An Analysis of Call-Site Patching without Strong Hardware Support for Self-Modifying-Code

DOI:
https://doi.org/10.1145/3357390.3361027

Document Version
Accepted author manuscript

An Analysis of Call-Site Patching without Strong Hardware Support for Self-Modifying-Code

Tim Hartley
The University of Manchester
United Kingdom
timothy.hartley-2@postgrad.manchester.ac.uk

Foivos S. Zakkak
The University of Manchester
United Kingdom
foivos.zakkak@manchester.ac.uk

Christos Kotselidis
The University of Manchester
United Kingdom
christos.kotselidis@manchester.ac.uk

Mikel Luján
The University of Manchester
United Kingdom
mikel.lujan@manchester.ac.uk

Abstract

With micro-services continuously gaining popularity and low-power processors making their way into data centers, efficient execution of managed runtime systems on low-power architectures is also gaining interest. Apart from the inherent performance differences between high and low power processors, porting a managed runtime system to a low-power architecture may result in spuriously introducing additional overheads and design trade-offs.

In this work we investigate how the lack of strong hardware support for Self Modifying Code (SMC) in low-power architectures, influences Just-In-Time (JIT) compilation and execution in modern virtual machines. In particular, we examine how low-power architectures, with no or limited hardware support for SMC, impose restrictions on call-site implementations, when the latter need to be patchable by the runtime system. We present four different memory-safe implementations for call-site generation and discuss their advantages and disadvantages in the absence of strong hardware support for SMC. Finally, we evaluate each technique on different workloads using micro-benchmarks and we evaluate the best two techniques on the Dacapo benchmark suite showcasing performance differences up to 15%.

CCS Concepts • **Software and its engineering → Software performance**; *Just-in-time compilers*; Runtime environments.

Keywords JIT compilation, Self modifying code, AArch64, RISC

MPLR '19, October 21–22, 2019, Athens, Greece

ACM Reference Format:
Tim Hartley, Foivos S. Zakkak, Christos Kotselidis, and Mikel Luján. 2019. 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 '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357390.3361027

1 Introduction

Since the introduction of smart-phones, low power processors have been constantly gaining momentum while entering new markets. Initially the term "embedded processors" was limited to micro-controllers and devices incapable of running code larger than few kilobytes in size. However, nowadays, the use of the term has been widened as low-power architectures are constantly becoming computationally more capable. Hence, low-power processors can be currently found, among others, in Internet of Things (IoT) devices, autonomous vehicles, edge nodes of distributed systems, and data centers [6]. This wide adoption has also resulted in a wide variety of applications running on such low power processors, ranging from embedded applications to demanding micro-services.

The constant increase in computational capability of these architectures has been also accompanied by a shift in the programming languages used to program them. In addition to traditional languages such as C and C++ , many applications are written in managed languages and rely on managed runtime systems to be efficiently executed [7, 18, 22]. Although managed languages apply the "write once run anywhere" approach across different architectures, the same does not hold for performance portability due to differences in the underlying platforms and architectures. Apart from the performance differences derived by the power/performance targets each architecture or processor tries to meet, porting a managed runtime system to a low-power processor can spuriously introduce additional overheads and design trade-offs.

A prime example is the variance of hardware support for Self-Modifying Code (SMC) between x86-64 and ARM or

RISC architectures.Managed runtime systems rely on Just-in-Time (JIT) compilation and dynamic code optimisations to translate platform agnostic bytecodes to native code, re-steering execution between the different versions of the generated code. Typically, code is optimised at method or block granularity and the re-steering of the execution to the optimised code is done by calling, i.e. branching to, the compiled methods or blocks. Additionally, there are cases where managed runtime systems need to re-compile some code segments. Such cases include: *de-optimisation*, where an assumption made during compilation is no longer valid; and *tiered compilation*, where methods that are on the "hot" path get further optimised by utilising more aggressive compiler optimisations. To handle such code alternations and to be able to execute the latest version of each method, managed runtime systems rely on call-site patching. They essentially modify code *on the fly* to change the target addresses of calls. To effectively patch all call-sites invoking a method, managed runtime systems 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 to make it lazily patch call-sites when they get executed. 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 feature strong hardware support for SMC. On such platforms simply overwriting a call instruction is sufficient in most cases to redirect a call-site to a different target without further concern. Note that the code being modified may be being executed by a different core at the same time, thus SMC requires either hardware support or synchronisation at the software level to ensure atomic updates and a coherent view of the code across the different cores of the processor.

In this work we investigate how the lack of strong hardware support for SMC in low-power architectures can affect the performance of managed runtime systems. More specifically this paper contributes the following:

- It discusses the constraints imposed by different architectures, such as AArch64 and x86-64 with respect to SMC (see §3).
- It studies (see §4) and evaluates (see §6) four different Java call-site implementations for low-power architectures, focusing on their performance and the corresponding overheads of their patching. The evaluation is performed on the AArch64 architecture against micro-benchmarks and standard Java benchmarks (for the best two implementations) in the context of MaxineVM [19, 28].
- It confirms the expectation that different call-site implementations exhibit not only different capabilities and implementation complexity, but also reveals a performance variation of up to 15%.

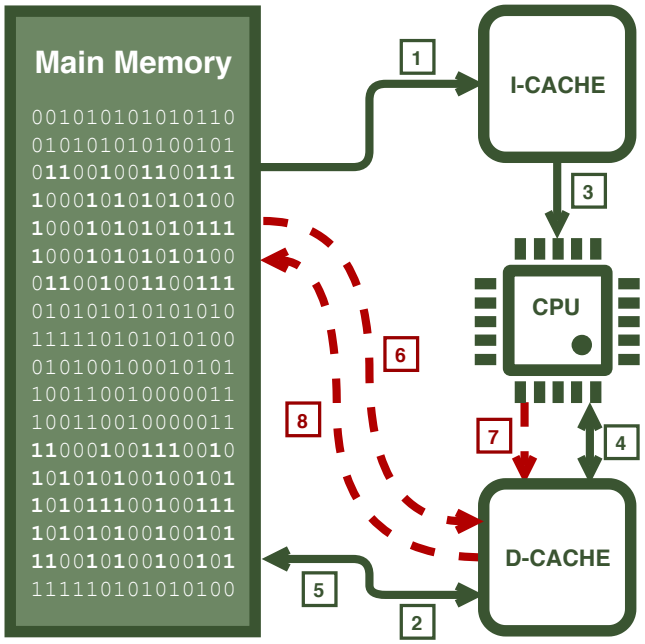

Figure 1. Code-patching illustration.

2 Background

Any form of JIT compilation results in Self-Modifying Code (SMC); i.e., the program itself generates new native code, replaces existing code, and executes it. Due to the conceptual separation of applications and the underlying runtime systems, developers typically are not aware of SMC since it is abstracted away by the runtime system via tiered JIT compilation. If we forget about this virtual distinction between applications and runtime systems, we observe that both of them are executed under the same process. As a result, from the processor's point of view, they are essentially a single program that generates and modifies its own code.

2.1 Von Neumann Architectures and SMC

Contemporary processors are based on the Von Neumann architecture[1] and rely on caches to optimise performance by reducing the latency required to access main memory, both for data and instructions. In such architectures, the instruction cache is utilised by hardware pre-fetchers to try and fetch ahead of time the instructions that are going to be executed in the near future.

Figure 1 illustrates the data-flow of Von Neumann architectures in the context of code patching. The solid green arrows indicate the typical data-flow, while the dashed red arrows indicate the data-flow of self-modifying code. The main memory contains both the code and the data. The instruction cache (I-Cache) caches code from the main memory, which is fetched by the I-Cache pre-fetcher or upon cache

[1] https://en.wikipedia.org/wiki/Von_Neumann_architecture

misses [1]. Similarly the data cache (D-Cache) caches data from the main memory [2]. Based on the current value of the Program Counter (PC), the processor reads instructions from the instruction cache and executes them [3]. Some of these instructions might modify data, resulting in writes to the D-Cache [4] which at some point are written-back to the main memory [5], depending on the memory model of the processor. In the case of SMC, the processor might need to fetch code into the data cache in order to read and modify it [6]. Consequently, the modified code is written to the data cache [7] which 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.

Due to the involvement of two kinds of caches (I-Cache and D-Cache), keeping the code coherent across the system is more complicated than keeping the data coherent. As a result, different architectures provide different hardware support for SMC [16] in a similar manner as they implement different memory models. That said, self-modifying-code needs to comply with the underlying architecture's constraints to ensure that the modified code will become visible and eventually executed.

2.2 Categorisation of Processors

SMC support combines elements of the memory model and the coherence mechanisms of the architecture. The stronger the guarantees, the more intuitive the architecture is to the user, but it also becomes more difficult to implement and scale to a high numbers 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 which target low power applications tend to offer weaker guarantees regarding either data or code coherence. Examples of such architectures are ARMv7, ARMv8 (AArch32/AArch64) and RISC-V. For the remainder of this paper we categorise processors with the terms high-end and low-power. Apart from providing weaker guarantees, typical low-power architectures also provide a reduced instruction set, compared to the one provided by high-end architectures.

In the following section we discuss how the differences between low-power and high-end processors impose different constraints on SMC and thus on the implementation of patchable call-sites in modern Java Virtual Machines.

3 Architectural Constraints on SMC

Due to their resource demanding nature general purpose managed runtime systems have evolved and been optimised to run primarily on high-end architectures. As a result, the implementation of some design decisions on low-power architectures may introduce some performance overheads compared to the corresponding implementations on high-end architectures. We note however that there are examples such as JavaME and Android that have been developed to specifically target resource constrained devices. Each type of architecture or processor design has different performance targets in a predefined power envelope. 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.

Excluding the aforementioned design decisions, a number of additional distinctive elements can also result in performance deviations between different types of architectures. Such differences derive mainly from: *i)* limitations imposed by the instruction set architecture (ISA), *ii)* explicit synchronisation required by the memory model, *iii)* limited support for self-modifying code. In the remainder of this Section we focus on self-modifying code and its implications on hardware implementations.

3.1 Call-Site Size, Patch Size, and Atomicity | ISA

Call-sites are essentially code segments that perform a function call. In high-end architectures, such as x86-64 [17], a call-site typically comprises a single instruction, e.g. `CALL <target>`. The x86-64 ISA enable *near* direct calls through the `CALL rel32` instruction. The `rel32` operand contains a 32-bit displacement 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. On the contrary, in low-power architectures with reduced instruction set computer (RISC) ISAs that feature fixed-size instructions, the range of a single instruction direct call is more restricted due to the limitations of the encoding. For instance in AArch64, a call-site can range from a single instruction (for short-range targets up to $\pm 128MiB$), e.g., `bl <target>`, up to three instructions (for long-range targets up to $\pm 4GiB$) as shown in Listing 1. The code in Listing 1 essentially initialises a register (`X16`) with the 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).

As a result, the ISA of an architecture defines both the sizes of the call-site and its potential patch. These sizes can be as small as a part of an instruction (patching an operand) or up to a few instructions. Due to the dominance of RISC ISAs on low-power architectures, call-sites for long-range targets come with an increased overhead while their patching is often more complicated.

Apart from the higher complexity of the patching logic itself due to the increased patch-size, multi-instruction patching comes with a number of additional implications. When patching call-sites, the managed runtime system treats the

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

Listing 1. AArch64 call-site example for a long-range target.

corresponding code segment as if it were data (see § 2). As a result, the implementation of the code performing the patching is also governed by the ISA. According to the ISA, the managed runtime system may be able to patch only a part of an instruction (e.g., half-word), a whole instruction, or even multiple instructions in an atomic manner. That said, if the ISA does not provide wide enough atomic write instructions to make the code modifications appear as an atomic instruction, then it is up to the managed runtime system to ensure that the patching will be performed in an atomic manner. Failing to do so may result in other threads observing a partially patched call-site and jumping to incorrect memory addresses. Atomic instructions are essentially both part of the ISA and the memory model specification. In the next subsection we discuss the constraints that might be imposed by the memory model.

3.2 Visibility and Timeliness | Memory Model

Since the managed runtime system treats the code segment being patched as data during patching, it is governed by the memory model and the guarantees it provides regarding data coherence. When patching call-sites on high-end systems, managed runtimes typically rely on atomic writes to ensure that the call-site patching is atomic and becomes immediately visible to all cores. This is possible due to the strong memory model that high-end processors typically implement. In low-power architectures however, weaker and more energy efficient memory models are usually implemented. As a result, on such architectures, SMC needs to request the appropriate memory barriers to ensure that patched call-sites will become visible in a timely manner. Furthermore, as illustrated in Figure 1, the code segment is part of two distinct streams, the code-stream and the data-stream. In high-end processors, due to the strong-memory models guarantees, both streams are kept in sync and coherent. Low-power architectures, however, usually separate the two streams and require explicit synchronisation to keep them coherent.

Listing 2 presents the AArch64 assembly code that needs to be executed on the core performing the code modifications. Similarly, Listing 3 presents the AArch64 assembly code that needs to be executed on the cores running the modified code to ensure that they will run the modified code and not an older or inconsistent version of it. Listing 2 essentially writes-back the modified cache-line from the data-cache up to the *Point of Unification* (line 1) – typically this is the lowest level component in the memory hierarchy that is shared among all cores – and then invalidates the modified cache-line in the instruction-cache (line 3). The `DSB ISH` instruction following the aforementioned cache operations (lines 2 and 4) are barriers that ensure that those operations have reached completion before continuing. On the other side of the synchronisation, the `ISB` instruction in Listing 3 ensures that the pipeline is empty before continuing. This essentially forces the processor to load instructions from the instruction cache or the main memory and not use a potentially stale copy from the pipeline buffers.

The aforementioned synchronisation, as expected, comes at the cost of performance overhead. As a result, the placement of such barriers creates a trade-off between how fast the patched code becomes 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 the performance and the correctness of the application. In the case of tiered compilation, performance can be negatively affected by failing to invoke the latest and highest-performing compiled method. In the case of de-optimisation, correctness may be affected by invoking an optimised version of the code that was based on an assumption that is no longer true.

3.3 Patchable Instructions | SMC Support

Ensuring atomicity and visibility when modifying code on the fly does not suffice. A third dimension that needs to be taken into consideration comprises restrictions explicitly imposed by the architecture. x86-64 allows programs to modify any instruction in the code-stream by overwriting with any other instruction. This however is not always the case. For instance, on AArch64, ARMv7, and Power only a limited set of instructions can be safely modified without explicit synchronisation. Namely for AArch64, these are `B`, `BL`, `BRK`, `HVC`, `ISB`, `NOP`, `SMC`, and `SVC` [5]. Note that in order for a code modification to be safe, both the old and the new instructions need to belong in that set. In any other case, *"Concurrent modification and execution of instructions can lead to the resulting instruction performing any behaviour that can be achieved by executing any sequence of instructions that can be executed from the same Exception level"* [5]. In the case of the Power ISA the set of instructions (referred as *patch class* in the manual [15]) that can be concurrently modified is even more strict, containing only direct branches and the no-op (`ori 0,0,0`) instruction.

For the rest of the instructions, the managed runtime system needs to ensure exclusive access to the corresponding instructions, which in the case of call-site patching can be only achieved by performing a "Stop-the-World" operation. "Stop-the-World" is essentially a pause of all application threads

```
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
```

Listing 2. AArch64 JIT compiler side synchronisation [5].

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

Listing 3. AArch64 call-site side synchronisation [5].

in order to allow for the managed runtime to perform operations that cannot be safely performed while the application is running. "Stop-the-World" pauses are mainly used by garbage collection algorithms to ensure that while the collection happens the state of the heap does not change by the application. In the case of call-site patching a "Stop-the-World" 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).

In the following section we examine the case of AArch64 and present a number of different possible call-site implementations, that allow safe patching.

4 Call-Site Implementations

In this section we present four different call-site implementations that enable safe patching without the need for a "Stop-the-World" pause even in architectures with weak SMC support. We present each implementation and discuss its HW support prerequisites. At the end of § 4.5 we also discuss patching with a "Stop-the-World" pause to provide a solution for architectures that do not offer any HW support for SMC. We base our implementations and discussion on the AArch64 architecture, which we find interesting due to its weak HW support for SMC. We do not, however, limit the discussion to the AArch64 architecture.

The AArch64 architecture [5] constrains the implementation of patchable call-sites by:

1. Supporting only short-range direct branches. This is a common characteristic of most RISC architectures due to the use of fixed size instructions, which inevitably puts a limit on the immediate operands that are used as the call offset in direct branches.
2. Supporting 64-bit atomic writes. Note that it can go up to 128-bit atomic writes using Load-Exclusive Pair (`LDXP`) and Store-Exclusive Pair (`STXP`) instructions, but requires the memory being modified to be 128-bit aligned. Most architectures provide at least word-sized atomic writes, and typically instructions are not bigger than a word. Regarding the alignment constrain for 128-bit atomic writes, it is worth noting that x86-64 also requires instructions to be word-aligned in order to be patchable.
3. Requiring explicit memory barriers (see § 3.2). The necessity of using memory barriers is also common on most low-power architectures, since to keep energy consumption low they avoid implementing strong memory models, rendering software responsible for performing some synchronisation operations.
4. Limiting the instructions that can be safely patched on-the-fly (see § 3.3). From our experience, the main limitation imposed by this constraint is that one can not swap direct branches (i.e. `B` and `BL`) with indirect branches (i.e. `BR` and `BLR`) and vice versa. This limitation is present on all three ISAs mentioned in § 3.3.

4.1 Direct Branching

Similarly to x86-64, AArch64 JIT compilers can use a single direct branch (`B` or `BL`) to implement call-sites as long as the target callee is in the range of the direct call. In the case of AArch64 the direct call range is ±128MiBs from the call-site. Since both `B` and `BL` are safe to modify concurrently with the execution, patching in this case is as simple as placing the encoded new direct branch in a register and storing it to the memory address of the call-site being patched. To ensure the visibility of the modified code on other cores, we need to execute the appropriate barriers as well (see Listing 2). The main advantage of this approach is its simplicity and performance, while its main disadvantage is the limited range of call targets. Although the ±128MiBs AArch64 branch range allows for a substantial managed code cache that may be sufficient for a range of applications, it is nonetheless a hard design limit in the absence of other solutions. Furthermore other contemporary 32–bit ISAs, such as RISC-V, have a more limited branch range of ±1MiB [12]. This constraint of direct branches warrants the investigation of complementary techniques.

4.2 Absolute-load Indirect Branching

Absolute-load indirect branching avoids modifying code, by loading the target address from a memory address and jumping to it. As a result, patching is done by overwriting the target address in the corresponding memory location without modifying the code-stream. In this case, explicit synchronisation might not be necessary depending on the cache coherency protocol of the underlying architecture. In the case of AArch64 no explicit synchronisation is required

```
1 MOVZ X16, #0xABCD ;Craft the address
2 MOVK X16, #0xEF89, lsl #16 ; holding
3 MOVK X16, #0x7654, lsl #32 ;     the
4 MOVK X16, #0x0213, lsl #48 ;  target
5 LDR  X16, [X16]
6 BLR  X16
```

Listing 4. AArch64 absolute-load indirect branching.

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

Listing 5. AArch64 relative-load indirect branching.

as the cache coherency protocol will ensure that any shared copies are invalidated and re-fetched from memory. Listing 4 shows the corresponding instructions for an absolute-load indirect branching call-site implementation. Lines 1–4 place the memory address holding the target address in register X16. Then in line 5 the target address is loaded in register X16 while in line 6 the branch is performed.

The implementation in Listing 4 allows the managed runtime system to keep the target address in any memory address in the system. This way, each method can be mapped to a memory address holding the location of its compiled version. This approach not only simplifies patching, by not requiring code patching, but it also saves the managed runtime system from detecting all the potential call-sites targeting the method at hand. Unfortunately, however, this simplicity comes at the cost of increased number of instructions, due to the inline crafting (Lines 1–4) of the address. In the case of AArch64 this can be up to 4 instructions per call-site (some values can be crafted with fewer instructions), which is in the common path. Note that patching is expected to occur less often than calling a method.

4.3 Relative-load Indirect Branching

Taking advantage of PC-relative addressing we can optimise the absolute-load indirect branching approach (§ 4.2) by storing the target address in a PC-relative address, e.g., after the end of the callers compiled code. Listing 5 shows the corresponding instructions for the relative-load indirect branching call-site implementation. The first lines illustrate an array of PC-relative call-site targets, one for each of the callers target methods. Note that the labels are only placed to improve readability, the JIT compiler can directly place the corresponding offset when generating the LDR instruction.

```
1 CALLEE_1: .quad 0x0123456789ABCDEF
2             ...
3 CALLEE_N: .quad 0x01234ABCDEF56789
4 START:    ...
5           NOP
6           BL SHORT_TARGET
```

Listing 6. Patchable direct branch relying on relative-load indirect branching for long range calls. Unsafe on AArch64.

```
1 L: LDR X16, CALLEE
2    BR X16 ; Don't link
3    CALLEE: .quad 0x0123456789ABCDEF
4    ...
5    BL SHORT_TARGET ; or L
```

Listing 7. Trampoline implementation with relative-load call-sites.

Non-AArch64-compatible approach: Note that in architectures without the constraint 4 of AArch64, about which instructions can be safely patched, the relative-load indirect branching can be combined with direct branching to improve performance of short range calls (by using direct calls). In such a scenario the JIT compiler would use a NOP followed by a BL for short range calls. When patching for a long range call, the runtime would first write the new target to CALLEE_1 patch NOP to LDR X16, CALLEE_1 and then patch BL SHORT_TARGET to BLR X16. When patching from a long range call to a short range the opposite would happen, patching first the BLR X16 to BL SHORT_TARGET and then LDR X16, CALLEE_1 to NOP.

4.4 Trampolines

Another approach for safe patching of call-sites is to use trampolines. A trampoline is essentially an instruction that jumps to a call-site that eventually calls the desired callee. Trampolines are implemented using a single BL instruction, which can be safely patched without exclusive access. The trampoline can either branch directly to the target method (if it is within the range of a direct branch), or branches to an out-of-line long-range call-site, see Listing 7. Note that the call sites just branch without linking (line 2), this way when the callee returns it does not return to the out-of-line call-site but right after the trampoline (line 5). This approach is currently being used by the OpenJDK AArch64 port. A limitation of this approach is that the out-of-line call-site still needs to be in the direct branch range. This can be achieved by reserving space for a call-site per callee in the method header, in a similar manner to the way we allocate space for target addresses for relative-load indirect branching.

4.5 Patching only at Safe-points

Managed runtime systems use safe-points to "stop the world" and perform critical operations, such as garbage collection, that are not safe to be performed while application threads are running. Safe-point checks are usually injected by the JIT compiler at back-edges of branches and right after (or before) call-sites. Whenever a critical operation needs to be performed, the managed runtime system enables the safe-point flag forcing the application threads to pause execution the next time they reach a safe-point. An early version of the AArch64 OpenJDK port [13] employed this technique to obviate the complexities of call-site patching imposed by the architecture. By employing call-site patching at safe-points it is ensured that only the patching thread will have access to the corresponding call-site. That said, JIT compilers can use **any** instructions (even from the AArch64 ISA) to generate the call-sites in this case. A possible call-site implementation for this approach would be that of Listing 1. In this approach patching is achieved by overwriting the call-site as a whole or partially, depending on the chosen call-site implementation. When patching only at safe-points the application threads can be safely synchronized when notified that the safe-point, and hence the patching, is complete. The major advantage of this approach is that it gives the flexibility to the managed runtime system to use any instruction sequence to implement the call-site. However, it requires a stop-the-world pause which might not be desirable, especially in interactive or latency-critical applications.

5 Comparison of Call-Site Implementations

In this section we perform a comparison between the different call-site implementations that we present in § 4. Table 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.

5.1 Code vs Data Patching and SMC Support

We characterise each implementation based on whether patching needs to alter code or data. Implementations that require code patching, e.g. Direct (§ 4.1), depend on SMC support and cannot be implemented in the absence of it. On the other hand, implementations requiring only data patching, e.g. Absolute-load Indirect (§ 4.2), do not require SMC support since they do not alter the code itself, they just indirectly redirect execution. Note that Relative-load Indirect (§ 4.3) despite writing to memory holding code it does not overwrite actual code. It just overwrites data which reside with code, thus we characterise it as *Hybrid*, and it does not require SMC support. Similarly we characterise Trampolines (§ 4.4) as *Hybrid* since they also write data in memory holding code for the long-range call-sites. In contrast to Relative-load Indirect, Trampolines require SMC support to patch short-range calls, and to patch short-range calls to long-range and vice versa. Depending on the underlying hardware Hybrid implementations may or may not have side-effects on the code-stream (see Figure 1 1 and 3).

5.2 Code Size

Code size is an important metric not only because bigger code size is associated with more instructions and thus more cycles, but also because bigger code sizes impact the cache performance, especially in embedded systems. For each implementation we report the number of 32-bit instructions. Note that Relative-load Indirect is once more *special*. Despite requiring only two instructions per call-site, it also requires a 64-bit segment (marked as +2 in Table 1) per callee per compiled method. These 64 bits reside inline with the code and hold the target address of the corresponding call-sites. This additional space in the method body contributes to the overall code size and may impact the performance of the code cache. Trampolines depending on the call-range require executing from a single instruction (for short-range calls) up to three instructions (for long-range calls). For short-range the reserved space is four instructions, while for long-range the reserved space is two instructions (marked as +4 and +2 respectively in Table 1).

5.3 Patching Complexity

The complexity of the patching code, although not in 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 enough to make future calls jump to the new target. However, different implementations have different complexity. In the Absolute-load Indirect branching (§ 4.2) 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 (§ 4.3) 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 $O(m)$, where m is the number of call-sites that invoke the corresponding method and $m \geq n$. To effectively

Table 1. Comparison of call-site implementation approaches.

	Direct	Absolute-load Indirect	Relative-load Indirect	Trampolines	Any | Safe-point
Characterisation	Code	Data	Hybrid	Hybrid	Any
Requires SMC support	Yes	No	No	Yes	Maybe
Size (in 32-bit instructions)	1	6	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

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 to make it lazily patch call-sites when they get executed. The latter allows patching to run in constant time, but penalizes the first execution of each un-patched call-site.

6 Evaluation

6.1 Platform and Methodology

Hardware & Software: To evaluate the different call-site implementations we run our experiments on an ARM-based Odroid-C2. The Odroid-C2 features a quad-core Cortex-A53 processor clocked at 1.54 GHz and 2 GB of DDR3 RAM. The Cortex-A53 is an "8-stage pipelined processor with 2-way superscalar, in-order execution pipeline, a 4 KiB conditional branch predictor, and a 256-entry indirect branch predictor[2]". The software configuration consists of Ubuntu 18.04.2 LTS with the odroid 3.16.68-41 kernel, GCC 8.3.0, and MaxineVM 2.8.0 built with Open JDK 8 u212.

We evaluate the presented call-site implementations on a microbenchmark and the DaCapo benchmark suite [8]. Our discussion of cache effects does not consider non-uniform cache line sizes, such as those on some implementations of big.LITTLE configurations.

Microbenchmark: In order to perform a fine-grained micro-architectural and performance analysis of the call-site implementations presented in § 4, we implemented a micro-benchmark that performs a predefined number of invocations, where each callee performs just a return instruction. We use a total of two callees, one is used as the default target and the other as the patching target. For each call-site implementation we generate the corresponding sequence of assembly instructions. In addition, the microbenchmark allows us to generate inline call-sites and measure their performance in a cycle accurate manner. Finally, we can also calculate the cost of the patching code per call-site implementation by isolating its execution.

[2] https://en.wikipedia.org/wiki/ARM_Cortex-A53#Overview

DaCapo Suite: For the DaCapo benchmark suite (version 9.12-bach), we implemented and evaluated the two highest performing call-site implementations in MaxineVM [19, 28]. We chose Maxine VM as our experimental JVM following on directly from our experience of porting it to Arm architectures [19], and our search for an optimal call-site strategy. Namely, in MaxineVM we evaluate against DaCapo the `Direct`, and a modified version of `Relative-load Indirect`. We run nine out of the fourteen DaCapo benchmarks, namely: `avrora`, `fop`, `h2`, `jython`, `luindex`, `lusearch`, `pmd`, `sunflow`, and `xalan`. The remainder five either do not run due to limitations of the bundled Java platform (`batik`, `eclipse`, and `tomcat`), or fail to start in MaxineVM (`tradesoap` and `tradebeans`). For each run we set the heap size to 1GB (`-Xms1G -Xmx1G`). We run each benchmark 20 times (using the DaCapo harness) and report the average of the runs after reaching peak-performance.

Methodology: For the micro-benchmark we perform 100 runs for each configuration and report the average measurements. When plotting we use bar-plots with 95% confidence intervals, to indicate the variance across the runs. On the Odroid-C2 we obtain the measurements by directly accessing the performance monitor unit (PMU) with Table 2 listing the hardware counters that we read when measuring. After analysing the obtained performance counters, we report those that directly influence and contribute to the performance results. In addition, we disable dynamic voltage and frequency scaling (DVFS) by setting each core to work at the highest possible frequency using the `cpufreq-set` utility and the *userspace* governor.

6.2 Microbenchmark Results

Figure 2 and Table 3 summarise the results obtained by running the microbenchmark on the Odroid-C2. Note that for `Trampolines` we only evaluate the performance of long-range calls since the performance of short-range calls is equivalent to that of the `Direct` implementation. Note also that the number of cycles represents calling the "return" function, i.e. corresponds to the call-site plus the `ret` instruction.

Call Cost (Best/Worst Case): Figure 2 illustrates the number of cycles spent for each call implementation and the `ret`

Table 2. PMU hardware counters used in our experiments and evaluation. We mark with bold the counters used for generating Figure 2 and Table 3.

Name	Event ID	Description
CPU_CYCLES	**0x11**	**Cycle count**
L1I_CACHE_REFILL	**0x01**	**L1 Instruction Cache Refills**
L1I_CACHE	0x14	L1 Instruction Cache Accesses
L1D_CACHE_REFILL	**0x03**	**L1 Data Cache Refills**
L1D_CACHE	0x04	L1 Data Cache Accesses
L2D_CACHE_REFILL	0x17	L2 Data Cache Refills
L2D_CACHE	0x16	L2 Data Cache Accesses
PC_WRITE_RETIRED	0x0c	Branches (Including Returns)
BR_IMMED_RETIRED	0x0d	Immediate Branches
BR_MIS_PRED	0x10	Branch Misses
BR_PRED	0x12	Predictable Branches
BR_INDIRECT_SPEC	0x7a	Speculatively Executed Indirect Branches
INST_RETIRED	0x08	Retired instructions

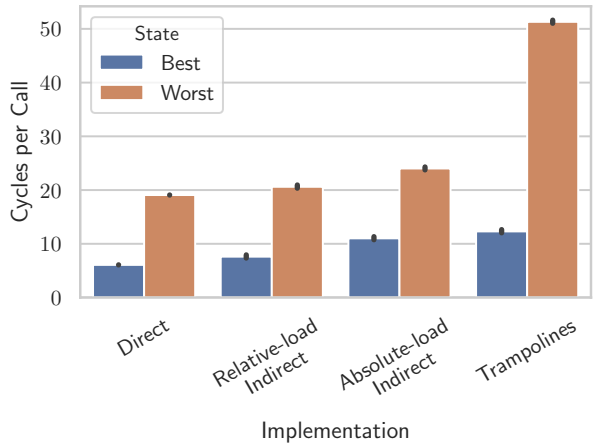

Figure 2. Micro-Benchmark results from Odroid-C2.

instruction. As shown, both the best and worst-case costs are presented. Regarding the best case, this is the cost that each particular call-site implementation has when the memory accesses hit on the L1 caches. On the contrary, the worst-case refers to the first method call invocation directly after patching, in the case where the patching has caused shared cache line invalidations. In the case of `Trampolines` the worst-case may include up to two instruction-cache and one data-cache miss, due to the number and placement of instructions that might happen to be on different cache-lines. In the case of `Direct` the worst-case includes a single instruction-cache miss and in the case of data-patching implementations (`Absolute-load Indirect` and `Relative-load Indirect`) the costs include one data-cache miss.

In the best-case the `Direct` branch implementation outperforms all others, while the `Trampolines` is the worst performing (6 versus 12 cycles). The two data-patching implementations fall in-between with 7 (`Relative-load Indirect`) and 11 (`Absolute-load Indirect`) cycles. The same trends are also observed in the worst-case with the various call-site implementations ranging from 18 cycles for `Direct` to 50 cycles for `Trampolines`. When applications reach their peak performance state, the call-site costs will be equivalent to the best-case presented.

Patching Cost: Table 3 summarises the cost of performing a call-site patch for each call-site implementation. The first row shows the number of cycles spent to execute the patch logic. Note that these numbers are indicative since different runtime systems are expected to require different number of cycles for the patching. For instance, in our microbenchmark we use the return register and calculate the memory address that needs to be patched. In a realistic runtime system we would have to traverse some data-structure, or even the code itself to find the addresses that need to be patched. Nevertheless, the core functionality of the patching would be the same and the order between the various implementation is expected to remain the same, i.e. we expect the `Absolute-load Indirect` implementation to always have the fastest patching implementation, and the `Trampolines` implementation to have the slowest. The additional overhead of instruction-cache management operations due to explicit instruction modification can be seen in the `Direct` and `Trampoline` call-site implementations whose patching costs more cycles in comparison to that of the `Absolute-load Indirect` and `Relative-load Indirect` implementations that require only data cache management operations.

Level 1 Instruction Cache Refills: The second row of Table 3 presents the total number of potential L1 instruction cache refills expected as a result of a patch operation for each implementation. Data-patching implementations are

Table 3. Comparison of patching call-site implementation.

	Direct	Absolute-load Indirect	Relative-load Indirect	Trampolines
Cycles	45	12	24	56
L1I-refills on caller's site	1	0	0	0–2
L1D-refills on caller's site	0	1	1	0–1

not expected to result in any instruction-cache refills since they do not alter code, nor do they perform any explicit instruction-cache management operations. On the other hand code-patching implementations, namely `Direct` and `Trampoline`, may cause instruction-cache misses. The patching of a `Direct` call-site may cause one instruction-cache miss if the patched instruction needs to be re-fetched in order to be executed. The patching of a `Trampoline` call-site is more complex, though. Depending on the patch operation, we may observe from zero up to two instruction cache misses (due to the patching). When patching a long-range call-site to a long-range call-site, patching a `Trampoline` call-site is essentially identical to patching an `Relative-load Indirect` call-site. Consequently no explicit instruction cache management are required in the patching process, and no refill is expected to happen. On the contrary, when patching a short-range call to perform long-range call, we may observe instruction cache refills. The first instruction refill is needed to fetch the newly patched trampoline branch, and the second one is needed to fetch the out-of-line call-site that will actually perform the long-range call. Finally, when patching either a short-range or a long-range call to perform a short-range call, then only a single instruction-cache refill is expected; the one required to fetch the patched inline direct branch.

Level 1 Data Cache Refills: The third row of Table 3 presents the total number of possible L1 data cache refills expected as a result of a patch operation for each implementation. In this case, data-patching implementations could cause one cache-miss when a caller loads the patched target address. Patching a `Direct` call-site is not expected to result in any data-cache refill, since direct branches do not perform any reads/loads. On the contrary, in the case of a `Trampoline` call-site if the new target is long-range one, then after performing a direct branch (for the trampoline) the caller will need to load the target address from memory, resulting in data-cache refill to serve that load.

Both Level 1 instruction and data cache refills discussed above, assume that the caller is running on a different core than the patcher. Otherwise, in the case that the patcher is sharing the level 1 cache with the caller, the caller does not need to perform any refills, all the data will be already available to it.

```
1 ADR X17, CALL ; Get address of BLR
2 LDR  X16, OFFSET     ; Load offset
3 ADD X16, X16, X17    ; Add them
4 B #8     ; Jump over inline offset
5 OFFSET: .int CALL - CALLEE_1
6 CALL:   BLR  X16
```

Listing 8. Indirect-Maxine: MaxineVM version of Relative-load Indirect branching.

6.3 DaCapo Results

After conducting the micro-benchmark studies, we implemented versions of the presented call-site implementations in MaxineVM in order to assess their real-world performance. From the call-site implementations described in § 4, we opted for implementing the fastest ones as demonstrated by the micro-benchmark; namely `Direct` and `Relative-load Indirect`. Regarding the `Relative-load Indirect` implementation shown in Listing 5, we slightly modified its implementation in MaxineVM due to its meta-circular nature. In this scheme, we encode the target address of the different callees inside the methods. However, this is not possible in MaxineVM where due to its meta-circular nature the methods that comprise the bootimage do not have known target addresses. Since MaxineVM is written in Java and compiled by its own compiler (C1X), the methods that are part of the bootimage must have relative offsets rather than absolute addresses in order for the VM to be portable. A potential solution would be to create the boot image with relative offsets and later patch it to use absolute addresses at load time, however this entails significant effort that is out of the scope of this paper. Other meta-circular VMs such as Jikes RVM [1] allocate the bootimage on predefined memory addresses and hence could use the `Relative-load Indirect` scheme of Listing 5 as is.

Listing 8 shows the version of `Relative-load Indirect` call-site that was prototyped on the Maxine VM. In addition to the above constraint, we placed the target offset inside the call site for simplicity, which enabled a prototype to be created without making significant changes to the runtime. Ideally the offset would be moved out-of-line, as in Listing 5, to avoid incurring the additional branch on line 4. Line 1 gets the address of the `BLR` in line 6. Line 2 loads the offset stored inside the code (in line 5) using PC-relative addressing. Line

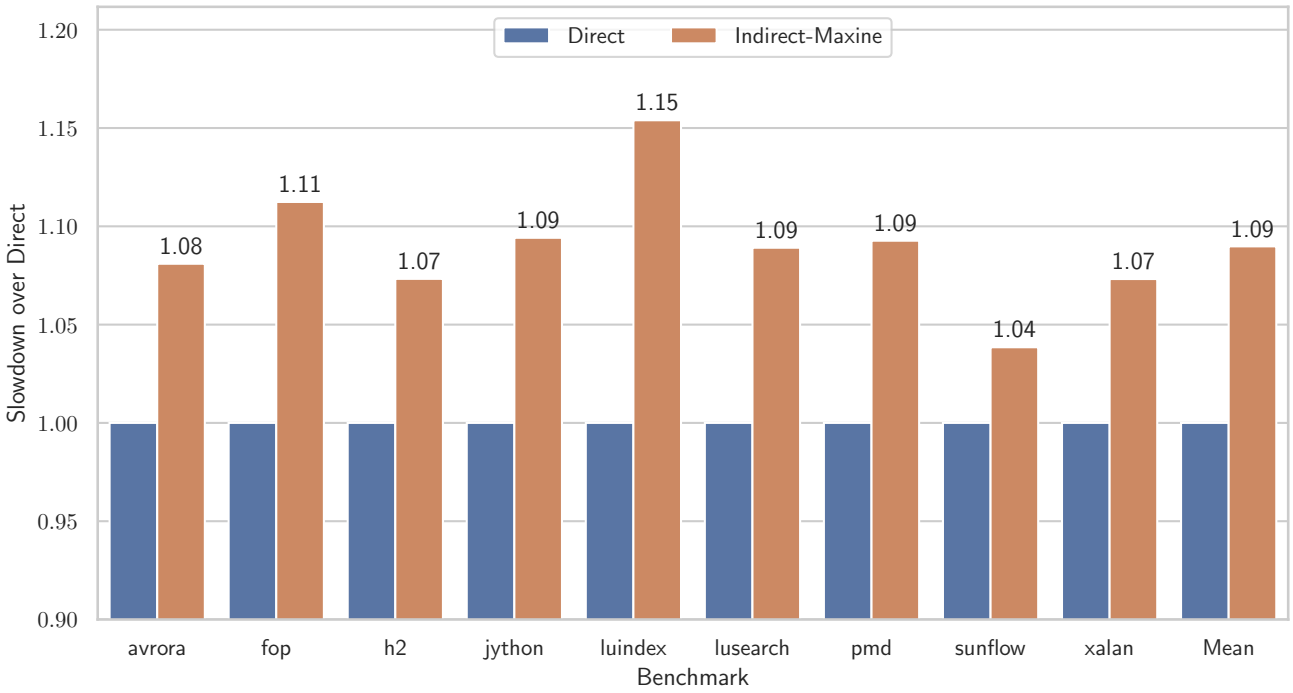

Figure 3. DaCapo benchmark results from Odroid-C2.

3 adds the offset to the BLR address. Line 4 jumps over the inlined offset, and finally Line 6 branches to the target. To estimate the relative cost of this sub-optimal implementation, we also evaluate the `Relative-load Indirect` implementation of MaxineVM (`Indirect-Maxine`) against the original four implementations with the micro-benchmark. As shown in Figure 4, despite being a little slower, the `Indirect-Maxine` implementation has a roughly similar performance to the `Absolute-load Indirect`.

As shown in Figure 3, the `Indirect-Maxine` implementation performs notably worse than the `Direct` branch implementation with slowdowns varying from 4% (sunflow) to 15% (luindex). Of course the speedup that direct branches have over indirect come at the cost of limited branch range as explained in § 4. However, in non-metacircular virtual machines we expect the performance of indirect branches to be better if we apply the `Relative-load Indirect` implementation (see Figure 4).

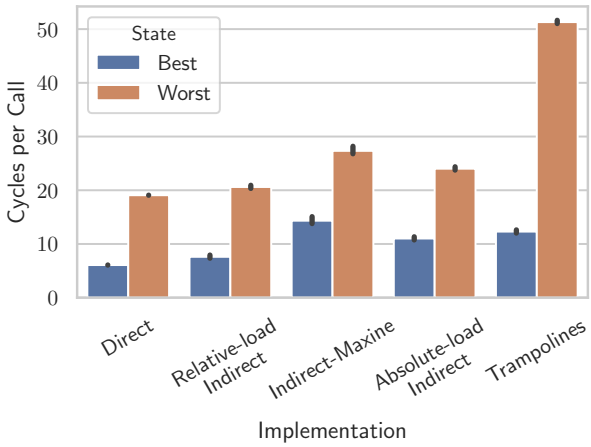

Figure 4. Micro-benchmark results from Odroid-C2 including Maxine's variation of Indirect-Relative.

7 Related Work

The majority of prior studies has focused on detecting [10], modelling [3], analysing [4], and verifying [9, 23] SMC. SMC is of main interest in the area of security where it can be potentially utilised to either improve the security of systems [2, 14, 27], or to obfuscate code in order to trick malware detectors [11, 24] or to prevent reverse-engineering [21]. In managed languages, in particular, SMC is part of the JIT tiered compilation employed for increased performance. 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 [20, 25, 26], to the best of our knowledge, no prior work exists on characterising the performance implications of SMC in the context of managed language runtimes. Our work aims at providing the first in-depth characterisation of alternative memory-safe call-site implementations.

8 Conclusions & Future Work

In this paper we explored how managed runtime systems can perform method calls and safe code patching in the absence of hardware support for SMC. After discussing the implications that the lack of hardware support for SMC has on low-power architectures such as AArch64, we performed an analysis over alternative call-site implementations and patching strategies, highlighting their advantages and disadvantages. Consequently, we evaluated four alternative implementations along with their associated patching strategies on a microbenchmark. Furthermore we evaluated the two most promising strategies on the DaCapo benchmark suite in the context of the open-source MaxineVM running on AArch64. Our low-level architectural characterisation and performance analysis showcased a performance variation of up to 15% between the different implementations.

For future work, we plan to extend this study to more architectures that are more constrained with respect to branch instruction encodings such as RISC-V. Furthermore, we plan to explore additional patching strategies such as polymorphic inline caches, and study safe patching at natural synchronisation points of managed runtimes, such as safe-points.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback and Lubomír Bulej for his guidance while preparing the camera-ready version. This work is partially supported by an ARM iCASE scholarship and the EU Horizon 2020 ACTiCLOUD 732366 and E2Data 780245 programme grants. Mikel Luján is supported by an Arm/RAEng Research Chair award and is a Royal Society Wolfson Fellow.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J. . Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The JalapeÃśo virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238. https://doi.org/10.1147/sj.391.0211

[2] Antoine Amarilli, Sascha Müller, David Naccache, Daniel Page, Pablo Rauzy, and Michael Tunstall. 2011. Can Code Polymorphism Limit Information Leakage?. In *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication*, Claudio A. Ardagna and Jianying Zhou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–21.

[3] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. 2007. A Model for Self-Modifying Code. In *Information Hiding*, Jan L. Camenisch, Christian S. Collberg, Neil F. Johnson, and Phil Sallee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 232–248.

[4] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. 2007. A Model for Self-modifying Code. In *Proceedings of the 8th International Conference on Information Hiding (IH'06)*. Springer-Verlag, Berlin, Heidelberg, 232–248. http://dl.acm.org/citation.cfm?id=1759048.1759066

[5] ARM. 2019. Arm Architecture Reference Manual Armv8. https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf?_ga=2.153168888.1503102752.1563189579-1528906649.1556140495

[6] ARM. 2019. Neoverse E1. https://www.arm.com/products/silicon-ip-cpu/neoverse/neoverse-e1

[7] Azul Systems. 2019. Zulu Embedded Open Source Java for Embedded Systems. https://www.azul.com/products/zulu-embedded/

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press.

[9] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. 2007. Certified Self-modifying Code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 66–77. https://doi.org/10.1145/1250734.1250743

[10] Shi Dawei, Lv Delong, and Ye Zhibin. 2018. Dynamic Self-modifying Code Detection Based on Backward Analysis. In *Proceedings of the 2018 10th International Conference on Computer and Automation Engineering (ICCAE 2018)*. ACM, New York, NY, USA, 199–204. https://doi.org/10.1145/3192975.3193016

[11] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. 2019. 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)*. ACM, New York, NY, USA, 15–27. https://doi.org/10.1145/3321705.3329819

[12] RISC-V Foundation. 2019. The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20190608-Base-Ratified. https://riscv.org/specifications/

[13] Andrew Haley and Andrew Dinn. FOSDEM 2014. OpenJDK on AArch64 Update. https://aph.fedorapeople.org/Aarch64-fosdem-2014.pdf

[14] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2013. Librando: transparent code randomization for just-in-time compilers. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS '13)*. ACM, New York, NY, USA, 993–1004. https://doi.org/10.1145/2508859.2516675

[15] IBM. 2017. Power ISA Version 3.0B. https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0

[16] Intel. 2015. Method and apparatus for providing hardware support for self-modifying code, PCT/US2015/030411. https://patents.google.com/patent/EP3143496A1/en

[17] Intelm. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[18] JamaicaVM. 2019. A hard realtime Java bytecode-based Virtual Machine. https://www.aicas.com/cms/en/JamaicaVM

[19] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. 2017. 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). ACM, New York, NY, USA, 74–82. https://doi.org/10.1145/3050748.3050764

[20] Dumitrel Loghin, Bogdan Marius Tudor, Hao Zhang, Beng Chin Ooi, and Yong Meng Teo. 2015. A Performance Study of Big Data on Small Nodes. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 762–773. https://doi.org/10.14778/2752939.2752945

[21] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. 2011. A taxonomy of self-modifying code for obfuscation. *Computers & Security* 30, 8 (2011), 679 – 691. https://doi.org/10.1016/j.cose.2011.08.007

[22] MicroPython. 2019. Python for microcontrollers. https://micropython.org

[23] Magnus O. Myreen. 2010. Verified Just-in-time Compiler on x86. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 107–118. https://doi.org/10.1145/1706299.1706313

[24] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *Proceedings of the Seventh European Workshop on System Security (EuroSec '14)*. ACM, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/2592791.2592796

[25] Nikola Rajovic, Lluis Vilanova, Carlos Villavieja, Nikola Puzovic, and Alex Ramirez. 2013. The low power architecture approach towards exascale computing. *Journal of Computational Science* 4, 6 (2013), 439 – 443. https://doi.org/10.1016/j.jocs.2013.01.002 Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011.

[26] Bogdan Marius Tudor and Yong Meng Teo. 2013. 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)*. ACM, New York, NY, USA, 267–278. https://doi.org/10.1145/2465529.2465553

[27] Nicolas Veyrat-Charvillon, Marcel Medwed, Stéphanie Kerckhof, and François-Xavier Standaert. 2012. Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note. In *Advances in Cryptology – ASIACRYPT 2012*, Xiaoyun Wang and Kazue Sako (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 740–757.

[28] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.* 9, 4, Article 30 (Jan. 2013), 24 pages. https://doi.org/10.1145/2400682.2400689