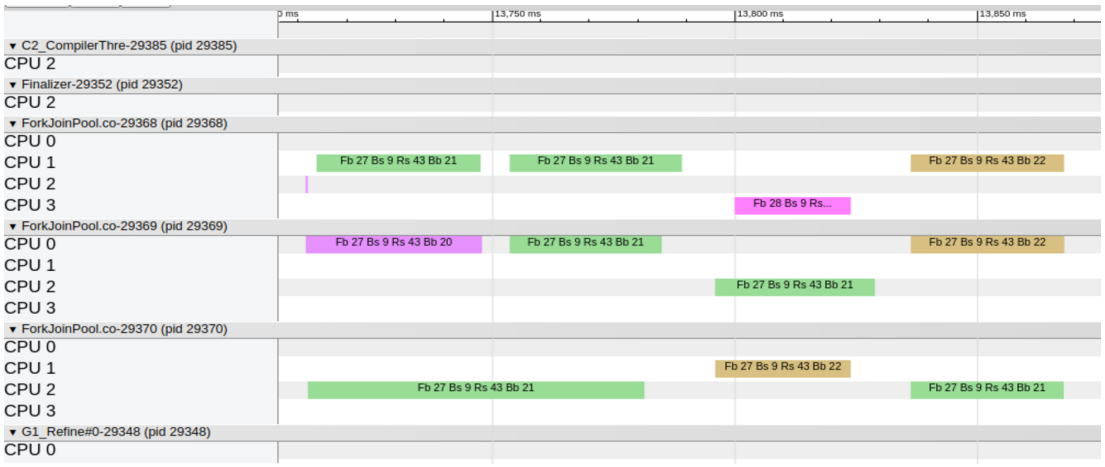

Figure 3.8: Timeline view of top-down microarchitecture utilisation statistics for the Renaissance akka-uct benchmark.

wide variations in their execution time leading to some relatively large geomean slowdowns being present in the violin plot representation of the distribution. Figure 3.13 demonstrates the relative slowdown of our tool for the Ampere machine with G1 GC. Geomean slowdowns for Renaissance benchmarks on X86 and ARM (Shenandoah & G1) are presented in figure 3.9.

The overhead analysis in figures 3.10, 3.11, 3.13 & 3.14 only compare the last 300s of complete warmed up execution iterations for a single timed duration run of 600s of BPFxVM instrumented and uninstrumented (normal execution). The results can therefore be viewed as indicative rather than as complete, as a rigorous analysis would run both versions multiple times to create a confidence interval for the mean. The ARM machine is running an earlier kernel, and an earlier release of the BCC/eBPF tooling framework. The results presented in the paper make use of a BPF_HASH map to update information about performance counter values that have been accumulated for each thread. Optimisation opportunities yet to be exploited include i), the use of BPF_PERCPU_HASH maps (not available on the ARM machine due to a lower kernel and BCC versions), that will remove contention for access to a BPF_HASH map that is shared between all cpus, ii), omitting storing information in BPF maps at all if data is pushed to user space every quantum, and iii), investigating optimisation of how to push information to user space.

We draw attention to several benchmarks that exhibit a speed up for traced execution compared to untraced, e.g. finagle-chirper, future-genetic, reactors and scala-stm-bench7 shown in figure 3.11 for X86 / G1 GC. These benchmarks show a significant

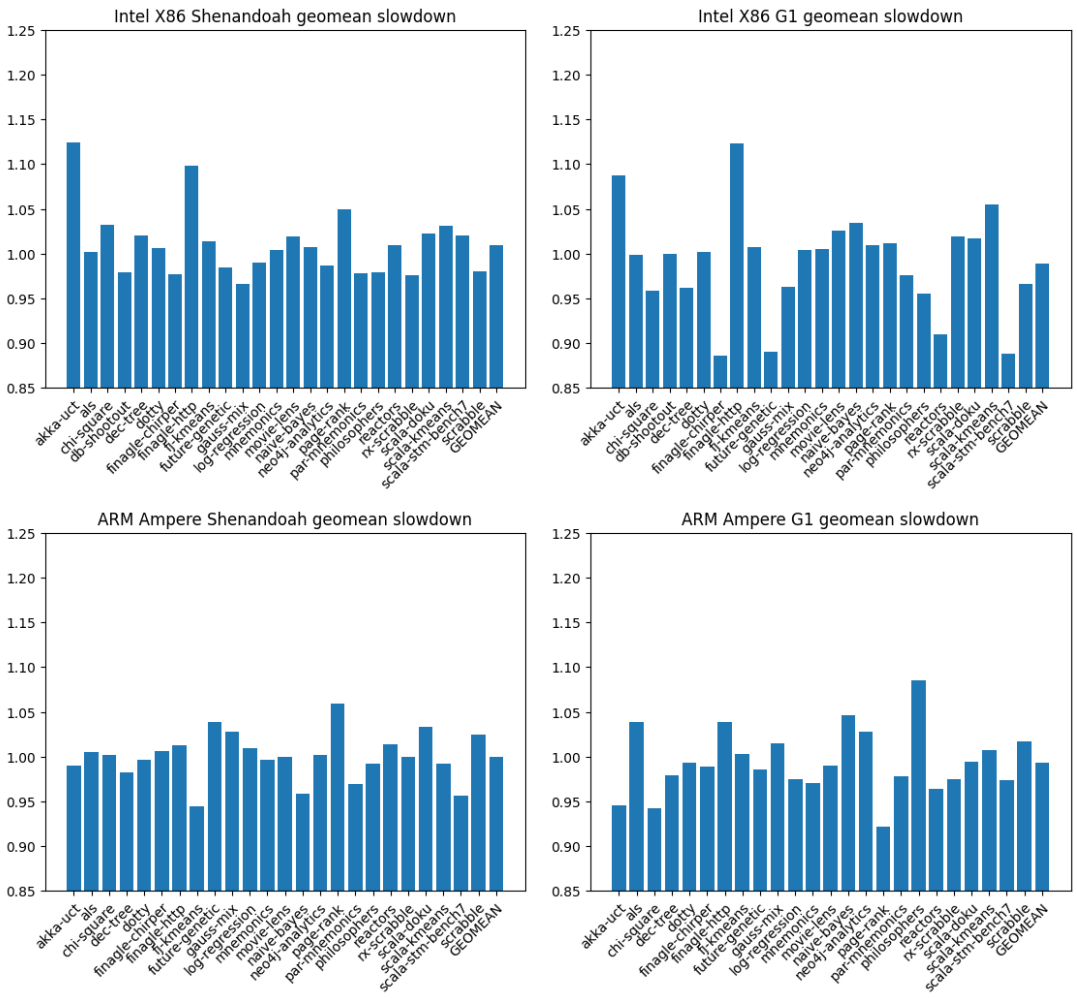

Figure 3.9: Effect of Tracing: slowdown on Intel using both G1 and Shenandoah GC for Renaissance benchmarks.

10% speedup when traced with well clustered results, except for reactors which has wide variation for all platforms/GC's. Our initial investigation suggests that these results are repeatable, however, as already mentioned further experiments are required in order to establish the statistical significance. Other similar speedups, although of lesser magnitude, can be seen in figure 3.9 which shows the slowdown effects for both platforms and GC algorithms. We intend to investigate these behaviours further and try to understand the causes behind these observations.

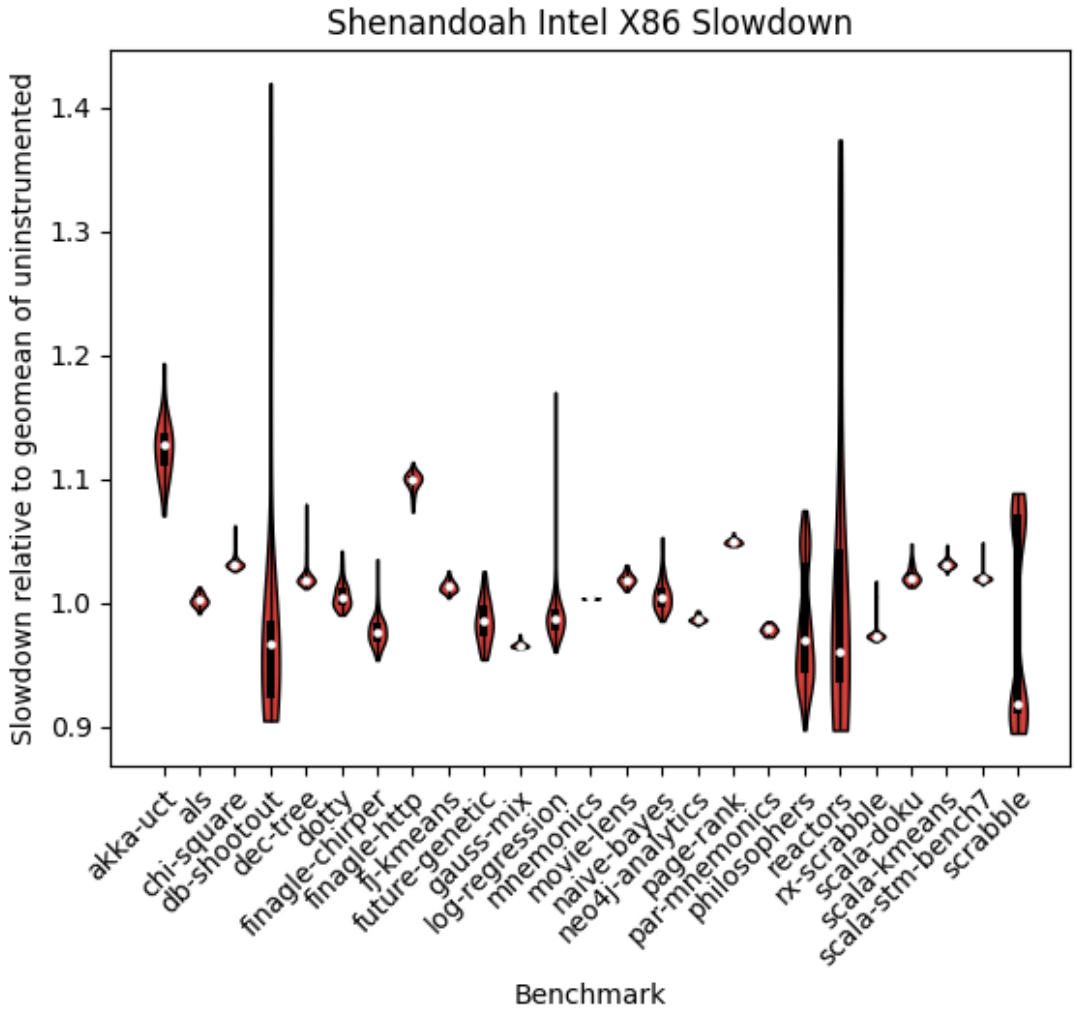

Figure 3.10: Customized violin plot [doc] of Renaissance benchmark slowdown using Shenandoah, calculated against the geomean of uninstrumented execution with Shenandoah for each benchmark. 600s duration run, X86 machine, measurements used from the last 300s, frequency fixed to 3.3GHz with a userspace governor, turbo boost and cooperative power management disabled. The 25,50 (clear circle) and 75 percentiles are displayed within each benchmark's violin plot. Mnemonics has very little variation in execution time or overhead, making it difficult to visualize the violin plot in the figure.

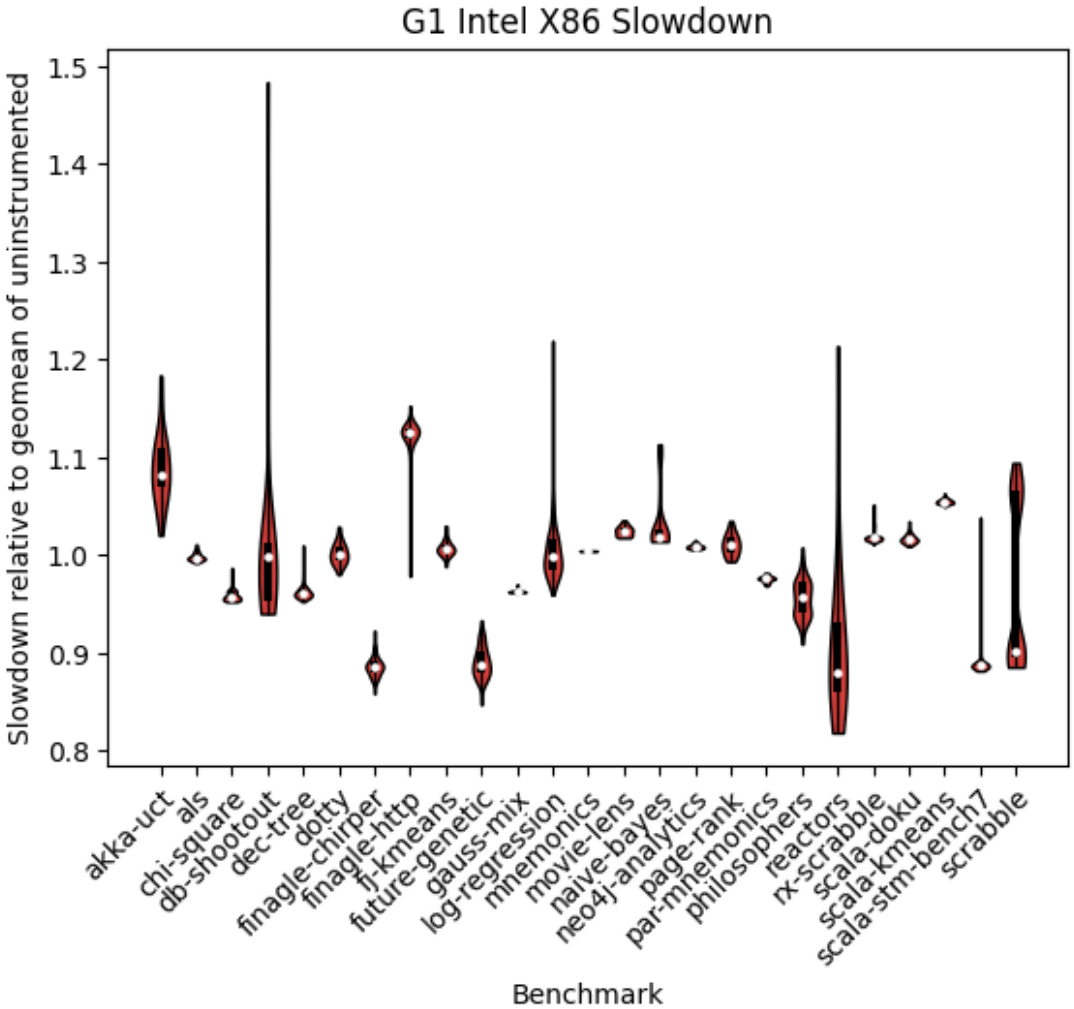

Figure 3.11: Customized violin plot [doc] of Renaissance benchmark slowdown using G1, calculated against the geomean of uninstrumented execution with G1 for each benchmark. 600s duration run, X86 machine, measurements used from the last 300s, frequency fixed to 3.3GHz with a userspace governor, turbo boost and cooperative power management disabled. The 25,50 (clear circle) and 75 percentiles are displayed within each benchmark's violin plot. Mnemonics has very little variation in execution time or overhead, making it difficult to visualize the violin plot in the figure.

Figure 3.12: Customized violin plot of finagle-http slowdowns using BPFxVM on an X86 machine. Violin plot markings as in figure 3.11. The geomean slowdowns are annotated on the figure to give an indication of overhead.

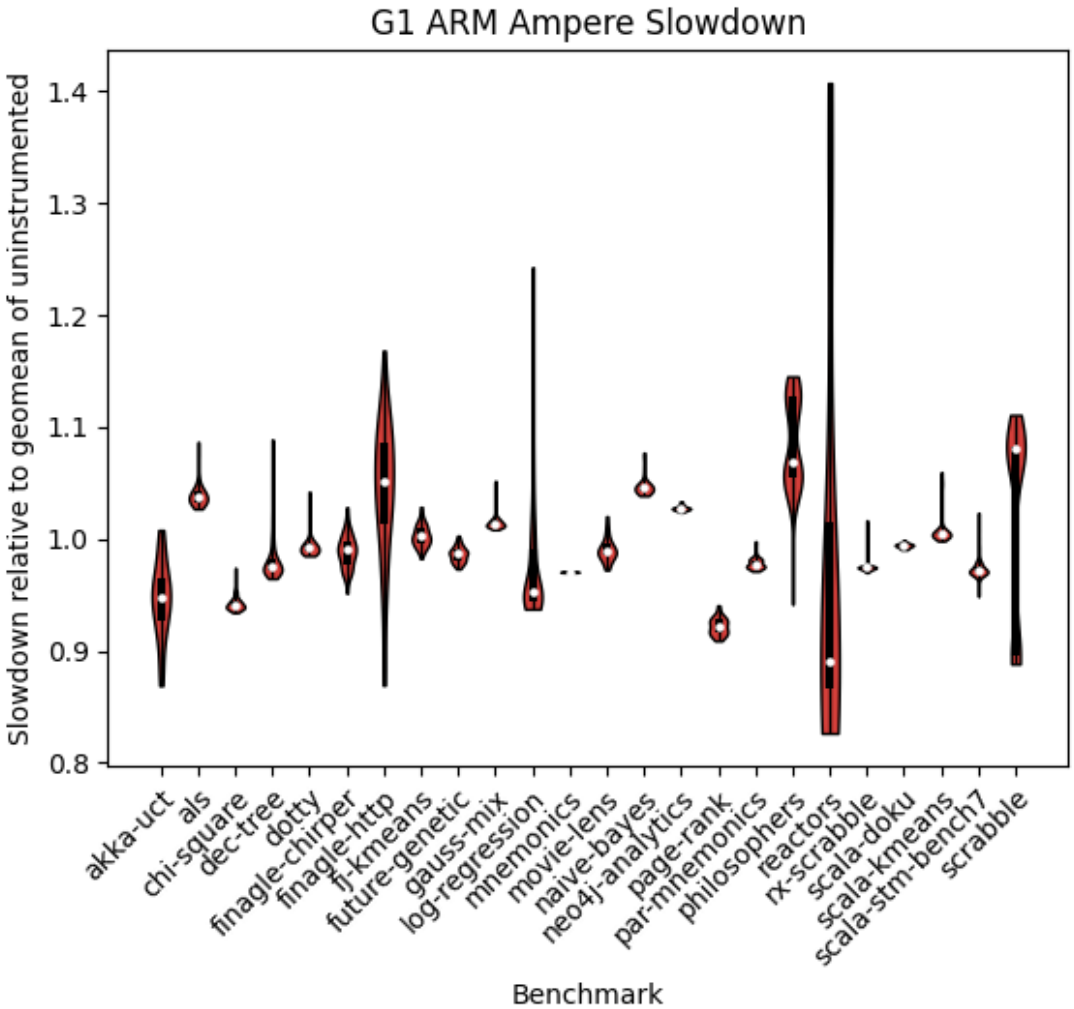

Figure 3.13: Customized violin plot of Renaissance benchmark slowdown using G1, (4 cores controlled using taskset), calculated against the geomean of uninstrumented execution with G1 for each benchmark. 600s duration run, ARM machine, measurements used from the last 300s, frequency 2GHz with a performance governor. Violin plot markings as in figure 3.11. db-shootout omitted as it failed to run correctly.

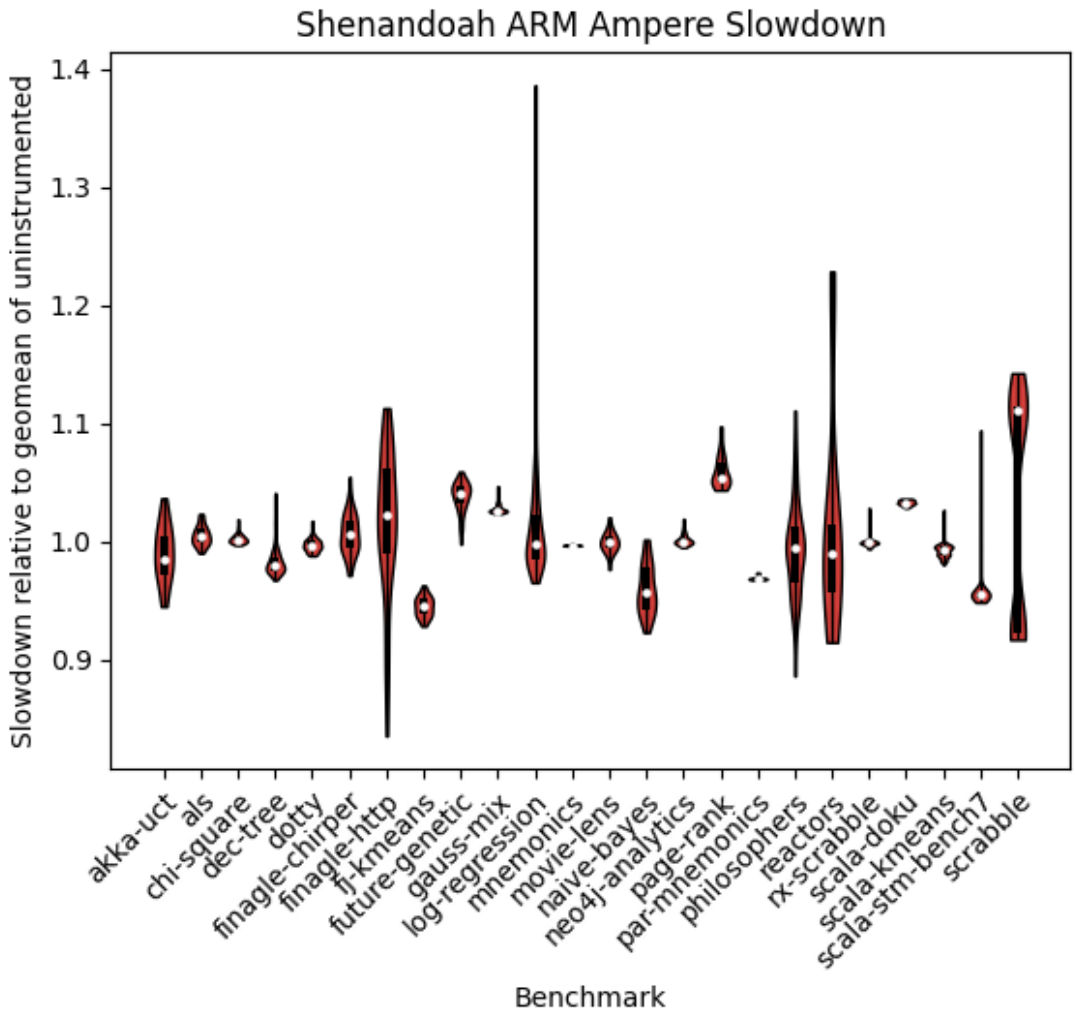

Figure 3.14: Customized violin plot of Renaissance benchmark slowdown using Shenandoah, (4 cores controlled using taskset), calculated against the geomean of uninstrumented execution with Shenandoah for each benchmark. 600s duration run, ARM machine, measurements used from the last 300s, frequency 2GHz with a performance governor. Violin plot markings as in figure 3.11. db-shootout omitted as it failed to run correctly.

3.7.3 Phase Classification based on Changepoint Analysis

In our initial experiments we investigated the use of elapsed wall-clock time, the measured on-cpu execution time of threads, and the number of retired micro-ops (µops) by threads. We used the measured quantities to perform changepoint analysis, with the expectation that deterministic applications would have a repeatable amount of work per iteration unless significant operation of VM service threads was present in an iteration. We hypothesise that there is merit investigating using counts of retired µops for phase classification rather than the more traditional elapsed wall clock time, as the latter is potentially sensitive to delays resulting, for example, from OS scheduling such as task migration and context switching, or from the hardware such as cache effects. For particularly short running benchmarks these delays have the potential to perturb iteration times.

Initially, we found unusual results when investigating small variations in the execution time of the deterministic programs used by [BBTK+17] to investigate warmup and steady state. For example, the results indicated that sometimes the total on-cpu cycles decreased when the execution time went up. We investigated turning off all unnecessary processor and ACPI sleep/power managements states but this did not remove the issue. We inserted error checks to ensure that we had not lost any measurements of thread quanta. Then we turned our attention to the filtering used by BCC tools that instrument *sched_switch* to investigate on-cpu issues. Some tools were filtering on the TASK_RUNNING state to determine if measurements should be attributed to a thread. We found that it was necessary to include Linux (kernel) thread scheduler states other than TASK_RUNNING to accurately measure the time and retired µops associated with each iteration, particularly the sleep states: TASK_INTERRUPTIBLE and TASK_UNINTERRUPTIBLE. This is because threads that voluntarily yield the OS scheduler quantum do so by setting their task state to the appropriate sleep state before calling into the linux *schedule* function. These tasks, when subsequently switched out, leave the run queue with the sleep task state, despite having done useful work and retired a significant number of µops. This is to be contrasted with tasks that are involuntarily scheduled off by the kernel, as a result of them using up their scheduling quantum. Such tasks leave the run queue with the TASK_RUNNING state. For short running benchmark iterations, the accumulated statistics of tasks leaving the run queue with the additional sleep states can add up to a significant part of the execution, distorting the results if omitted. Further, threads awaiting conditional synchronization may be periodically scheduled by the kernel to check if they can make progress, and

these events can begin to consume a small but significant number of retired µops and hence execution time.

In order to test our hypothesis of the suitability of using counts of retired µops to determine steady state in benchmark experiments, rather than the more traditional wall clock time, we base our experiments on the comprehensive work in [BBTK+17]. We use the same approach where practical, outlined below, including the same benchmarks, software and settings although we do not use a heavily controlled execution environment apart from fixing the CPU frequency. We do not believe that diminishes the veracity of our experiments since we are not trying to determine the same goal, i.e. of verifying whether or not steady state, warmup or any other state is achieved. We are simply comparing the use of alternate signals for the changepoint analysis; furthermore all data required for our results are collected concurrently during the same run i.e. under the exact same conditions.

We had some problem getting reliable performance out of the Richards benchmark, which tended to slow down execution and was omitted and so we use Java versions of: *binarytrees, fannkuchredux, fasta, nbody* and *spectralnorm*. The benchmarks take a seed parameter that affects the amount of work done in the iteration and hence the iteration time. The values we used gave iterations in the range of 1s - 3s depending on the benchmark, approximately 10 times the duration than used in [BBTK+17]. In our analysis we also use the R `changepoint` package `meanvar` routine which implements the PELT algorithm [KFE12]. For the penalty value we use Schwarz information criterion (SIC) $\beta = plogn$. Here p is the number of additional parameters introduced by adding a changepoint (set to $p = 15$ as in [BBTK+17]), and n is the number of observations which is 2000 iterations minus the number of outliers. Note that choosing an incorrect penalty value can lead to over or underfitting the time-series data of execution times for different iterations, leading to too many, or too few changepoints being detected. We use a minimum interval size of 2, and default values for all other arguments. We use 10 process executions of 2000 iterations, based on recommendations in [BBTK+17] which suggests that 10 process iterations are sufficient for statistically reliable results. We exclude outliers using the same algorithm described in [BBTK+17] and we fix the CPU frequency to avoid perturbation from DVFS.

3.7.3.1 Discriminating Section Trends

The results from the changepoint meanvar function returns the changepoints as iteration indices, and the section means and variances. We now must use this information to

classify any trends behaviour between the sections. In [BBTK+17] to discriminate the difference between the adjacent sections they compare the section means $\pm d$, where d is the maximum of the variance, or 0.001 (based on their observation that iteration times within the same section are typically within 0.001s of each other). We note that for the short running benchmarks, measured in fractions of a second, the difference between individual points and the mean is a small fraction, and so the variance approaches zero. Under these conditions, d, most often becomes the 0.001 term. For our experiments we encountered three problems using the variance in this way.

- Variance is not scale invariant. For large numbered values such as retired µops the difference between the individual benchmark iteration counts and their section mean is a large value and the variance then typically becomes huge, often eclipsing the measured values. (Similarly, also if using nanosecond units for the iteration times rather than sub 1s decimal fractions to perform the same computation)

- A quantity must somehow be chosen to compare with the variance in order to apply the discrimination function that is calibrated for the benchmark and the values used in the changepoint input calculation. This is possibly error prone, and risks fitting the analysis to the experimental results.

- The units of variance are not the same as the mean, and so it is questionable whether a meaningful value can be inferred from mean $\pm$ variance.

To avoid these problem areas, for the statistical dispersion term used for the discrimination function, we also experiment with using the standard deviation and compare segments using the mean $\pm sd$. This approach requires no extra parameters and the units are the same.

3.7.3.2 Results

The results of our phase classification benchmark experiments are summarised in table 3.5 and examples of executions in figures 3.15 & 3.16. We show results using both retired µops and elapsed wall clock time for the changepoint signal. Foreach benchmark, GC algorithm and changepoint signal, table 3.5 shows the median index that steady state was achieved (SSi), and the median time (SS time) and number of µops (SS µops) in the steady state phase. We also show the classifications,

whether flat (F), warmup (W) or slowdown (S) was attributed according to the algorithm in [BBTK+17]. When applying the classification/steady state algorithm we use both standard deviation and variance in the case of wall-clock time, however we use only standard deviation for the retired µops signal, for the reasons discussed in section 3.7.3.1.

Firstly we note that all executions achieve a steady state, whereas this was not the case in [BBTK+17]. The absence of steady state is declared when there are two non equivalent sections after iteration 1500. Possibly our use of longer iteration times ensured that our measurements showed less variation from induced delays. We leave an analysis using shorter iterations for future work. Looking down the *SSi* column, we see that there is broad agreement when steady state was achieved foreach signal and analysis. Where there is an obvious discrepancy (nbody g1) where the variance calculated steady state was noted at iteration 102, in contrast to the 0,1 in the other cases, there was no significant difference in the steady state time (see also figure 3.16). We note that when discriminating between flat and warmup the margin in the calculation is sometimes small with there being no significant observable change in the execution times (also figure 3.16).

In summary, our results show that using total µops as a univariate signal offers some promise as a measure of work done for a JVM application in execution. This preliminary experimental result, on a much less heavily constrained execution environment than the original work (albeit with less experimental rigour) of [BBTK+17] suggests further investigation is warranted.

3.7.4 Top Down Analysis

BPFxVM allows us to collect PMU event counts at OS scheduling quantum and thread granularity. We can use these data to investigate variations in the top-down behaviours of different threads during the course of execution. We show here our early work investigating this approach to gaining performance understanding in MREs.

Figure 3.17 captures such behaviour during a run of the Renaissance scrabble benchmark on an X86 machine with the G1 collector using violin plots. Mis-speculation overhead is typically low across all thread-groups. Threads attributed to GC demonstrate a wide and relatively evenly distributed variation of backend and retiring bound microarchitecture utilization, without any significant bulges in the violin plot that indicate frequently occurring values. Our technique reveals that some threads show a multimodal distribution of behaviours, for example ForkJoinPool.co and java.

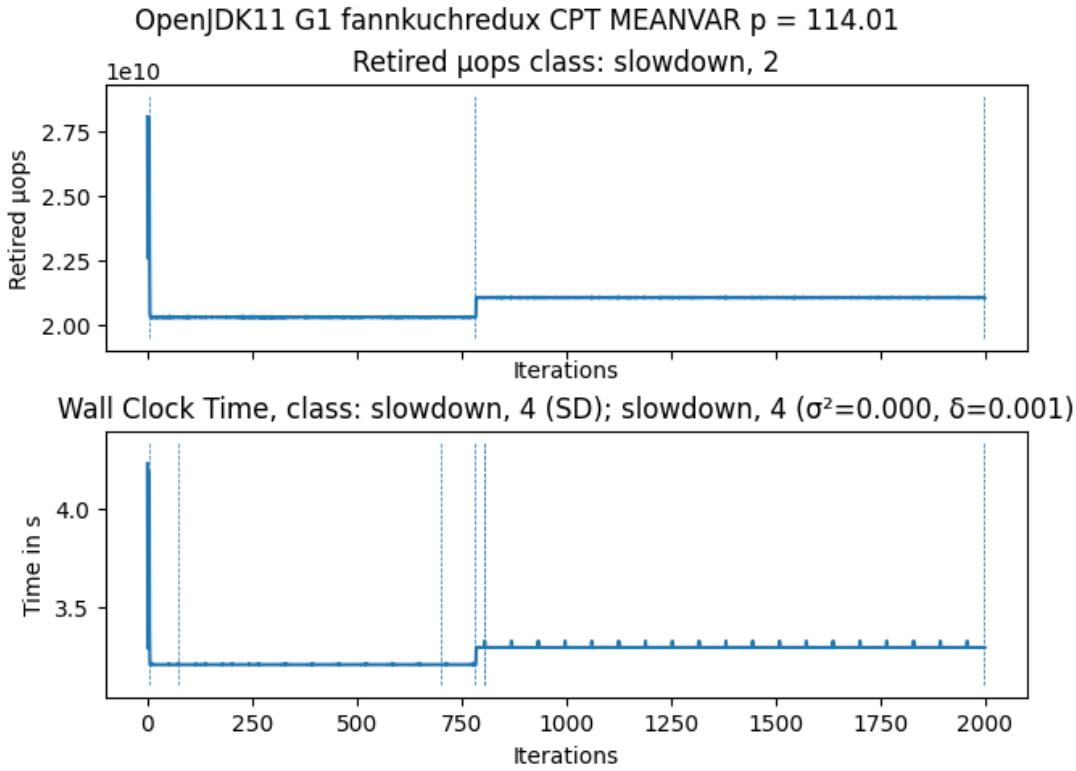

Figure 3.15: R Changepoint meanvar analysis of the Java fannkuchredux benchmark using the total number of µops retired each iteration (top) and the elapsed wall clock time for each iteration, showing both warmup and slowdown behaviour. The steady state/classification algorithm works backwards from the last changepoint and stops when it encounters the slowdown transition around iteration 783. This places the steady state after the second changepoint in the top plot (slowdown, 2) and the fourth changepoint (slowdown, 4) in the bottom plot when using both the standard deviation analysis (SD) and the variance analysis. The two terms in the lower title (σ^2 and δ) show that the section variance (σ^2) was small and so the dominant term (δ) used in the classification analysis was 0.001 (section 3.7.3.1). This was almost always the case.

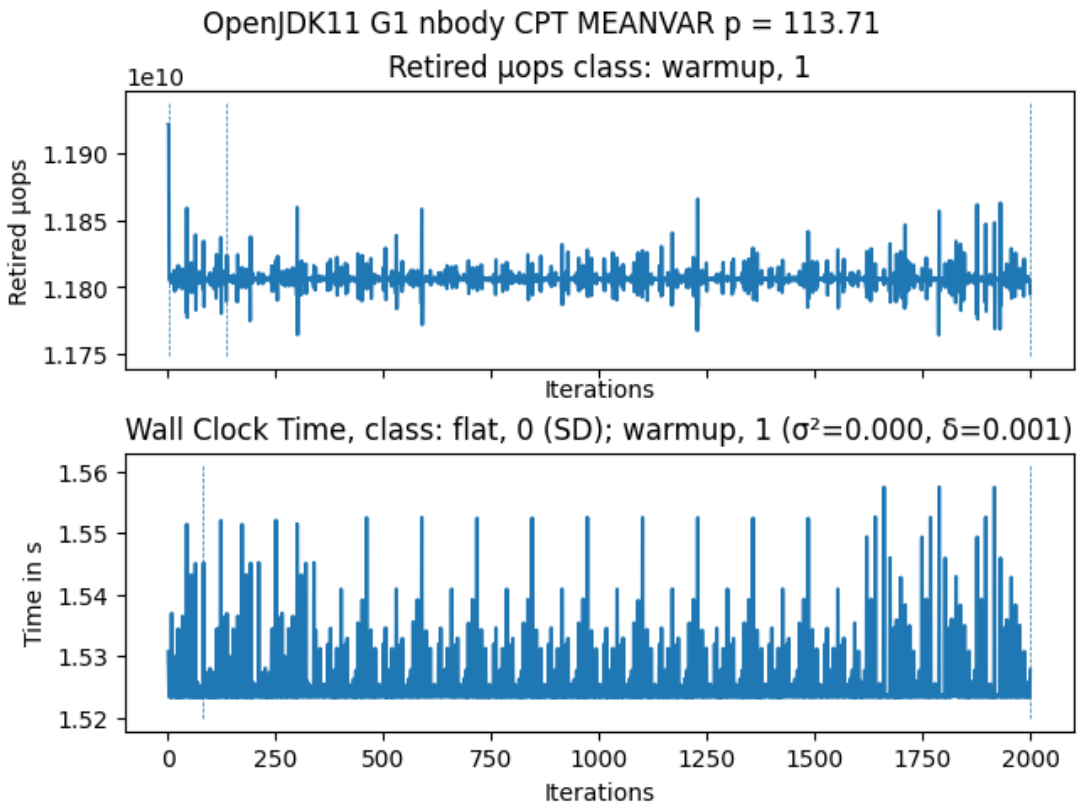

Figure 3.16: Showing an execution of the nbody benchmark using G1 GC. Where there is no clear trend in the signal, the discrepancy causing a flat or warmup classification is sometimes neither obvious nor particularly significant. According to the top signal (µops) warmup was attributed to be at the first changepoint at iteration 1. According to the wall clock time signal and comparing the mean $\pm sd$ the first section is considered to be equivalent to the second (last) and so the execution is given to be flat starting from iteration 0. Using the mean ± 0.001 shown in the second classification in the lower plot title, a difference is detected between the two sections and the execution is assigned warmup at iteration 102 without there being a clear downward trend in the execution times. See also nbody g1 in table 3.5.

		Retired µops			Wall Clock Time					
					Std Deviation			Variance		
Benchmark	**GC**	**Classification**	**SSi**	**SS µops**	**Classification**	**SSi**	**SS time(s)**	**Classification**	**SSi**	**SS time(s)**
binarytrees	g1	F[10]	0	1.90e+10 ± 1.79e+08	F[9],W[1]	0	1.736 ± 0.009	W[10]	1	1.938 ± 0.2065
binarytrees	shen	F[10]	0	1.82e+10 ± 1.61e+09	F[7],W[3]	0	1.663 ± 0.008	W[10]	1	1.995 ± 0.3305
fannkuchredux	g1	S[10]	783	2.08e+10 ± 9.40e+06	S[10]	783	3.2885 ± 0.001	S[10]	783	3.2885 ± 0.001
fannkuchredux	shen	F[8],S[1],W[1]	0	2.20e+10 ± 3.33e+09	F[3],S[3],W[4]	2	3.473 ± 0.0015	F[2],S[3],W[5]	3.5	3.48 ± 0.001
fasta	g1	F[1],W[9]	1	6.57e+09 ± 7.17e+07	F[2],W[8]	1	1.968	W[10]	1	1.97
fasta	shen	W[10]	1	6.63e+09 ± 8.14e+07	W[10]	1	1.85	W[10]	1	1.85
nbody	g1	F[3],W[7]	1	1.14e+10 ± 6.37e+07	F[10]	0	1.499 ± 0.003	F[1],W[9]	102	1.5 ± 0.0055
nbody	shen	F[2],W[8]	1	1.41e+10 ± 9.72e+06	F[5],W[5]	0.5	1.8955 ± 0.01	F[3],W[7]	1	1.8955 ± 0.01
spectralnorm	g1	S[10]	259	1.82e+10 ± 5.00e+07	S[10]	259	5.159	S[10]	261	5.076 ± 0.082
spectralnorm	shen	S[10]	258.5	1.82e+10 ± 4.87e+08	S[10]	259	5.141	S[10]	261	5.059 ± 0.082

Table 3.5: The results of the phase classification benchmark experiment using both retired µops and elapsed wall clock times. For the wall clock times we show phase classification/steady state analysis using both standard deviation and the variance based calculation (section 3.7.3.1). The experiment comprised 10 independent executions of 2000 iterations for each benchmark and GC (Shenandoah and G1). Of the 10 runs, the SSi column shows the median iteration number that steady state was achieved. Similarly, SS µops, and SS time(s), shows the median counts/times from the steady state execution with their standard deviations. The Classification indicates whether the run was determined to be flat (F), warmup (W) or slowdown (S) with the count in [] over the 10 executions.

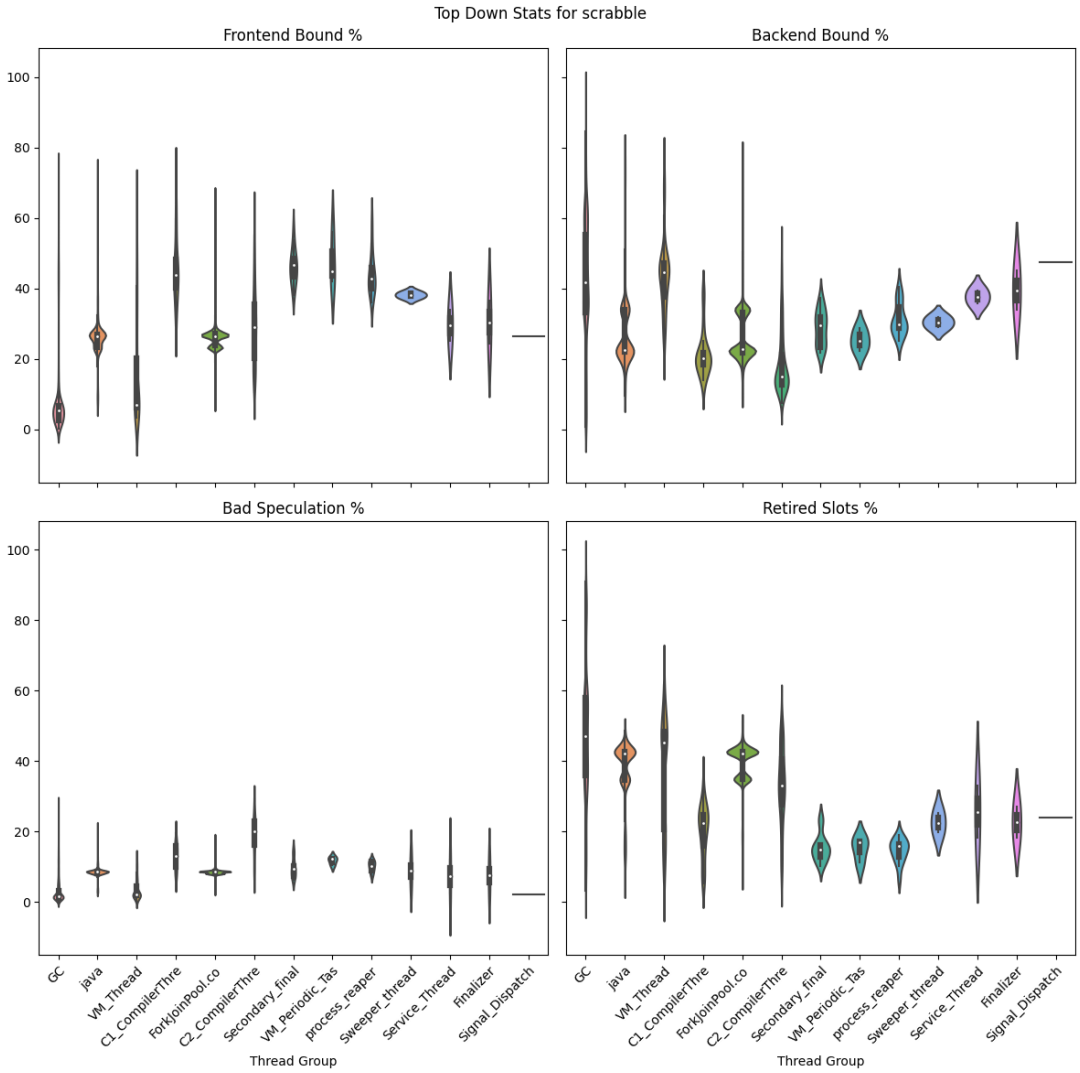

Figure 3.17: Top-down statistical analysis over all thread groups during a run of 200 iterations of the scrabble benchmark with the G1 garbage collector with a 12GByte heap.

Figure 3.18 shows top down statistics for GC threads only using the G1, Shenandoah and Parallelold garbage collectors during a run of the Renaissance reactors benchmark. Mis-speculation by the microprocessor is again low in all cases. The back end bound category is high for all GC algorithms, over the threshold of 20% that justifies further investigation according to [Yas] and particularly high for Parallelold with significant distributions centered on 60% and 75%. The commonly held belief is that GC threads are typically memory bound, but further investigation would be required to confirm that suspicion based on these results. It is possible to sample alternative performance counters, to determine whether backend bound stalls are contributed by memory bound or core bound behaviour, but we leave a comprehensive survey for future work. We do see some variation in microarchitecture utilisation across the GC, in particular for parallelold in the front end bound and backend bound categories.

We will continue this early work investigating trends in microarchitecture utilisation across different threads and workloads. We will extend the analysis to other architectures and MREs.

3.8 Conclusions

We have presented techniques that demonstrate how we can collect extremely detailed information concerning the top-down behaviour of each thread execution quantum at relatively low-overhead for both ARM and Intel processor microarchitectures. The use of system time in nanosesonds since boot provides a means to align measurements from the kernel, and from function hooked events concerning the execution and occurrence of specific events of interest related to the application, and to the managed language runtime itself. We have investigated previous approaches for determining if a managed language application reaches a state of warm-up, where steady state execution time is demonstrated. We argue for a different measure, namely whether an application reaches a steady-state of behaviour that is described by the number of µops or instructions retired that are attributed to not only application threads, but also to the many threads associated with supporting the managed language runtime environment.

In future work, we plan to use our tools to investigate criticality analysis and offline causal profiling using information traces that capture all on-cpu quanta and all thread-state transitions between runnable and sleeping/blocked states. For managed runtimes such as Node.js/V8 where threads do not have specific roles, it is necessary to trace the start/stop of specific operations related to virtual machine activities, for example

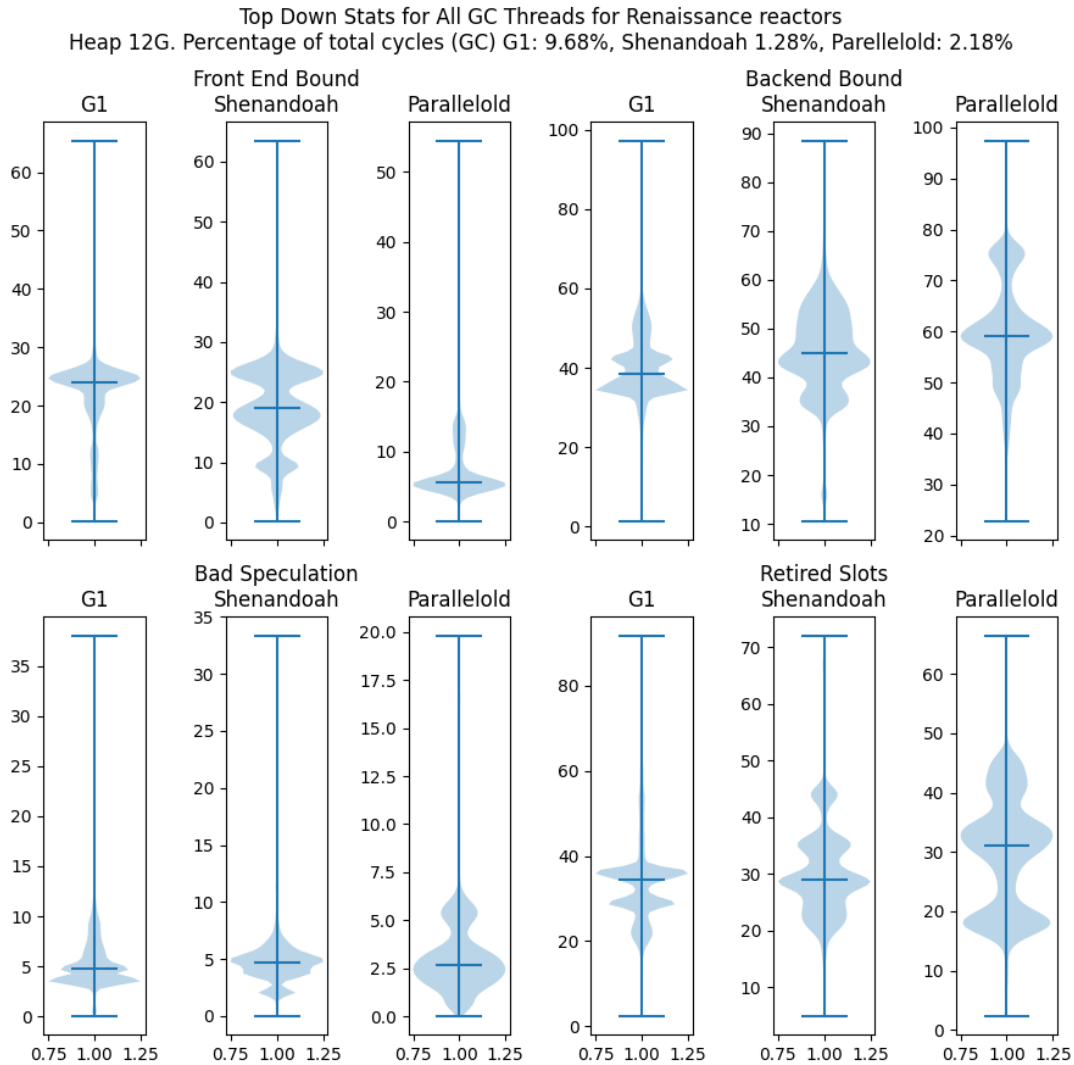

Figure 3.18: Showing the top-level top-down microarchitecture utilisation statistics over a run of the Renaissance reactors benchmark.

due to garbage collection, deoptimization and compilation. Although specific user statically defined tracepoints exist in Node.js for GC, there are no such tracepoints for deoptimization. It is possible for dynamically compiled application code to add statically defined tracepoints to JIT compiled dynamic languages such as Javascript, Python and Go by defining the tracepoints in a small shared-library that is linked at runtime[5]. Node.js/V8 will require special treatment as its event-loop based architecture where threads do not have specific roles requires the tracing of the start/stop of events in the managed runtime and the application itself.

[5]Such an approach is used here: `https://github.com/sthima/libstapsdt`.

Chapter 4

Conclusions

This thesis presents firstly an investigation into performance implications arising from choices encountered when porting the JIT compilation system of a managed runtime to target a new architecture, ARMv8. Secondly, it presents two fine-grained, thread-level studies of the microarchitectural behaviour of a managed runtime using our own tool, BPFxVM, a framework based on the eBPF subsystem in the Linux kernel.

4.1 CallSites: ARM JIT Compilation

The work presented in Chapter 2 considered the aspects of the ARMv8 architecture that constrain the implementation of call-sites for JIT compiled code: namely the ISA, memory consistency model, the cache coherency protocol and the limit to the class of instructions that are safe to modify and execute concurrently. An investigation was conducted into the performance of alternate strategies using benchmark experiments, and profiles derived from microarchitecture performance events and bytecode instrumentation.

The results have shown variation in the performance of the different call-site schemes of up to 12% in the JVM benchmark suite experiments and also variation across different implementations of the architecture. We have shown that for the advanced Neoverse N1 platform there is little difference in the performance of the alternate schemes. We have also demonstrated that code management and code cache organisation affects the behaviour of the Trampoline scheme which degenerates to the optimum scheme of a single instruction direct branch below 128MB displacements. This finding is relevant to other ISAs with limited direct branch displacements.

Concerning the Experiments The JVM benchmark experiments in 2.7.4 were designed in order to report a confidence interval, when comparing the performance of the alternate implementations for the iteration based experiments (DaCapo and Renaissance). The multiple warmed up iteration times were used to calculate a mean with a confidence interval, that could then be normalised to the optimum Direct scheme to form a ratio with its own confidence interval according to Fieller's theorem [Fie54]. Given the magnitude of the variation in the relative performance of the callsite schemes in MaxineVM was small, the experiments could have been better designed. Using multiple independent executions would have enabled the calculation of an independent confidence interval in the mean for each scheme and a more robust statistical analysis including testing the null hypothesis, i.e. that the alternate strategies show no significant variation in performance. This would have also provided a mean result with a confidence interval for the SPECjvm2008 benchmark results. Rather than timed iterations, the SPECjvm2008 benchmark suite runs for a fixed interval and reports a single result in operations per unit time for each benchmark, and so must be run multiple times to form a mean and a confidence interval.

4.1.1 Future Work

Experiments with the Trampoline scheme suggest that there is potential for managing code and the code cache, in order to localise code on the critical path, to gain a higher percentage of direct branches, and so a more optimised execution profile. We plan to experiment with reorganising the code cache partitions in MaxineVM in order to study the potential benefit that may be achievable.

4.2 MRE Performance Understanding using BPFxVM

This chapter has presented two studies of managed runtime performance using BPFxVM, a low overhead, extensible tool that enables fine-grained investigation into the behaviour and performance MREs running on Linux. We have shown the geomean overhead of BPFxVM is less than 2% and we have noted that some of the benchmarks exhibit speedups when traced, which we will investigate further in order to establish the statistical significance and causes.

Top Down Microarchitecture Analysis This study reveals for the first time we believe, top-down microarchitecture utilisation statistics at individual thread, and scheduling quantum granularity for a MRE and its application, running on both x86 and ARMv8 architectures. We have shown the dynamic behaviour of threads utilisation of the microarchitecture in both violin plots and traces that can be viewed in the Chrome profiler. This approach builds on previous studies [Yas14] and tools [Kle22] that produce a single mean value for top down stats, whether for the application as a whole, or specific CPU cores. The results reveal multimodal distributions offering the potential for greater understanding of optimisation opportunities that would otherwise be obscured by mean values. We have demonstrated that GC threads have a high back-end bound behaviour that may confirm the conventional wisdom that such threads are typically memory bound.

Warm-up and Steady-State Analysis This study has compared using microarchitecture performance counters against the more traditional elapsed wall-clock time to determine warm-up and steady-state behaviours. The work is based on the methodology of [BBTK+17] and has also been influenced by [CR16] that questions the reliability of timed measurements. The results show that there is promise in using counts of retired µops per benchmark iteration rather than elapsed time, supporting the hypothesis that warmed-up iterations should comprise a consistent amount of measurable work done. We note however that there are limitations to this approach, in particular for parallel programs, where variation in synchronisation operations e.g. due to resource contention may skew retired µop counts. A phenomenon that also similarly affects timed experiments.

4.2.1 Future Work

Although the experiments have focused mainly on the x86 platform, some preliminary experiments have been run on ARMv8 and the work will be extended to fully include the architecture, and also include other MREs such as Node.js/V8.

Phase and Steady State Analysis

This study has shown the utility of retired µops as a measure of work done for phase and steady-state classification using changepoint analysis. We intend to continue this

study to investigate the work contributed by specific thread groups such as VM components including JIT compilation or GC. We will also look at examining changes across top-down statistics both application wide and using specific thread groups. Whereas this thesis has shown the use of univariate signals for changepoint analysis, it is also possible to perform multivariate analysis using a number of signals. We believe that top-down statistics are potentially interesting candidates for this investigation. We will include the popular mainstream benchmark suites in our characterisation.

Top-Down Analysis

This preliminary work on top-down analysis has focused on demonstrating the top-level statistics for individual threads, or thread groups. We will continue with this work in order to produce detailed characterisation of workload and VM service components under different conditions in order to better understand variation in microarchitecture utilisation. These early results show statistics that are over the threshold warranting further investigation [Yas], such as the GC threads in figure 3.18 and we will look further into the sub categories [Yas14] in order to gain an understanding of the root causes of the stalls and to seek out optimisation opportunities.

Other Tracepoints

The use of BPFxVM in this thesis has demonstrated the performance aspects of threads derived from changes in microarchitecture performance event counts. We can extend the tool with additional features by hooking specific VM operations or other kernel tracepoints. In addition to the comprehensive set of tracepoints exposed by the kernel, eBPF enables tracing dynamic shared library functions by inserting *uprobes*. We have experimented using this approach to trace *deoptimization* [HCU92] in the OpenJDK JVM in order to understand the contribution of adaptive optimisation on performance. These techniques will particularly useful for our investigations of other MREs such as Node.js/V8 where threads expose no clearly identifiable role to the OS.

Bibliography

[AMDB07a] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. A model for self-modifying code. In Jan L. Camenisch, Christian S. Collberg, Neil F. Johnson, and Phil Sallee, editors, *Information Hiding*, pages 232–248, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[AMDB07b] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. A model for self-modifying code. In *Proceedings of the 8th International Conference on Information Hiding*, IH'06, pages 232–248, Berlin, Heidelberg, 2007. Springer-Verlag.

[AMN+11] Antoine Amarilli, Sascha Müller, David Naccache, Daniel Page, Pablo Rauzy, and Michael Tunstall. Can code polymorphism limit information leakage? In Claudio A. Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, pages 1–21, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[AR19] Andreas Abel and Jan Reineke. Uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 673–686, New York, NY, USA, 2019. Association for Computing Machinery.

[arm] Arm v8-a, topdown metrics source. `https://www.brighttalk.com/webcast/17792/384060/top-down-performance-analysis`, note=Accessed: 2020-0-03.

[ARM17a] ARM. Arm cortex-a57 software optimization guide. Technical report, 2017.

[ARM17b] ARM. Arm cortex-a72 software optimization guide. Technical report, 2017.

[ARM19a] ARM. Arm Architecture Reference Manual Armv8, 2019.

[ARM19b] ARM. Arm neoverse n1 cpu. Technical report, 2019.

[ARM19c] ARM. Neoverse E1, 2019.

[ARM20a] ARM. Arm cortex-a55 software optimization guide. Technical report, 2020.

[ARM20b] ARM. Arm neoverse n1 core software optimization guide. Technical report, 2020.

[ARM20c] ARM. Arm neoverse n1 core technical reference manual. Technical report, 2020.

[ARM20d] ARM. Arm neoverse n1 (mp050) software developer errata notice. Technical report, 2020.

[Azu19] Azul Systems. Zulu Embedded Open Source Java for Embedded Systems, 2019.

[BBTK+17] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[CA20] K. Criswell and T. Adegbija. A survey of phase classification techniques for characterizing variable application behavior. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):224–236, 2020.

[CBGM12] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. *SIGARCH Comput. Archit. News*, 40(3):225–236, June 2012.

[CR16] Jiahao Chen and Jarrett Revels. Robust benchmarking in noisy environments, 2016.

[CSV07] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 66–77, New York, NY, USA, 2007. ACM.

[DCN+19] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, pages 15–27, New York, NY, USA, 2019. ACM.

[DDZ18] Shi Dawei, Lv Delong, and Ye Zhibin. Dynamic self-modifying code detection based on backward analysis. In *Proceedings of the 2018 10th International Conference on Computer and Automation Engineering*, ICCAE 2018, pages 199–204, New York, NY, USA, 2018. ACM.

[Dev20] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual, Volume 1: Application Programming*, 2020.

[doc] Matplotlib documentation. Violin plot cusotmization. `https://matplotlib.org/stable/gallery/statistics/customized_violin.html`, not=Accessed: 2021-09-03.

[EDE12] S. Eyerman, K. Du Bois, and L. Eeckhout. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *2012 IEEE International Symposium on Performance Analysis of Systems Software*, pages 145–155, 2012.

[EFK11] I. A. Eckley, P. Fearnhead, and R. Killick. Analysis of changepoint models. In *Bayesian Time Series Models*. Cambridge University Press, 2011.

[Fie54] Edgar Fieller. Some problems in interval estimation. *Journal of the Royal Statistical Society. Series B (Methodological)*, 16(2):175–185, 1954.

[fun] Sigill issue with uretprobes. `https://github.com/iovisor/bcc/issues/3034`, note=accessed: 2021-09-03.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '07, page 57–76, New York, NY, USA, 2007. Association for Computing Machinery.

[Gha96] Kourosh Gharachorloo. *Memory consistency models for shared-memory multiprocessors*. PhD thesis, Stanford University, 1996.

[GNL18] Swapnil Gaikwad, Andy Nisbet, and Mikel Luján. Performance analysis for languages hosted on the truffle framework. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang '18, New York, NY, USA, 2018. Association for Computing Machinery.

[Gou04] Isaac Gouy. The computer language benchmarks game. `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`, 2004. Accessed: 2021-3-1.

[Gre] Brendan Gregg. Java performance analysis on linux with flame graphs, javaone 2016 presentation. `https://www.brendangregg.com/Slides/JavaOne2016_JavaFlameGraphs.pdf`. Accessed: 2021-09-03.

[GSF12] P. Gopi, G. Singh, and G. Favor. X-gene™: 64-bit arm cpu and soc. In *2012 IEEE Hot Chips 24 Symposium (HCS)*, pages 1–19, 2012.

[HBLF13] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Librando: transparent code randomization for just-in-time compilers. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 993–1004, New York, NY, USA, 2013. ACM.

[HCU92] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM.

[HD14] Andrew Haley and Andrew Dinn. Openjdk on aarch64 update. `https://aph.fedorapeople.org/Aarch64-fosdem-2014.pdf`, FOSDEM 2014.

[HM11] Christian Häubl and Hanspeter Mössenböck. Trace-based compilation for the java hotspot virtual machine. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, page 129–138, New York, NY, USA, 2011. Association for Computing Machinery.

[HM21] Ranjan Hebbar and Aleksandar Milenković. An experimental evaluation of workload driven dvfs. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, ICPE '21, page 95–102, New York, NY, USA, 2021. Association for Computing Machinery.

[HNG14] Tobias Hartmann, Albert Noll, and Thomas Gross. Efficient code management for dynamic multi-tiered compilation systems. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, page 51–62, New York, NY, USA, 2014. Association for Computing Machinery.

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Amsterdam, 5 edition, 2012.

[HZKL19] Tim Hartley, Foivos S. Zakkak, Christos Kotselidis, and Mikel Luján. An analysis of call-site patching without strong hardware support for self-modifying-code. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, page 131–143, New York, NY, USA, 2019. Association for Computing Machinery.

[Int15] Intel. Method and apparatus for providing hardware support for self-modifying code, PCT/US2015/030411, 2015.

[Int16] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1*, 2016.

[Int19] Intelm. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2, 2019.

[IOV22] IOVisor. BPF Compiler Collection (BCC), accessed 2022. `https://github.com/iovisor/bcc`.

[Jam19] JamaicaVM. A hard realtime Java bytecode-based Virtual Machine, 2019.

[jik] Jikervm project issues dashboard. `https://xtenlang.atlassian.net/jira/software/c/projects/RVM/issues/?filter=updatedrecently`. accessed: 2021-11-05.

[KCR+17] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. Heterogeneous managed runtime systems: A computer vision case study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 74–82, New York, NY, USA, 2017. ACM.

[KFE12] R. Killick, P. Fearnhead, and I. A. Eckley. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 107(500):1590–1598, 2012.

[KJ13] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 international symposium on memory management*, pages 63–74, 2013.

[Kle22] Andi Kleen. Intel pmu profiling tools: Github repository. `https://github.com/andikleen/pmu-tools`, 2022. Accessed: 2022-12-04.

[LTCC+16] Michael A Laurenzano, Ananta Tiwari, Allyson Cauble-Chantrenne, Adam Jundt, William A Ward, Roy Campbell, and Laura Carrington. Characterization and bottleneck analysis of a 64-bit armv8 platform. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 36–45. IEEE, 2016.

[Ltd20] Arm Ltd. Cortex-a53 – arm, (accessed November 2020). https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a53.

[LTZ+15] Dumitrel Loghin, Bogdan Marius Tudor, Hao Zhang, Beng Chin Ooi, and Yong Meng Teo. A performance study of big data on small nodes. *Proc. VLDB Endow.*, 8(7):762–773, February 2015.

[Maj] Zoltan Majo. Allow stack walking pass through method handle intrinsic frames. https://bugs.openjdk.java.net/browse/JDK-8153167. Accessed: 2021-09-03.

[MDHS10a] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of java profilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 187–197, New York, NY, USA, 2010. Association for Computing Machinery.

[MDHS10b] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of java profilers. *SIGPLAN Not.*, 45(6):187–197, June 2010.

[Mic19] MicroPython. Python for microcontrollers, 2019.

[MKP11] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679 – 691, 2011.

[Myr10] Magnus O. Myreen. Verified just-in-time compiler on x86. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 107–118, New York, NY, USA, 2010. ACM.

[NNRL19] Andy Nisbet, Nuno Miguel Nobre, Graham Riley, and Mikel Luján. Profiling and tracing support for java applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 119–126, New York, NY, USA, 2019. Association for Computing Machinery.

[Pan] Andrei Pangin. Asyncgetcalltrace fails to traverse valid java stacks. `https://bugs.openjdk.java.net/browse/JDK-8178287`. Accessed: 2021-09-03.

[PM20] Roldan Pozo and Bruce Miller. Java scimark 2.0, 2004 (accessed November 2020). `https://math.nist.gov/scimark2/index.html`.

[PRL+19a] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 31–47, New York, NY, USA, 2019. Association for Computing Machinery.

[PRL+19b] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Wuerthinger, and Walter Binder. On evaluating the renaissance benchmarking suite: Variety, performance, and complexity, 2019.

[PSB+20] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, Jamshed Jalal, Mark Werkheiser, and Anitha Kona. The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc. *IEEE Micro*, 40(2):53–62, 2020.

[PVA+14] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, pages 5:1–5:6, New York, NY, USA, 2014. ACM.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, volume 1, pages 1–12, 2001.

[RCA99] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 16–27. IEEE, 1999.

[RO16] Carl G. Ritson and Scott Owens. Benchmarking weak memory models. *SIGPLAN Not.*, 51(8), February 2016.

[Ros01] John Rose. Jdk-4506997, 2001.

[RSM+18] Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. Sulong, and thanks for all the bugs: Finding errors in c programs by abstracting from the native execution model. *SIGPLAN Not.*, 53(2):377–391, March 2018.

[RVV+13] Nikola Rajovic, Lluis Vilanova, Carlos Villavieja, Nikola Puzovic, and Alex Ramirez. The low power architecture approach towards exascale computing. *Journal of Computational Science*, 4(6):439 – 443, 2013. Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011.

[SBEE17] J. B. Sartor, K. D. Bois, S. Eyerman, and L. Eeckhout. Analyzing the scalability of managed language applications with speedup stacks. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 23–32, 2017.

[SEH11] A. Sembrant, D. Eklov, and E. Hagersten. Efficient software-based online phase classification. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 104–115, 2011.

[SFP+20] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. Armv8-a system semantics: instruction fetch in relaxed architectures. In *ESOP 2020-29th European Symposium on Programming*, 2020.

[SSO+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

[TGB19] K. Taht, J. Greensky, and R. Balasubramonian. The pop detector: A lightweight online program phase detection framework. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 48–57, 2019.

[TT13] Bogdan Marius Tudor and Yong Meng Teo. On understanding the energy consumption of arm-based multicore servers. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 267–278, New York, NY, USA, 2013. ACM.

[VCMKS12] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. Shuffling against side-channel attacks: A comprehensive study with cautionary note. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, pages 740–757, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[WHVDV+13] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9(4):30:1–30:24, January 2013.

[Wik18] WikiChip. X-gene 1 apm883408-x1 - appliedmicro, 2018.

[XWH+18] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. DCAPS: dynamic cache allocation with partial sharing. In Rui Oliveira, Pascal Felber, and Y. Charlie Hu, editors, *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 13:1–13:15. ACM, 2018.

[Yas] Ahmad Yasin. Intel tma metrics. `https://download.01.org/perfmon/TMA_Metrics.xlsx`. Accessed: 2021-09-03.

[Yas14] A. Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014.

[YBM15] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. Computer performance microscopy with shim. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 170–184, New York, NY, USA, 2015. Association for Computing Machinery.

[ZBB15] Yudi Zheng, Lubomír Bulej, and Walter Binder. Accurate profiling in the presence of dynamic compilation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 433–450, New York, NY, USA, 2015. Association for Computing Machinery.