Evaluation of dynamic binary translation techniques for full system virtualisation on ARMv7-A

Niels Penneman[a,*], Danielius Kudinskas[b], Alasdair Rawsthorne[b], Bjorn De Sutter[a], Koen De Bosschere[a]

[a]*Computer Systems Lab, Ghent University, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium*
[b]*School of Computer Science, The University of Manchester, Oxford Road, Manchester M13 9PL, UK*

Abstract

We present the STAR hypervisor, the first open source software-only hypervisor for the ARMv7-A architecture that offers full system virtualisation using dynamic binary translation (DBT). We analyse techniques for user-space DBT on ARM and propose several solutions to adapt them to full system virtualisation. We evaluate the performance of a naive configuration of our hypervisor on a real embedded hardware platform and propose techniques to reduce DBT-based overhead. We analyse the impact of our optimisations using both micro-benchmarks and real applications. While the naive version of our hypervisor can get several times slower than native, our optimisations bring down the run-time overhead of real applications to at most 16%.

Keywords: Binary translation, Hypervisor, Instruction set architecture, Virtualisation, Virtual machine monitor

1. Introduction

The term virtualisation is used in many different contexts, ranging from storage and network technologies to execution environments. In this paper we discuss system virtualisation—a technology that enables multiple guest operating systems to be executed on the same physical machine simultaneously by isolating the physical machine from the operating systems by means of a hypervisor.

While system virtualisation in the server world has already matured, it is still an area of ongoing research for embedded systems [4, 21, 25]. Virtualisation solutions for data centres and desktop computers cannot be readily applied to embedded systems because of differences in requirements, use cases, and computer architecture. ARM is by far the leading architecture in the embedded and mobile market [45], and over the past 8 years multiple virtualisation solutions have been developed for it.

Most virtualisation efforts for ARM started with paravirtualisation. This is not surprising, because the architecture is not classically virtualisable as shown by our previous work [39]. Paravirtualisation has many drawbacks [14], which can all be solved by full virtualisation. However, full virtualisation requires special hardware support, i.e. classic virtualisability, or special software support, such as dynamically rewriting the code of a guest at run time, a technique known as dynamic binary translation (DBT).

ARM has recently extended its ARMv7-A architecture with hardware support for full virtualisation [3]. While hypervisors built for these extensions easily outperform DBT-based hypervisors, they do not run on older hardware. Furthermore, DBT remains useful in a handful of scenarios, ranging from full-system

*Corresponding author. Tel.: +32 (0)51 303045
Email addresses: Niels.Penneman@elis.UGent.be (Niels Penneman), Danielius.Kudinskas@gmail.com (Danielius Kudinskas), Alasdair.Rawsthorne@manchester.ac.uk (Alasdair Rawsthorne), Bjorn.DeSutter@elis.UGent.be (Bjorn De Sutter), Koen.DeBosschere@elis.UGent.be (Koen De Bosschere)

instrumentation to legacy system software emulation. It is therefore useful to study DBT to virtualise the ARMv7-A architecture. There is, however, little related work for DBT on the ARM architecture in the context of full system virtualisation. Most research focuses on DBT of user-space applications, but not all user-space techniques apply to full system virtualisation.

This paper makes the following contributions:

- We present the STAR hypervisor, the first open source software-only hypervisor for the ARMv7-A architecture that offers full virtualisation using DBT.
- We analyse existing techniques for user-space DBT on ARM and propose several solutions to adapt them to full system virtualisation.
- We analyse the performance of a naive configuration of our hypervisor and propose techniques to reduce the overhead of a DBT engine specific to full system virtualisation on ARM.
- We provide an evaluation of all our techniques on a real embedded hardware platform.

The complete source code of the presented hypervisor is available at `http://star.elis.ugent.be`.

The rest of this paper is organised as follows: Section 2 provides background and discusses problems with existing virtualisation solutions for the ARM architecture. It also discusses why existing user-space DBT techniques for the ARM architecture cannot be readily applied to full system virtualisation. Section 3 provides a brief overview of our hypervisor. In Section 4, we propose techniques specific to system-level DBT for dealing with ARM's exposed program counter (PC). We analyse the performance of a naive version of our hypervisor in Section 5 to classify the run-time overhead of our DBT engine, focusing specifically on issues with full system virtualisation, and we propose several optimisations. In Section 6, we evaluate how our techniques affect the run-time overhead of the DBT engine on an embedded hardware platform.

2. Background and related work

2.1. Hypervisors for the ARM architecture

Many different virtualisation solutions for the ARM architecture have been designed over the past 8 years. Early designs such as ARMvisor [19], B-Labs Codezero Embedded Hypervisor [6], KVM for ARM [15], TRANGO Virtual Processors Hypervisor [49], NEC VIRTUS [32], VirtualLogix VLX [5, 51], VMware MVP [8] and Xen ARM PV [30, 35] all use paravirtualisation, as their development predates the introduction of ARM's hardware support for full virtualisation. Such hypervisors have many drawbacks [14]: as paravirtualisation hypervisors present their own, custom interface to guests, each guest must be adapted for each specific hypervisor. Since none of the efforts to standardise such interfaces has spread to more than a few operating systems or hypervisors [2, 36, 42], the majority of operating systems does not support such interfaces out of the box. Moreover, the development, maintenance and testing of patches for guests is tedious and costly. Furthermore, licensing may prevent or restrict modifications to operating system source code, and often imposes rules on the distribution of patch sets or patched code.

KVM for ARM and Xen ARM PV were superseded by KVM/ARM and a new Xen port in which paravirtualisation support was dropped in favour of ARM's hardware support [16, 17, 53]. VirtualLogix VLX was used by Red Bend software as the basis for vLogix Mobile, which now also uses ARM's hardware support instead of paravirtualisation [41]. Some hypervisors support both paravirtualisation and full virtualisation, such as PikeOS [34, 48], OKL4 [26, 50] and Xvisor [54]. PikeOS and OKL4 are commercial microkernel-based hypervisors specifically designed for real-time systems virtualisation. They rely on paravirtualisation for real-time scheduling decisions.

Hypervisors built for ARM's hardware extensions easily outperform DBT-based hypervisors, but do not run on older hardware. Furthermore, DBT remains useful by virtue of its versatility [39]. Virtualisation is often considered as a solution to address software portability issues across different architectures, but hardware extensions merely recreate this problem at a different level. DBT can transparently enable a multitude of other virtualisation usage scenarios such as legacy system emulation and optimisation [13, 20, 24], full system instrumentation and testing [11], optimisations across the border between operating system kernels and applications [1], and even load balancing in heterogeneous multi-core systems [31, 52].

The ITRI Hypervisor presents an interesting case as it uses a combination of static binary translation (SBT) for CPU virtualisation and paravirtualisation for memory management unit (MMU) virtualisation [44]. SBT can be regarded as paravirtualisation, although it does not require the sources of the guest kernels and hence aims to be a more generic approach. Any benefits of SBT over traditional paravirtualisation are however lost due to MMU paravirtualisation. Furthermore, SBT cannot support self-modifying code, and complicates dynamically loading code at run time, as is often done in operating system kernels for device drivers and other kernel modules. The ITRI Hypervisor therefore only supports non-modular kernels.

SBT on ARM is hard, because it is common for data to be embedded within code sections, and code may use a mix of 32-bit ARM and variable-width Thumb-2 instructions in which the encoding of each byte can only be deduced by control flow analysis. In the absence of symbolic debug information, extensive analyses and heuristics are required to distinguish code from data [12, 28], and any uncertainties must be dealt with at run time using a combination of interpretation and DBT. There is however no mention of such techniques in any of the ITRI Hypervisor documents [43, 44]. Relying on the availability of symbolic debug information makes SBT-based virtualisation no better than traditional paravirtualisation: symbolic debug information is often not available for commercial software, and its representation may vary across operating systems, so that a translator must support each of them.

2.2. Classic virtualisability

Popek and Goldberg [40] formally derived sufficient (but not necessary) criteria that determine whether an architecture is suitable for full virtualisation, currently referred to as classic virtualisability. They made an abstract model of a computer architecture with two privilege levels, and a classification of instructions based on their interactions with the state of the architecture.

They define *privileged instructions* as instructions that trap when executed in the unprivileged mode, but not in the privileged mode. They define *sensitive instructions* as instructions whose behaviour depends on the mode and configuration of resources in the system, or whose behaviour is to alter the mode and/or configuration. They proved that an "efficient" hypervisor can be constructed if all sensitive instructions are also privileged. Such a hypervisor is known as a trap-and-emulate hypervisor, which gets its efficiency from being able to run all original user-space guest software as is, with hypervisor interventions only needed when guest kernel code is executed.

In our previous work [39], we have extended Popek and Goldberg's architectural model to contemporary architectures, and we have shown that the ARMv7-A architecture is not classically virtualisable. We have used our extended model to analyse the feasibility of a DBT-based hypervisor for the ARMv7-A architecture, and studied how DBT affects the behaviour of instructions that are otherwise innocuous. We have used our results to construct the DBT engine of the hypervisor presented in this paper.

2.3. DBT on ARM

DBT engines translate short sequences of binary instructions as needed during the execution of a program. The translated code is stored so that it can be reused. Figure 1 depicts the operational cycle of a generic DBT engine.

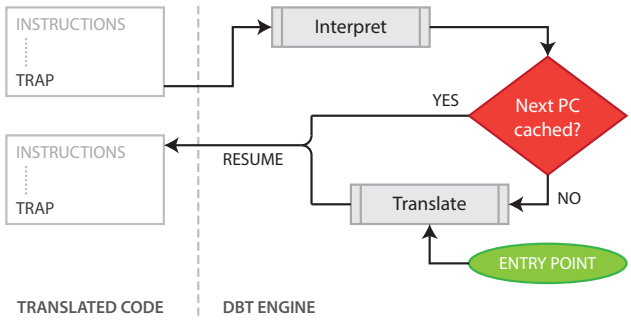

Figure 1: Operational cycle of a generic dynamic binary translation engine

The DBT engine starts by translating the first block of instructions of a program. The translator finalises a block when it encounters a control flow instruction. Such instructions are translated into a trap to the DBT engine. The translated code is then executed until the trap. The trap is handled by emulating the original control flow instruction, after which the next address is known from which translation must continue. Next, the DBT engine checks if the code at the target address has already been translated, and if not, it invokes the translator. It then resumes executing the translated code. A DBT engine used for full system virtualisation will also require sensitive instructions to be replaced and interpreted in addition to control flow instructions.

QEMU [9] and Varmosa [29], a product of a VMware-funded MIT project, are the only hypervisors capable of fully virtualising the ARM architecture by means of DBT. They therefore do not depend on any kind of hardware support for full virtualisation. QEMU is a hosted hypervisor which runs in user space. Its DBT engine translates all instructions to an intermediate representation, and then to the instruction set of the host machine (e.g., x86), such that it can perform cross-architecture virtualisation. Because it runs in user space, its design is fundamentally different from the solution presented in this paper: it cannot benefit from privilege separation and it can hence not enforce proper isolation of hypervisor and guest. Its translation techniques therefore do not apply to bare-metal hypervisors.

Varmosa targets the dated ARMv4 architecture, and its translation techniques have unfortunately never been published. A VMware talk and patent on DBT techniques for ARM published by members of the team that supported Varmosa reveals that they used *in-place translation* [10, 18], a technique also known as patching [46]. The general idea is that traps to the DBT engine can be injected directly into the guest kernel's original code stream for both sensitive and control flow instructions—the translations therefore overwrite the original code. The technique is flawed by design, however.

The MMU on the ARMv7-A architecture does not distinguish between data accesses and instruction fetches in enforcing access permissions on memory mappings. For code to be executable, it must also be readable [3]. Therefore, a guest can observe modifications to its code made by the hypervisor's DBT engine. This is problematic in a number of scenarios, such as code that verifies itself, code that performs self-modifications, or code that relocates itself in memory, even if the guest correctly uses cache maintenance operations to indicate its code has changed. The hypervisor can at best detect that a guest has corrupted itself, without any way to recover. Although self-modifications of kernel code may be rare in practice, relocations and verification are more common. It is necessary for a hypervisor to support all these cases because one of the core strengths of full virtualisation is the ability to virtualise a priori unknown operating systems.

The problems with in-place translation can be solved by storing the translated code separately from the

original code. The translated code is then stored in a software cache, at a different virtual address than the original code. Translation to a software cache makes room for binary code optimisations, as the size of translated blocks is no longer limited to the size of the original blocks. However, as the PC in the ARM architecture is exposed as a "general-purpose" register that is usable in several instructions ranging from arithmetic to memory operations, *all* instructions whose behaviour depends on their PC value must be translated in addition to control flow and sensitive instructions [23, 38, 39].

On ARM, almost all ALU instructions as well as load and store instructions can use the PC as source or address operand. Such instructions are quite frequent and a DBT engine should therefore avoid having to intervene at run time as much as possible. Solutions have already been provided by the authors of Pin [23] and Strata-ARM [38]. Instructions that access the PC are translated into equivalent sequences that do not require run-time intervention of the DBT engine. The DBT engine constructs the original address of the code into a substitute register and then translates the original instruction into one that uses the substitute register. This technique works as long as the DBT engine can find a suitable substitute register. For instructions that write their result into a destination register that is not reused as source operand, the destination register can be used as substitute. Otherwise, another register must be freed by saving its value to a spill location. It is then used as substitute for the PC in the translated instruction, and subsequently restored. There are two ways to do this without run time intervention of the DBT engine:

1. by reusing the application's stack; or
2. by performing a store to and load from a dedicated spill location using a PC-relative addressing mode that does not involve other registers for addressing.

Method 1 is used in Pin [23]. It complicates the translation of instructions that use the stack pointer, because the spill and restore operations alter the stack pointer. It also assumes that the stack pointer is valid. While this assumption holds for most user applications, it does not hold for kernel code, as kernels manage their own stacks, and the hypervisor cannot rely on those stacks being set up or usable at all. Even if stacks are set up, introducing extra stack accesses may cause unintended side effects. This method therefore does not work for system-level DBT.

Method 2 does not corrupt the stack but can only be used to address a spill location in memory at most 4 kB before or after the current instruction[1]. Because on ARM the smallest granularity of memory protection is 4 kB, this requires part of the software cache to be writable by the translated code. While it is possible to interleave read-only pages of code with dedicated read-write spill pages, this would dedicate half of the virtual address space of the software cache to storing the value of a single spilled register. The practical alternative is to make the entire software cache writable to the translated code. This approach is used in Strata-ARM [38].

Making the translated code writable to itself is never a good idea, however. Even if all translated instructions observe their native PC value, they can still access the software cache directly using absolute addresses. Since such accesses cannot be detected, faulty code can corrupt hypervisor memory, and malicious code can inject specific instructions to escape control of the DBT engine, e.g., by avoiding the trap at the end of a translated block. While this may not be a problem in user-space DBT engines, it is unacceptable in a hypervisor that relies on DBT to isolate guests from one another and from the hardware.

In user-space DBT engines such as Strata-ARM, software caches are writable by design, as both DBT engine and translated code run in user space, and hence cannot benefit from hardware-enforced privilege separation. A bare-metal hypervisor can leverage hardware techniques to isolate translated code, but doing so implies that translated code cannot spill and restore registers using the same techniques as found in user-space DBT engines.

[1] This is an instruction set limitation; see `LDR` (literal) and `STR` in [3].

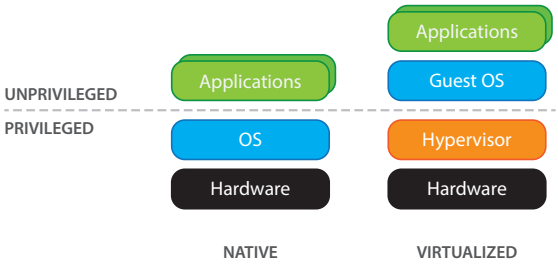

Figure 2: Native vs. virtualised privilege levels

3. The STAR hypervisor

The STAR (Software Translation for ARM) hypervisor is a bare-metal hypervisor that fully virtualises the ARMv7-A architecture on top of an ARMv7-A-based platform. It runs unmodified guest operating systems, decoupled from the hardware through DBT, and uses shadow translation tables to virtualise the MMU.

3.1. Architectural overview

The basic ARMv7-A architecture offers only two privilege levels. There are several equally privileged modes, but only one unprivileged mode, *user* mode. In a traditional operating system, the privileged modes are used for the kernel, and the unprivileged mode is used for user applications. When virtualised, only the hypervisor is privileged and guest operating systems together with their user applications must be executed in the unprivileged mode, as shown in Figure 2. The hypervisor thus gains the additional responsibility of enforcing privilege separation between the guest kernels and their user applications.

In order to make this work, the hypervisor has to solve two important problems. Firstly, as discussed in Section 2, executing a guest operating system kernel in the unprivileged mode causes all sensitive instructions shown in Table 1 to behave differently, because the ARMv7-A architecture is not classically virtualisable [39]. Instructions from guest kernels must therefore be translated. Instructions from user applications behave identically under native and virtualised execution. User applications can therefore be executed without translation and without any kind of supervision by the DBT engine.

Secondly, the MMU must be shared between the hypervisor and the guest: The guest kernel will create memory mappings, but the hypervisor must also be able to make its own mappings and to modify the guest's mappings. This requires the use of shadow translation tables. These two problems define the tasks of the two largest components of our hypervisor: the DBT engine and the virtual MMU. In this paper, we mainly focus on the challenges in the construction of our DBT engine for the 32-bit ARM instruction set. We nevertheless provide a brief overview of how shadow translation tables work on ARM, as this has important consequences for the DBT engine.

Our hypervisor aims to be usable for several different use cases, including full-system debugging and instrumentation. The naive version of our hypervisor must therefore avoid techniques that cause guest-observable differences in behaviour. In Section 5, we present optimisations over our naive version, which can be enabled or disabled based on the usage scenario of the hypervisor as some of these optimisations may, in rare cases, cause observability issues.

3.2. MMU virtualisation

Operating systems use an MMU for various tasks ranging from address translation and memory protection to cache configuration. All these tasks are administered through translation tables. A hypervisor

essentially performs the same tasks: it uses address translation for itself and for relocating guests within physical memory. The hypervisor needs to isolate its own memory from guests, and guests from one another. Guest memory may need to be protected to keep the hypervisor's software caches consistent with the guest. Furthermore, the hypervisor needs complete control over the hardware caches. The hypervisor therefore sets up its own translation tables, which are used together with the guests' translation tables.

The MMU in the bare ARMv7-A architecture can only use one set of translation tables at a time. Combining the hypervisor's translation tables with a guest's translation tables requires a software mechanism that merges the tables into a single set of shadow translation tables [1, 7, 46].

Operating systems enforce privilege separation between kernel and user space by specifying different permissions for memory accesses from privileged and unprivileged modes in their translation tables. As shown in Figure 2, once virtualised, both the guest kernel and its user applications run in the unprivileged mode. The MMU can therefore no longer distinguish between a guest's kernel and its user applications. To enforce privilege separation in the guest, the hypervisor creates two sets of shadow translation tables: a privileged set, which is set up to enforce access permissions for the guest's kernel, and an unprivileged set, configured to enforce access permissions for guest user applications. The MMU is configured by the hypervisor with the appropriate set of shadow translation tables depending on the guest's virtualised privilege level. This approach is known as *double shadowing* [19].

3.3. Sensitive instructions

Table 1 provides an overview of all 32-bit ARM instructions that require special handling by our DBT engine, based on our previous analysis of the ARM architecture [39]. Table 1 contains both control flow instructions and sensitive instructions. Control flow must be tracked in order to keep control over the execution of a guest kernel. Sensitive instructions must be interpreted or translated into equivalent instruction sequences, because their behaviour changes when they are executed in unprivileged mode (right of Figure 2) instead of the originally intended privileged mode (left of Figure 2). We briefly reiterate our findings from our previous analysis to clarify which instructions are sensitive and why.

Guests must be isolated from the physical hardware platform; instructions that access physical devices such as coprocessors are therefore sensitive. As guests are always executed in the unprivileged mode, the DBT engine becomes responsible for emulating the different modes that exist natively. As a result, all instructions that depend on the current mode or that alter the current mode, including exception return instructions, are also sensitive.

The DBT engine maintains a shadow register file, containing all general-purpose registers, the program status registers, etc. On ARM, some registers are *banked*, i.e., physically duplicated, for different modes; e.g., almost every mode has its own stack pointer (SP) and link register (LR). Our shadow register file stores all copies of the banked registers separately. Instructions that access banked registers are sensitive and must be translated to access the shadow register file. For example, `LDM` and `STM` instructions can be used to access the SP and LR of the unprivileged user mode from a privileged mode. A kernel uses them to back up the state of a user application, such as when switching tasks.

`LDRT`, `STRT` and similar instructions perform memory accesses as if they were executed from the unprivileged mode, regardless of their actual privilege level. They are used by a kernel to access memory with the permissions of an application; this is necessary to efficiently enforce privilege separation and to support copy-on-write semantics when copying data to or from user applications within kernel code. Such instructions no longer behave correctly when double shadowing is used, as the permissions for the guest's unprivileged mode are unknown to the MMU at the time guest kernel code is being executed. Therefore, all such instructions are also sensitive.

Table 1: Control flow and sensitive instructions in the 32-bit ARM instruction set

Instruction	Control flow	Sensitive
ALU* branches (e.g., `MOV pc, lr`)	●	○
ALU* exception returns (e.g., `MOVS pc, lr`)	●	●
`B`, `BL`, `BLX`, `BX`, `BXJ` branches	●	○
`CPS`, `MSR` mode changes	○	●
`CDP`, `LDC`, `MCR`, `MCRR`, `MRC`, `MRRC`, `STC` coprocessor accesses	○	●
`LDM`, `LDR`, `POP` branches	●	○
`LDM`, `RFE` exception returns	●	●
`LDM`, `STM` user mode multiple register restore and save operations	○	●
`LDRT`, `LDRBT`, `LDRSBT`, `LDRHT`, `LDRSHT`, `STRT`, `STRBT`, `STRHT` load and store as unprivileged	○	●
`MRS`, `MSR`, `SRS` accesses to banked registers, the CPSR and the SPSR	○	●
`SVC` system calls	●	●
`WFE`, `WFI` sleep instructions	○	●

* In the 32-bit ARM instruction set, all ALU instructions that write their result into a register can be used as a branch or exception return. They are `ADC`, `ADD`, `AND`, `ASR`, `BIC`, `EOR`, `LSL`, `LSR`, `MOV`, `MVN`, `ORR`, `ROR`, `RRX`, `RSB`, `RSC`, `SBC` and `SUB`.

3.4. Operation of the DBT engine

Our DBT engine consists of an instruction decoder and encoder, a translator, an interpreter, a software cache for translated code, called the *code cache*, and a second software cache to hold metadata about translated code, called the *metadata cache*. Figure 3 depicts the operational cycle of our DBT engine, adapted from the generic cycle shown earlier in Figure 1. Translated guest kernel code executes from the code cache in unprivileged mode. Two such translated code blocks in the code cache are shown on the left of Figure 3. Guest user applications are not touched by our DBT engine as already explained in Section 3.1. Our interpreter emulates control flow and sensitive instructions in a privileged mode on the guest's virtual machine state. After interpretation, we check whether or not the guest has switched its execution mode from a kernel mode to user mode. If so, the DBT engine is exited, and we perform a context switch to the guest's user application. When the guest remains in a kernel mode, the flow is the same as discussed in Section 2.3.

Traps to the interpreter are implemented using the `SVC` instruction, which is normally used by an operating system to implement system calls. When the hypervisor receives a system call trap, it first checks whether the trap came from the translated code, or from a guest's user application. In the latter case, the trap is forwarded to the guest's exception handlers.

For each trap to the interpreter, our DBT engine must look up the metadata of the corresponding block to determine which instruction was replaced, and which interpreter function must be invoked. To ensure a fast look-up process with $O(1)$ complexity, we implement our metadata cache as a fixed-size array, and use the indexes of the cache items as operands to the `SVC` instructions that terminate our translated blocks.

Another look-up process is required to find the next translated block in the metadata cache, based on the native program counter (PC) value. We implement this look-up using hashing with open addressing; this means that newly translated blocks are assigned an entry in the metadata cache based on their source address.

The translator operates on the instructions of the source block one by one. Instructions are decoded to determine whether or not they can be copied to the code cache as is. Most instructions do not alter control

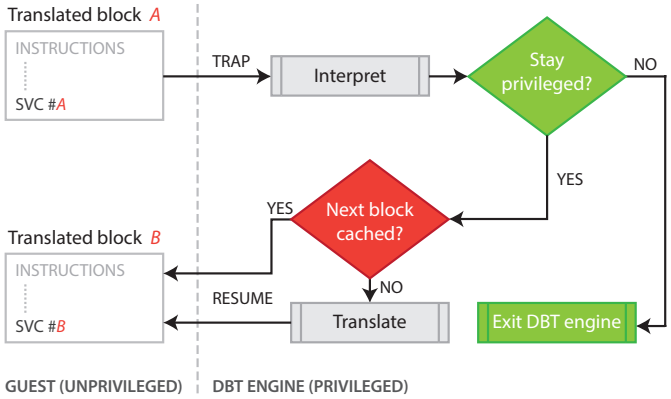

Figure 3: The basic operational cycle of our DBT engine

flow, and are not sensitive in any way; they are copied as is. For control flow and sensitive instructions, the decoder decides whether they are replaced by traps to the interpreter, or by equivalent sequences of instructions that can be executed without run time intervention of the DBT engine. For example, instructions that make use of the PC but are otherwise innocuous are translated to make sure they observe the value of the PC as if they were executed natively.

3.5. Design choices and limitations

Our design of the metadata cache imposes two important constraints on translated blocks: blocks have exactly one entry point and exactly one exit point. The first constraint is a result of our requirement for efficient metadata cache look-ups: by using hashing based on the source address of the first instruction of a translated block, we cannot easily look up blocks using the source address of other instructions in the block. This causes some code fragments (i.e., those reachable through branches as well as a fall-through path) to be translated more than once, but requires less metadata to be maintained for each translated block.

Our DBT engine replaces conditional control flow and conditional sensitive instructions with unconditional traps to the interpreter, resulting in the second constraint of having exactly one exit point. This also simplifies the metadata we store for each translated block, because the metadata stores which instruction is replaced and which interpreter function should be invoked to emulate its behaviour. By letting each unconditional trap replace only one instruction, the metadata structure is kept simple and fixed in size for all translated blocks. Replacing conditional instructions by unconditional traps causes unnecessary traps for instructions whose condition would normally not be met; these traps can be eliminated without altering the structure of the metadata as will be shown in Section 5.

Unlike existing user space DBT engines, the STAR hypervisor features a bounded code cache with a fixed size of 1 MB. When it is full and more code needs to be translated, all previously translated blocks are unconditionally flushed from the metadata cache and the code cache. In theory, it is possible to selectively clean cold blocks from the cache. In practice, however, such techniques cause substantial run-time overhead as they complicate software cache management, and they require more metadata to be stored to enable optimisations that create links between translated blocks, and that will be presented in Section 5.

3.6. Translating PC-sensitive instructions

ARM instructions frequently use the PC as a source operand: as they can only embed small constants of at most 16 bits, larger constants are placed in between code, in so called *literal pools*, which are accessed through instructions that use PC-relative addressing. All such instructions should be translated without trapping to the interpreter if they do not alter control flow and if they are not sensitive in the classic sense according to Popek and Goldberg [40]. They are:

- the move and shift instructions `MOV`, `MVN`, `ASR`, `LSL`, `LSR`, `ROR` and `RRX`, which copy the PC into another register, optionally shifting or complementing its value;
- the generic computation instructions `ADC`, `ADD`, `AND`, `BIC`, `EOR`, `ORR`, `RSB`, `RSC`, `SBC` and `SUB`, which can be used to perform calculate addresses based on the PC;
- the test and compare instructions `CMN`, `CMP`, `TEQ` and `TST`, which compare the value of the PC to a constant value or to the value of another register without storing the result, only updating the condition flags;
- the load and store instructions `LDR`, `LDRB`, `LDRD`, `LDRH`, `LDRSB`, `LDRSH`, `STR`, `STRB`, `STRD` and `STRH`, for PC-relative addressing to access to literal pools; and
- the store instructions `STR` and `STM`, as they can store the value of the PC to memory.

Except `STM`, the translations of all these instructions are straightforward; example translations are shown in Table 2. The most simple instruction is the `MOV` instruction of example (1). We translate a `MOV` instruction that copies the PC into an equivalent pair of `MOVW` and `MOVT` instructions. Moore et al. [38] instead embed the PC value as a constant in the code cache and use a PC-relative load to retrieve it, because their techniques target ARMv5, which predates the introduction of the `MOVW` and `MOVT` instructions.

In general, if an instruction writes to some destination register `Rd` that is not also used as source register, that `Rd` can be reused as substitute for the PC, as shown in example (2). When `Rd` is used as source register, or in the absence of some `Rd`, an unrelated register must be used instead. We do not perform full liveness analysis on the block to be translated to find such a register, as blocks are small and the opportunities for finding available registers are limited. Instead, we always spill and restore an unrelated register before and after the translation. Since we propose multiple techniques for spilling and restoring in Section 4, the examples in Table 2 use the `SPILL` and `REST` pseudo-instructions. Translations of conditional instructions, such as in example (3), can only be made wholly conditional if the condition flags remain the same throughout the translation. Otherwise, we skip the translation using a conditional branch with a condition code opposite to the one of the original instruction, as shown in example (4).

Moore et al. [38] fold computations on the PC if all operands are known at translation time. For example, in translations of shift instructions, they directly write the shifted value of the PC to the destination register, saving one instruction in the translation. Folding has limited applicability, as it cannot eliminate updates of the condition flags. Furthermore, it is a micro-optimisation as it at most avoids emitting one ALU instruction. It is in other words not essential; our DBT engine does not yet support it.

Translating `STM` instructions requires an approach that is very different from the other instructions. `STM` instructions are used to store two or more given registers to memory with a single instruction. Their most common use case is to push registers to the stack, but handcrafted assembly may use them in entirely different ways. We therefore need a generic translation algorithm.

In a limited number of cases, the PC can be substituted with another register. If not, the `STM` must be split. The simplest way is to split the `STM` into individual stores. However, this may require up to 16 stores, inflating the size of the translated code. We therefore propose an algorithm that splits the original `STM` into one `STM` of all registers other than the PC, and an individual store of the PC.

Table 2: Example translations of PC-sensitive instructions

Example PC-sensitive instruction	Translation
(1) `MOV<c> Rd, PC`	`MOVW<c> Rd, #(NativePCValue[15:0])` `MOVT<c> Rd, #(NativePCValue[31:16])`

(2) `ADD<c> Rd, PC, #constant`	`MOVW<c> Rd, #(NativePCValue[15:0])`
	`MOVT<c> Rd, #(NativePCValue[31:16])`
	`ADD<c> Rd, Rd, #constant`

(3) `ADD<c> Rd, PC, Rd`	`SPILL<c> Rs`
	`MOVW<c> Rs, #(NativePCValue[15:0])`
	`MOVT<c> Rs, #(NativePCValue[31:16])`
	`ADD<c> Rd, Rs, Rd`
	`REST<c> Rs`

(4) `ADDS<c> Rd, PC, Rd` (with c $\neq$ AL)	`B<¬c> skip`
	`SPILL Rs`
	`MOVW Rs, #(NativePCValue[15:0])`
	`MOVT Rs, #(NativePCValue[31:16])`
	`ADDS Rd, Rs, Rd`
	`REST Rs`
	`skip:`

All STM instructions store registers to consecutive words in memory in the order of their index: R0 first, PC (R15) last. There are four different addressing modes: increment after (IA), increment before (IB), decrement after (DA) and decrement before (DB). For incrementing modes, the address to which the first register is stored is the base address, optionally incremented by one word with the "before" mode. In decrementing modes, the last address is the word before or at ("after") the base address; the first address thus depends on the number of registers being stored. All addressing modes also support "writeback", i.e., updating the value of the base address register at the end of the operation.

Algorithm 1 shows how we translate STM instructions. In all cases we spill to obtain a scratch register Rs. If there is some Rs different from the base address register, and indexed lower than the PC but higher than all other registers being stored, then the we do not split the STM as Rs can substitute the PC (lines 8 and 19). Otherwise, we split the instruction into a pair of STM and STR instructions; the base address must then be corrected to make up for any differences caused by storing one register less in the translated STM. For descending addressing modes, the base address must be adjusted upfront (line 13). The addressing mode of the STR instruction can always be chosen such that no further arithmetic is required to reach the final value of the base address register (lines 21 to 30).

Spilling can be avoided if the STM is split, and one of the registers other than the PC is used as scratch register. The scratch register can then be restored from guest memory. However, such optimisations may be observed by the guest. We therefore omitted them from our algorithm. In case any of the translated instructions cause an MMU fault, our approach introduces extra overhead in handling this fault over splitting the STM into individual stores: depending on the location of the abort, both the base address register and Rs must be restored. When there are few such faults, however, the run-time overhead of our solution is lower compared to using individual stores. In practice, such MMU faults rarely occur in kernel code. This validates our design choice.

Algorithm 1: 32-bit ARM `STM` instruction translation

```
Data: OriginalSTM, OriginalPC
Result: an instruction sequence of which the functional behaviour is equivalent to OriginalSTM, and which can
        be executed from a virtual address different from OriginalPC

C ⟵ get_condition_code(OriginalSTM)
Rn ⟵ get_base_address_register(OriginalSTM)
Registers ⟵ get_registers_to_store(OriginalSTM) \ {15}
Rs ⟵ max(Registers ∪ {Rn}) + 1
HaveSubstitute ⟵ Rs < 15
TranslatedSTM ⟵ OriginalSTM

if HaveSubstitute then
    set_registers_to_store(TranslatedSTM, Registers ∪ {Rs})
else
    set_registers_to_store(TranslatedSTM, Registers)
    Rs ⟵ min({0,1} \ {Rn})
    if not is_incrementing(OriginalSTM) then
        emit(SUB<C> Rn, Rn, #4)
    emit(TranslatedSTM)

emit(SPILL<C> Rs)
emit(MOVW<C> Rs, #OriginalPC[15:0])
emit(MOVT<C> Rs, #OriginalPC[31:16])

if HaveSubstitute then
    emit(TranslatedSTM)
else
    if is_incrementing(OriginalSTM) = is_write_back(OriginalSTM) then
        if is_incrementing(OriginalSTM) = is_before(OriginalSTM) then
            emit(STR<C> Rs, [Rn, #4]!)
        else
            emit(STR<C> Rs, [Rn], #4)
    else
        Offset ⟵ 4 · |Registers|
        if is_incrementing(OriginalSTM) = is_before(OriginalSTM) then
            Offset ⟵ Offset + 4
        emit(STR<C> Rs, [Rn, #Offset])

emit(REST<C> Rs)
```

3.7. Guest exception handling

Handling exceptions in translated code is a problem typical to system-level virtual machines. Exceptions can occur either synchronously or asynchronously. Synchronous exceptions are tied to the execution of a particular instruction; e.g., a faulting memory operation, an undefined instruction, or a system call. Asynchronous exceptions occur due to external influences, such as an interrupt from a peripheral device. Guest exceptions outside translated code, i.e. in guest user space, are handled by forwarding the exception to the guest's exception handler. Exceptions in translated code are more complex to handle, as the code

cache cannot be exposed to the guest, and the guest state must be consistent upon delivery of an exception.

3.7.1. Synchronous exceptions

System calls and undefined instructions in a guest's kernel can be recognised at translation time, while faulting memory operations must be handled at run time. For exceptions handled at run time, the hypervisor must map the PC value at which the exception occurred to the native PC value of the guest, before delivering the exception to the guest's kernel. Such mapping can be achieved by maintaining a mapping of code cache addresses to native addresses, or through retranslation. Storing the address mapping for every instruction that can fault consumes a lot of memory. In practice, few instructions cause MMU faults. We have therefore chosen to use the retranslation approach. The native addresses are known on the boundaries of a translated block. In order to map an address in the code cache to a native address, we retranslate the block without writing to our software caches, up to the location of the faulting instruction. When exceptions are caught in code that spills and restores registers, the hypervisor must first make sure all register values are restored such that the guest is in a consistent state, before delivering the exception to the guest's kernel, as the guest's exception handler cannot return within the translated block.

Providing the guest with the original PC value of the exception ensures that the guest cannot observe the presence of our hypervisor. More importantly, guest fault handling code may depend on the address at which the exception occurred; one such dependency can be found in the `do_page_fault` function of the Linux kernel.

3.7.2. Asynchronous exceptions

Asynchronous exception delivery is postponed until the end of a translated block. This increases interrupt latency but avoids the cost of mapping the guest's current PC value in the code cache to its native PC value. Furthermore, when an asynchronous exception would be delivered to a guest in the middle of a translated block, the guest would return to an address that has already been translated, but as our translated blocks only have a single entry point, the translation would not be usable. We would therefore have to translate a new block, starting from the return address. Since asynchronous exceptions can arrive on every instruction, the number of new translations that would be created during exception handling is virtually unbounded. Breaking blocks for asynchronous exceptions would therefore nullify all benefits of caching translated code. By postponing the delivery until the end of a block is reached, this is avoided completely.

4. Spilling and restoring registers

Translating instructions that use the PC as source operand requires finding a substitute register to hold the native value of the PC. This may involve spilling and restoring a register. On the one hand, spilling and restoring must avoid a full context switch from the guest to the hypervisor, as that would defeat the purpose of translation. On the other hand, all translations must be protected from guest modification for security and reliability reasons; therefore, we cannot spill to the code cache without a trap.

We propose two generic solutions to the spilling problem. In our first solution, we use a lightweight trap, which avoids a full context switch, but at the same time enables translated code to write to otherwise protected locations in memory. Our second solution consists of using user-mode accessible coprocessor registers. When a guest is known to exhibit a certain behaviour, other tricks may be employed (e.g., using the guest's stack); however, such non-generic solutions are out of the scope of this paper.

Listing 1: Lightweight trap handler

```
1 MRS      sp_und, spsr
2 AND      sp_und, sp_und, #PSR_MODE
3 TEQ      sp_und, #PSR_MODE_USR
4 LDREQ    sp_und, [lr_und, #-4]
5 MVNSEQ   sp_und, sp_und
6 BXEQ     lr_und
```

4.1. Lightweight traps

Our hypervisor uses the system call mechanism (`SVC` instruction) for traps to the DBT engine. We implement lightweight traps using an undefined instruction trap, to avoid adding complexity to the system call handler. The sole objective of lightweight traps is to switch the mode in which the translated code is executed to a privileged mode: When this mode switch is performed in the middle of a translated code block, which is normally executed in the unprivileged mode, the subsequent instructions in the block will execute in the privileged UDF mode until a later instruction in the block changes the mode to unprivileged again. While executing in the privileged mode, the translated code can access the otherwise protected locations as necessary for spilling.

The undefined instruction trap and the mode switch are initiated by an undefined instruction inserted by the DBT engine in the translated code block. Guests cannot exploit this mechanism because (1) they cannot alter the contents of the software cache, and (2) because in case that same undefined instruction is encountered in the guest's instruction stream, the hypervisor will handle it as a sensitive instruction, and translate it into a trap into the DBT engine before it can be executed. Kernels normally do not cause undefined instruction traps. Our lightweight trap mechanism therefore does not introduce any additional overhead in the kernel's normal operation.

The key aspect of this approach is the design of an undefined instruction trap handler that can (1) classify caught traps, (2) return control to the translated code in some privileged mode when the trap is classified as the hypervisor's lightweight trap, (3) invoke a generic undefined instruction trap handler otherwise, and (4) does so in as little as possible time to minimise the performance overhead.

We have managed to come up with a mechanism and a handler that requires only eight instructions to enter the privileged UDF mode from within the code cache. This mechanism relies on a specific instruction encoding that is permanently defined to generate an undefined instruction trap by the ARM architecture [3], encoded as `0xFFFFFFFF`. This encoding is used in the code cache to perform the lightweight trap. The trap is caught by the hypervisor's exception vector, which contains branches to handlers for all kinds of exceptions. We use different exception vectors based on the virtual mode of the guest, so that undefined instruction traps from guest user space cannot reach our lightweight trap handler. Our exception vectors for the guest's privileged modes contain a simple branch to the lightweight trap handler shown in Listing 1.

In lines 1 to 3 of our handler, we check whether the undefined instruction trap was taken while executing in the unprivileged mode. If so, we know that the trap is caused by guest kernel code executed from the code cache. If not, lines 4 to 6 are not executed. In line 4, we load the instruction that caused the trap. In line 5, the encoding of the instruction is complemented, and the condition flags are updated; they are used in line 6 to jump back to the code cache if the bitwise complement of the instruction equals zero, which only happens if the trap was caused by our lightweight trap instruction with encoding `0xFFFFFFFF`. When the branch in line 6 is not executed, control falls through to a generic undefined instruction handler.

Our lightweight trap handler uses only banked registers to avoid saving guest registers. After jumping back to the code cache, both SP and LR are usable, such that spilling to memory can sometimes be avoided.

To resume execution in the unprivileged mode, the DBT engine inlines a PC-relative exception return instruction in the translated code.

We have implemented two different usage scenarios of our lightweight traps for spilling. In the first scenario, a private spill location is appended as needed at the end of a translated block inside the code cache. As is traditional on many ARM systems, our hardware platform has separate L1 instruction and data caches, and a unified L2 cache. In order to simplify cache maintenance requirements when translating new blocks, the code cache is mapped write-through at the L1 cache level, and write-back write-allocate at the L2 cache level. Therefore, spilling to and restoring from the code cache may cause many L1 cache misses. In a second scenario we spill to a shared spill location outside the code cache, which is mapped write-back and write-allocate at both L1 and L2 cache levels. This trades L1 cache hits for increased TLB pressure; since we aim to avoid guest-readable data structures outside the code cache, reading from the shared spill location may require a second lightweight trap if the translated instruction cannot be executed from a privileged mode. Such a second trap is never required in the first scenario.

4.2. User-mode accessible coprocessor registers

All modern ARM processors up to ARMv7-A support coprocessors. Coprocessors are devices whose registers can be accessed using specific coprocessor instructions, typically with lower latency than memory-mapped device registers. Each ARMv7-A processor comes with at least two coprocessors: the debug coprocessor and the system control coprocessor. The latter is used for CPU identification, MMU configuration, cache maintenance, etc.

The system control coprocessor contains a few registers that are accessible from user mode. We therefore researched the feasibility of using such registers as spill location. Suitable registers do not affect the behaviour of the hypervisor or the underlying hardware, are both readable and writable from user mode, and can hold a 32-bit word.

At least three coprocessor registers in the ARMv7-A architecture match our criteria: we can use either one of the performance counter registers (PMCCNTR and PMXEVCNTR), or the user-writable software thread ID register (TPIDRURW) [3]. The generic timer extension also offers suitable registers, but using those registers for spilling renders the timers unusable for the hypervisor.

Using hardware coprocessor registers as spill location does not interfere with a guest kernel's usage of coprocessors, as guest kernels normally do not interact with hardware coprocessor registers directly; our DBT engine ensures that the guest can only access a virtual copy of the coprocessor registers. Since our hypervisor runs user applications unmodified, all user-mode readable registers must be restored upon a guest context switch from kernel to user mode; all user-mode writable registers must additionally be saved upon a guest context switch from user mode to kernel. When such registers are not used for spilling, we can let the kernel update them directly to avoid the maintenance cost on every guest context switch.

Using performance counter registers for spilling comes with the disadvantage that the hypervisor can no longer make full use of the performance counters for itself. Performance counter registers are made accessible to the unprivileged mode through configuration of the system control coprocessor. Spilling to the performance counter registers therefore also requires maintenance on every context switch, to disable and re-enable access accordingly. Unlike the thread ID register, the maintenance for using the performance counters does not require extra memory accesses during guest context switching.

5. Tackling DBT-related overhead

The naive version of our hypervisor translates all control flow and sensitive instructions shown in Table 1 into traps to the interpreter. This gives the hypervisor maximum visibility into the guest kernel's

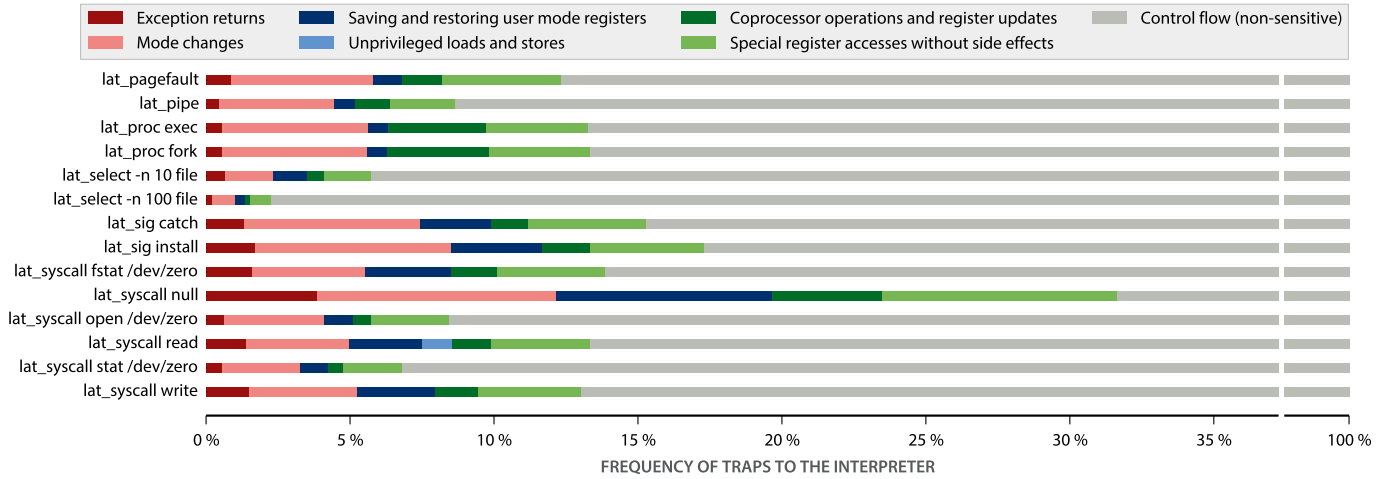

Figure 4: Frequency of traps to the interpreter in the naive version of our hypervisor by instruction class

execution, which is useful for tracing, but too slow for any other practical purpose. We have therefore studied which traps occur most frequently, and we propose optimisations to eliminate them.

All DBT-related overhead observed in user applications originates from interactions with the kernel and interrupt handling, because our hypervisor only translates kernel code. For a Linux guest, this means we can analyse the origins of DBT-related overhead by running micro-benchmarks for system calls and signal handling.

To do so, we have executed selected benchmarks from the *lmbench* version 3 suite [37] on a BeagleBoard, which is based on TI's OMAP 3 system-on-chip and which features a Cortex-A8 processor. While our hypervisor has also been ported to various other platforms with Cortex-A9 and Cortex-A15 processors to ensure its features are compatible with a wider range of hardware, it currently only supports running on a single processor core, and it always presents a minimal OMAP 3-based virtual machine to its guests regardless of the underlying hardware platform. These limitations however do not affect the generality of the DBT techniques presented in this paper. As guest OS, we have used a vanilla Linux 3.4.108 kernel. The benchmarks we use measure the virtualisation cost on frequently-used kernel functionality:[2]

- `lat_pagefault`: measures the cost of faulting on pages from a file.
- `lat_pipe`: uses two processes communicating through a UNIX pipe to measure inter-process communication latencies. The benchmark passes a token back and forth between the two processes which perform no other work.
- `lat_proc`: creates processes in different forms, to measure the time it takes to create a new basic thread of control.
  - `fork`: measures the time it takes to split a process into two nearly identical copies and have one of them exit.
  - `exec`: measures the time it takes to create a new process and have that new process run a new program, similar to the inner loop of a command interpreter.

[2] Benchmark descriptions copied and adapted from the lmbench manual pages.

- `lat_select`: measures the time it takes to perform a `select` system call on a given number of file descriptors.
- `lat_sig`: measures the time it takes to install and catch signals.
- `lat_syscall`: measures the time it takes for a simple entry into the operating system kernel.
  - `fstat`: measures the time it takes to `fstat()` an open file whose inode is already cached.
  - `null`: measures the time it takes to perform a `getppid` system call. This call involves only a minimal and bounded amount of work inside the kernel, and therefore serves as a good test to measure the round-trip time to and from the kernel back to user space.
  - `open`: measures the time it takes to open and then close a file.
  - `read`: measures the time it takes to read one byte from `/dev/zero`.
  - `stat`: measures the time it takes to `stat()` a file whose inode is already cached.
  - `write`: measures the time it takes to write one byte to `/dev/null`.

We have restricted our selection of benchmarks to those that are relevant to show the performance of our DBT engine. We therefore exclude benchmarks that are designed to test disk, network and memory bandwidth.

We have executed each benchmark once and measured the source and frequency of traps to the interpreter. Internally, we have forced each benchmark to execute its test 100 times in a loop, such that we can easily distinguish the traps specific to the benchmark from the traps caused by the creation and destruction of the benchmark process. Figure 4 shows an overview of the source and frequency of traps for each individual benchmark. As is typical in DBT, most of the overhead is caused by control flow. Control flow optimisations have already been studied extensively, so we have implemented some of the existing optimisations. For the traps not related to control flow, we propose new translations specific to full virtualisation on ARMv7-A.

5.1. Control flow

Our measurements indicate that over all benchmarks, 68% to 96% of our overhead is caused by non-sensitive control flow instructions. As shown in Figure 5, the majority of control flow instructions are direct branches. The traps caused by direct branches can be eliminated through lazy optimisation: after a trap caused by a branch, the block targeted by that branch is translated, and the trap is replaced by a branch within the code cache to the newly translated block. We say that the blocks are *linked* together.

Conditional direct branches are initially translated into two traps: a conditional trap with a condition code opposite to the original instruction, which models the fall through path of the original instruction, and a non-conditional trap. Both traps can be replaced lazily by a direct branch within the code cache. In general, an extra conditional trap is added to every translated conditional instruction, which can be linked like a direct branch.

5.1.1. Indirect branches

Blocks with indirect branches cannot be linked at translation time, as the destination addresses of the branches are not known until they are executed and as they can vary between executions. Several solutions to eliminate traps caused by indirect branches already exist. Moore et al. [38] use an indirect branch target cache, a global hash table of code cache addresses indexed by native PC values [27], which is used by the translated code to look up translations of branch targets. A similar technique, called a sieve, implements

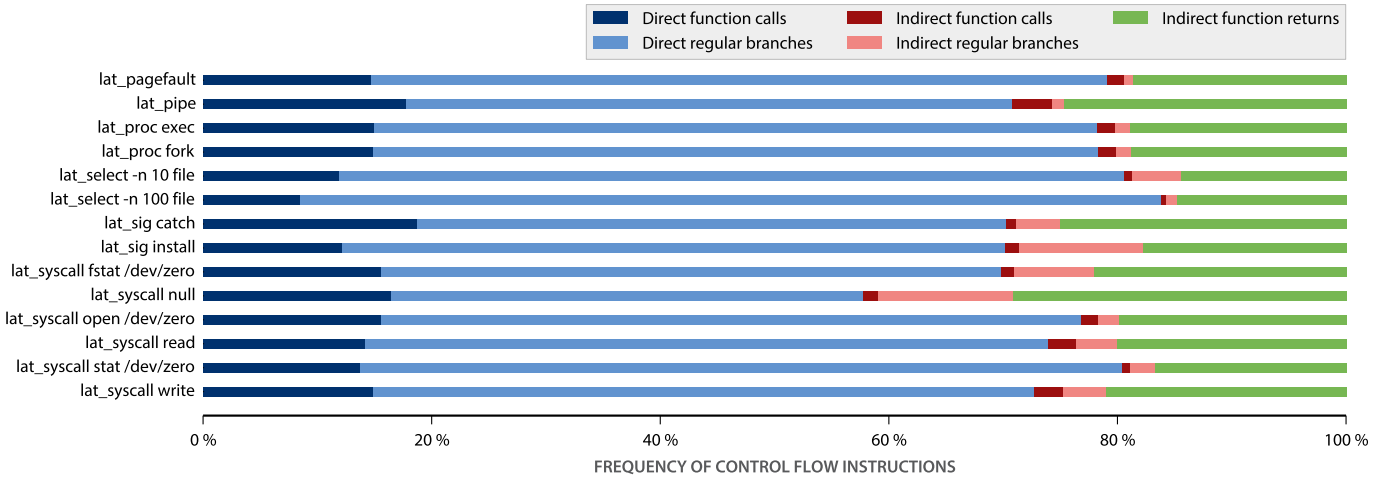

Figure 5: Frequency of traps caused by control flow instructions

the hash table in instructions rather than in data [47]. A limited set of code cache addresses known to be frequently targeted by specific indirect branches can also be inlined in the translation of that indirect branch [23, 27]. Most techniques treat function returns separately, as the target of a function return can be predicted easily.

Function calls can be recognised in branch instructions that automatically write the return address to LR (`BL` and `BLX`). Function returns are harder to identify, as the ARM architecture does not have a dedicated function return instruction, such as `RET` on x86. We therefore identify function returns based on the patterns recognised by ARM's hardware return address stack; they are:

- an indirect branch to the value in the LR (`BX lr`);
- a copy of the LR into the PC (`MOV pc, lr`);
- a load of the PC from the stack using any of the `LDR`, `LDMIA` and `LDMIB` instructions[3].

To avoid the traps caused by function returns, our hypervisor implements a software return address stack, also known as a shadow stack, similar to Pin [23, 33]. The patterns we use to identify function returns are not fully accurate, as on average they identify 4% more function returns as there are function calls; we suspect these are partly caused by handwritten assembly in the kernel. Our shadow stack is nevertheless capable of significantly reducing DBT-based overhead, as we will show in Section 6.

Unlike Pin and other user-space DBT engines, our shadow stack implementation does not use the guest's stack for saving and restoring registers during manipulations of the shadow stack; instead, we use the register spilling techniques that we proposed in Section 4.

As the amount of indirect branches other than function returns is fairly low, our hypervisor currently does not implement a generic indirect branch optimisation algorithm.

[3] ABI-compliant code uses descending stacks which are always accessed using the incrementing addressing modes of the `LDM` instruction.

5.1.2. Effects on asynchronous exception delivery

As discussed in Section 3.7.2, when the guest is executing in kernel space, we postpone the delivery of asynchronous exceptions such as interrupts until the end of a translated block. This is straightforward as long as the blocks end in a trap to the interpreter. When blocks are linked together, however, they can create loops in the code cache that do not contain a single trap. The execution of translated code can then continue in the code cache for quite some time without any trap being executed. Of course the hypervisor cannot wait until all loops have finished executing.

To overcome this problem, the DBT engine partially undoes block linking whenever an exception occurs by replacing any direct branches at the end of the active translated block by traps to the interpreter. Alternatively, if the block makes use of our shadow stack to perform a function return within the code cache, we corrupt the top entry of the shadow stack to ensure that the next look-up causes a trap.

5.2. Exception returns and other mode changes

Exception returns affect numerous data structures in the hypervisor, and involve hardware reconfiguration; it is therefore hard to avoid a full context switch from guest to hypervisor and back. As shown in Figure 4, exception returns can cause up to 4% of all traps in an optimised DBT engine. Mode changes using `CPS` and `MSR` instructions are very similar to exception returns, but do not influence control flow. Figure 4 shows that over 8% of the traps are caused by such mode changes.

Some of the mode change instructions merely enable and disable interrupts. Those instructions can be translated to equivalent instruction sequences that update the program status register of the guest in its shadow register file without requiring a trap to the interpreter.

5.3. Saving and restoring user mode registers

When an operating system performs a context switch from application to kernel or vice versa, it must save and restore all registers of the application. The ARM architecture provides special load and store multiple instructions (`LDM` and `STM`) that access the user mode SP and LR rather than the registers of the active mode, such that a kernel does not need to switch modes to access the user mode registers. These instructions are otherwise functionally equivalent to normal load and store multiple instructions.

Figure 4 shows that up to 8% of all traps in our DBT engine originate from instructions to save and restore user mode registers. In order to avoid these traps, we split the instructions into a first part that only involves the non-banked registers, and a second part that involves the banked registers. Only the second part requires accesses to the shadow register file to be inlined in the translation. When the instruction loads non-banked registers, the load of the banked registers is reordered to happen before the load of the non-banked registers, as all affected non-banked registers can then be used as scratch registers.

Listing 2 shows how our DBT engine translates `LDMDB sp, {r0-r12,sp,lr}^`, an instruction taken from the Linux kernel's system call return routine. This instruction restores all registers other than PC from the kernel's stack. Instead of updating the current SP and LR, however, it writes to their user mode counterparts. We use a lightweight trap to access the guest's shadow register file from the privileged UND mode (line 2). Because the base address resides in the banked SP register, we copy it into a scratch register before switching modes (line 1). We make use of the banked registers in the UND mode to hold the address of the guest's user mode registers in its shadow register file (lines 3–4), and to load values from the guest's stack using `LDRT` instructions (lines 5 and 7). Using `LDRT` makes the MMU authorise the loads using the guest's permissions, which is necessary to enforce guest isolation. We then store the loaded values into the guest's shadow registers (lines 6 and 8). `LDRT` instructions always operate on the address held in the base address register, and update its value with a specified constant after the load operation. We must therefore adjust the base address register upfront (line 1) and in between the loads from the shadow register file

Listing 2: Translation of `LDMDB sp,{r0-r12,sp,lr}^`

```
1  SUB     r0, sp_usr, #8
2  LWTRAP
3  MOVW    sp_und, #(AddressOfUSRRegs[15:0])
4  MOVT    sp_und, #(AddressOfUSRRegs[31:16])
5  LDRT    lr_und, r0, #4
6  STR     lr_und, [sp_und]
7  LDRT    lr_und, r0
8  STR     lr_und, [sp_und, #4]
9  SUBS    pc, pc, #4
10 SUB     r0, r0, #4
11 LDMDB   r0, {r0-r12}
```

(line 5). Afterwards, we return to the unprivileged mode (line 9), and we load the non-banked registers (line 11). As the original `LDMDB` instruction uses a decrementing mode, its base address must be adjusted to compensate for the removal of the banked registers; to avoid restoring the base address register, we write the adjusted base address into the scratch register (line 10).

Store multiple instructions that save user mode registers are translated similarly. Algorithm 1 shows how to compensate for the changes in the base address caused by altering the register list for all addressing modes.

5.4. Unprivileged loads and stores

Instructions that load and store as if executed from the unprivileged mode are used by a Linux kernel to access data from user applications. When such instructions are executed from the unprivileged mode, they behave as normal loads and stores. However, as explained in Section 3, this would cause the loads and stores to be executed with the permissions of the guest's kernel rather than those of its user applications due to double shadowing.

Emulating the behaviour of unprivileged loads and stores requires checking the permissions assigned to user applications, either by a software-based look-up in the guest's translation tables, or by switching the active set of shadow translation tables to the unprivileged set, and then performing the check in hardware. Trapping to the interpreter on every such load and store is costly.

Our initial approach to eliminate the trap consisted of inlining a switch to the unprivileged set of shadow translation tables before and after every unprivileged load and store instruction. Switching translation tables must be done from a privileged mode and therefore requires a lightweight trap. The code cache should not be accessible to the guest's user applications, in order to prevent exposing the translated kernel. The translated instruction must therefore be executed from a privileged mode. As the instruction performs memory accesses as if it was executed in the unprivileged mode, these accesses are always authorised with the permissions of the guest. There is a problem, however, when the instruction uses one or more banked registers: the value of the user mode registers must be fetched into the registers of the privileged UDF mode, and any updated registers must be restored afterwards. On the basic ARMv7-A architecture, the user mode registers can be retrieved by using `LDM` and `STM` instructions, or by switching to the SYS mode using `CPS` instructions, as the privileged SYS mode uses the user mode register bank. While the using `CPS` instructions leads to longer translated instruction sequences, we found it to perform slightly faster than using `LDM` and `STM` instructions. The virtualisation extensions offer a more elegant solution: they add special `MRS` and `MRS` instructions that copy values between banked and non-banked registers.

As noted by the authors of the ITRI hypervisor, we can improve our initial approach by eliminating unnecessary translation table switches in between consecutive unprivileged load and store instructions, as they often appear grouped together in the Linux kernel [44]. Notable examples of Linux kernel functions that contain such groups are `__copy_from_user` and `__copy_to_user`. Depending on the kernel version, other functions such as `restore_sigframe`, which is used when returning from a signal handler, may contain sequences that alternately perform an unprivileged load and a normal store; such instruction sequences cannot be optimised any further.

5.5. Coprocessor operations and register updates

A small fraction of the overhead in our benchmarks results from invoking operations on coprocessors and updating coprocessor registers on the system control coprocessor. As shown in Figure 4, this fraction is mostly limited to 5%. While the ARMv7-A architecture offers several instructions to access coprocessors (see Table 1), only `MCR` and `MRC` are functional for the system control coprocessor. Any other coprocessor access instruction that operates on the system control processor causes an undefined instruction trap. `MRC` instructions read from the coprocessor registers without causing any side effects; they are dealt with in the next section.

The `MCR` instruction is used to invoke an operation such as cleaning a cache line, and to update register values such as setting the address of the active translation table. Because the coprocessor register number is encoded as a constant within the `MCR` instruction, it is known at translation time. We can hence selectively inline equivalent instruction sequences specific to a coprocessor operation or register.

Coprocessor instructions that enable or disable caches and the MMU, or that reconfigure the MMU, always trap to the hypervisor. Some cache and TLB maintenance operations can however be inlined, depending on whether or not they are used by the hypervisor to keep track of a guest's modifications to its code and translation tables. Our hypervisor supports multiple such tracking mechanisms; some of which obsolete guest TLB maintenance, in which case a guest's TLB maintenance operations are removed by the translator. Some operations cannot be executed as is: e.g., the guest should not be allowed to invalidate the data caches without writing back dirty lines to memory, as this may discard hypervisor state. When cache maintenance operations can be executed unmodified[4], the hypervisor uses lightweight traps in their translations.

Some coprocessor operations can be executed regardless of the privilege level; this is the case for the deprecated coprocessor barrier instructions. We translate them to the newer equivalent instructions, without any kind of trap.

5.6. Special register accesses without side effects

The remainder of the traps to the interpreter consists of accesses to special registers, such as coprocessor registers reads using the `MRC` instruction, banked register reads using the `MRS` instruction, and writes to the `SPSR` using the `MSR` instruction. These accesses account for 1% to 9% of the traps to the interpreter. They do not cause any side effects beyond a simple register copy. We translate all such instructions into an equivalent sequence that accesses the guest's shadow registers using a lightweight trap.

[4] One example of such operation is a full data cache flush to memory. The hypervisor cannot perform a flush of guest memory alone, as the architecture cannot efficiently support such operation. On ARMv7-A, caches are either cleaned by cache line, or by set and way. Line-based operations have very fine granularity and invoking several such operations consecutively is very expensive, while sets and ways cannot be mapped to specific regions in the virtual address space.

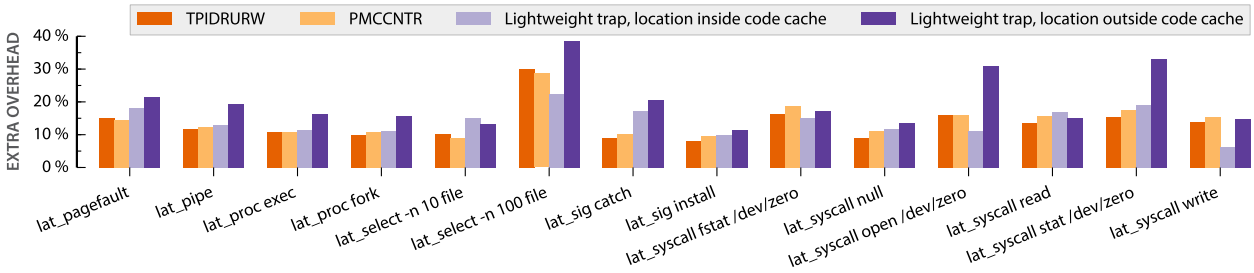

Figure 6: Overhead of register spilling techniques over a fully writable translation store, using at most one register

5.7. Summary

We explained how our hypervisor handles control flow by linking blocks together in the code cache, and how block linking affects exception handling. We showed that while a majority of the overhead is caused by control flow, sensitive instruction traps are not negligible. We therefore proposed optimisations to translate sensitive instructions to equivalent instruction sequences rather than to a trap for several ARM-specific instructions such as saving and restoring user mode registers, unprivileged loads and stores, and coprocessor accesses. In the next section, we evaluate how these optimisations reduce the DBT overhead.

6. Evaluation

As already mentioned, guest user-space code runs without intervention of the hypervisor, so in our evaluation we again focus on events related to mode switches and kernel operations, and we evaluate all our techniques using the same hardware and benchmarks as in Section 5.

We first configure the hypervisor for the different register spill techniques that we proposed in Section 4, to determine which technique, if any, performs best. To achieve a fair comparison between the different techniques, we enable only those optimisations that are not tied to a particular register spilling technique. We then proceed by demonstrating how our optimisations reduce the DBT-based overhead of the hypervisor, while using the best performing register spilling technique.

All lmbench benchmarks perform their own timing measurements. In order to ensure that these measurements are accurate, we grant the guest full and unsupervised access to a minimal set of hardware timers.

It is important to note that the used micro-benchmarks are stress tests that execute high-overhead events at a much higher rate than standard applications. So by no means are the reported results representative of real-world applications. They only serve our purpose of identifying the best DBT engine optimisations.

6.1. Register spilling techniques

We compare the results of running the benchmarks using our register spilling techniques to a set of results obtained with a fully writable code cache. To reduce the impact of external influences such as interrupts, we have executed each benchmark 100 times and we report averages.

Figure 6 shows the extra overhead incurred by our spilling techniques on each benchmark, when spilling and restoring at most one register. Spilling using lightweight traps to a location outside the code cache consistently performs worse, except in the `lat_select -n 10` and `lat_syscall read` benchmarks, where the difference with spilling inside the code cache is negligible. The extra overhead when spilling

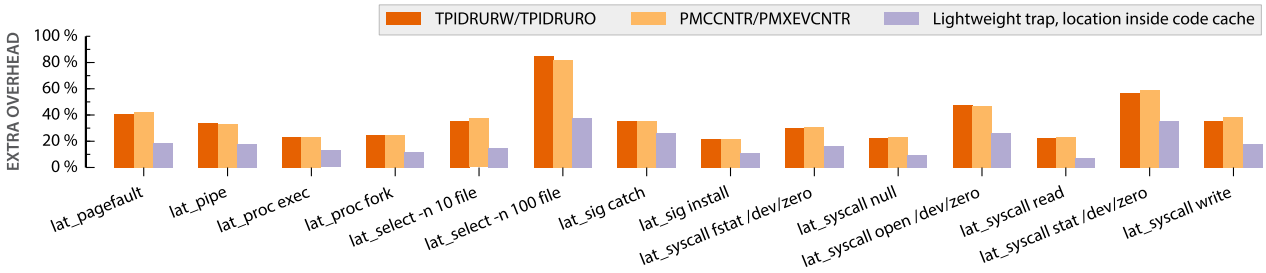

Figure 7: Overhead of register spilling techniques over a fully writable translation store, using at most two registers

to an external location is caused by the extra instructions required to construct the address of the spill location into a register, and by increased TLB pressure.

On the majority of the benchmarks, spilling to the performance counter register PMCCNTR incurs slightly more overhead than using the user-writable software thread ID register (TPIDRURW). We expected the performance counter registers to have higher access latencies than the thread ID register, as only the latter is intended to be updated frequently. As it turns out, the subtle differences between both measurements are only caused by the differences in maintenance requirements upon a guest context switch; the registers are otherwise equivalent.

On eight of the benchmarks, spilling inside the code cache performs only slightly worse than spilling to the thread ID register. This may indicate that the latency to access the thread ID register is of the same order of magnitude as the time taken to spill and restore using a lightweight trap. With lightweight traps, however, the majority of the overhead is caused by the trap to enter a privileged mode, rather than by the memory accesses that perform the actual spill and restore operations. Therefore, when spilling multiple registers at once, or when the spill location is accessed more frequently, we expect coprocessor latency to dominate. This also explains the bigger gap in performance between spilling inside the code cache and spilling to the thread ID register for the remaining benchmarks.

To investigate the impact of coprocessor latency, we have performed a second series of tests in which we have enabled the shadow stack optimisation. This optimisation requires spilling two registers, and accesses the spill locations several times. We do not provide measurements for spilling to a shared location using lightweight traps, since it is clear from our first set of measurements that this technique causes too much overhead. In order to support spilling two registers, we have adapted the remaining techniques as follows:

- When spilling to a private location inside the code cache, we resize the spill location to fit two words.
- When spilling to TPIDRURW, we use the user-read-only thread ID register (TPIDRURO) as secondary spill location. We do not require a second user-mode writable register as our shadow stack implementation only updates the secondary spill location from a privileged mode.
- When spilling to the performance counter registers, we use PMCCNTR as primary and PMXEVCNTR as secondary spill location.

As illustrated in Figure 7, coprocessor spilling incurs more than twice the overhead of spilling inside the code cache, when applied to our shadow stack. This overhead is caused by the access latency of the coprocessor registers.

Our results indicate that using a coprocessor register to spill can be useful in translations that only need to spill one register, and with few accesses to the spill location. In such scenarios, using coprocessor

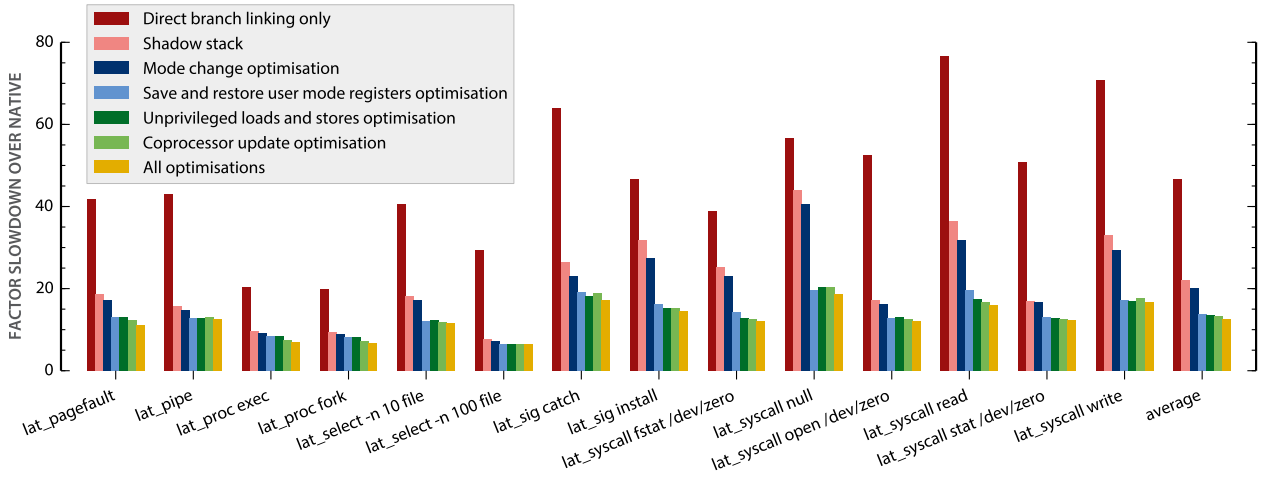

Figure 8: Slowdown over native for the different optimisations of the DBT engine

operations can be slightly faster and results in smaller translated code size. In other scenarios, however, spilling inside the code cache using lightweight traps should be preferred.

6.2. Optimisations to avoid traps to the DBT engine

We analyse the performance improvements gained from the optimisation techniques described in Section 5 on the same set of benchmarks from the lmbench suite. We have run each benchmark on different configurations of the hypervisor; we started with a configuration in which only direct branch linking is enabled, and then enabled our optimisations one by one, in the order they were discussed in Section 5. We have normalised the results based on a set of measurements on a native system, running the same kernel on the same hardware. All configurations unconditionally make use of lightweight traps to spill registers to a spill location inside the code cache.

Figure 8 provides an overview of our results. The unoptimised version of our hypervisor, visualised by the leftmost bar, causes slowdowns ranging from a factor 20 to a factor 76 compared to executing the benchmarks natively. We observe the lowest slowdown on larger benchmarks such as process creation (`lat_proc`) and `lat_select`. In the benchmarks for simple system calls the overhead of guest context switches dominates and causes bigger slowdowns. The optimisations proposed in this paper reduce the slowdown on the process creation benchmarks from a factor 20 to a factor 7 over native. On the the simple system call benchmarks, we achieve an overhead reduction from a factor 76 to a factor 15 over native.

When we investigated the major contributions to overhead reduction, we observed that enabling our shadow stack causes a significant speedup in all benchmarks. Mode change optimisations have little impact, as they only affect instructions that toggle interrupt masking; the majority of mode change instructions affects the processor mode and privilege level and must trap to the interpreter. Eliminating traps for instructions that save and store user registers yields significant speedups for the signal handler benchmarks (`lat_sig`), and for the system call benchmarks `lat_syscall null`, `read` and `write`, as predicted in Section 5.

Optimising unprivileged loads and stores results in a small speedup for the `lat_syscall read` benchmark. However, as this optimisation inlines translation table switches in translated code, it comes with significant space overhead. When applied to cold code, the optimisation has adverse effects on hardware

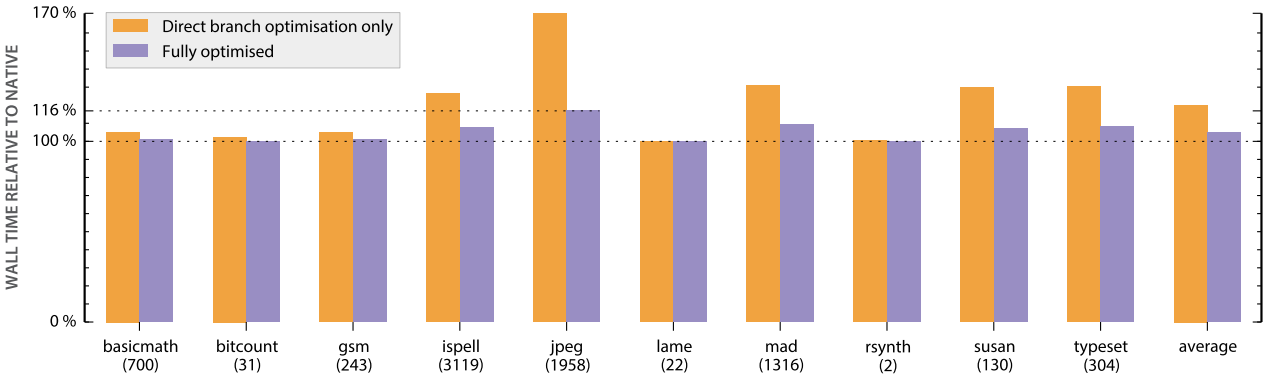

Figure 9: Virtualised execution time of selected mibench benchmarks, relative to native execution time

caches, resulting in a net slowdown in the `lat_pipe`, `lat_select -n 10`, `lat_syscall null` and `lat_syscall write` benchmarks. A similar issue manifests itself with inlining coprocessor update instructions in the `lat_sig catch` and `lat_syscall write` benchmarks. In the last step of our evaluation, we eliminated traps to the interpreter for accesses to coprocessor registers and banked registers. This optimisation causes minor speedups in all benchmarks.

Our results confirm that the architecture-specific optimisations presented in Section 5 are useful to reduce the overhead of our hypervisor beyond generic control flow optimisations: on average, we achieve a 45% reduction in slowdown when comparing the results from enabling the shadow stack to the fully optimised case.

6.3. Perceived slowdown

Micro-benchmarks are useful for studying the worst-case overhead in interactions between a guest's user applications and kernel. Many real applications however do not continuously perform system calls; the perceived slowdown while running real applications should therefore be lower than the slowdowns obtained from running micro-benchmarks.

In order to evaluate the perceived slowdown caused by our hypervisor, we run a selected number of benchmarks from the *mibench* version 1 suite [22]. Mibench is a benchmark suite that aims to be representative of commercial software normally run on embedded systems. We use wall time as a measure for perceived performance. We use the large input sets provided with mibench and run all benchmarks three times on the same Linux system: once natively, and to show the impact of our optimisations, virtualised without and with our optimisations. Similar to the measurements we did on micro-benchmarks, we do not disable block linking as doing so renders the system unusable.

Figure 9 shows that the perceived slowdown in user-space applications can reach almost a factor of two over native with our naive DBT engine, depending on the rate at which the application performs system calls. With our optimised DBT engine, however, the maximum perceived slowdown is limited to 16%.

The labels in Figure 9 also shows for each benchmark the rate at which the application performs system calls. `bitcount` and `rsynth` (speech synthesis) are small benchmarks that perform few system calls. `lame`, an MP3 encoder, is a computationally intensive and long-running benchmark. We can see that for `bitcount`, `rsynth` and `lame`, the very low rate of system calls results in almost negligible overhead. The `basicmath` benchmark is typically used to measure ALU performance. It performs small arithmetic operations and prints the result of every operation; every such print is a `write` system call. Similarly, the `gsm` benchmark,

an audio codec, mainly consists of `read` and `write` system calls. Both `basicmath` and `gsm` benchmarks still have a fairly low rate of system calls and therefore the overhead on both benchmarks on an optimised version of our hypervisor is also negligible. For the remaining benchmarks `ispell`, `jpeg`, `mad`, `susan` and `typeset` the rate of system calls is not correlated with the overhead, as they use a wider variety of system calls, some of which are more expensive than others. The remaining overhead in the `jpeg` benchmark is largely caused by memory management.

7. Conclusions

DBT remains useful in a handful of scenarios. However, all existing virtualisation solutions for the ARMv7-A architecture either rely on paravirtualisation or hardware extensions. We have therefore built the STAR hypervisor, the first open source software-only hypervisor for the ARMv7-A architecture that uses DBT to perform full system virtualisation.

We analysed existing techniques for user-space DBT on ARM, and argued that they are not directly suitable for full system virtualisation. We proposed new techniques for spilling registers to solve this problem, and we showed that the best technique to use depends on the usage scenario.

We measured the sources of DBT-based overhead for typical interactions between virtualised kernels and their user applications. As is typical with DBT, much of the overhead can be attributed to control flow, and eliminating traps caused by control flow yields significant speedups. However, we also showed that the remaining overhead can be further reduced by 45% on average, by using binary optimisations specific to ARMv7-A system virtualisation.

We found that a naive configuration of our hypervisor makes real applications run up to five times slower than native. With our optimisations, however, the maximum perceived slowdown is limited to 16%.

Our DBT engine can be further optimised, as the translator mostly operates on single instructions. For specific instruction patterns such as sequences of unprivileged loads and stores, considering multiple instructions at a time can be used to avoid unnecessary, potentially expensive, operations. Similar optimisation opportunities exist with other instructions, but we expect the impact on run-time performance to be of little significance.

Acknowledgements

The authors would like to thank ARM for providing the necessary development tools and excellent support.

The primary author of this paper was supported by the agency for Innovation by Science and Technology (IWT) under terms of a personal research grant (grant number 093488). Part of the research results presented in this paper were obtained in the context of the EURO-MILS project, which has received funding from the European Union's Seventh Framework Programme under grant agreement number ICT-318353.

References

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XII, pages 2–13, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0.

[2] Zach Amsden, Daniel Arai, Daniel Hecht, Anne Holler, and Pratap Subrahmanyam. VMI: an interface for paravirtualization. In *Proceedings of the Linux Symposium*, volume 2 of *2006 Linux Symposium*, pages 371–386, July 2006.

[3] ARM Limited. *ARM® Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. ARM Limited, ARM DDI 0406C.c (ID051414) edition, May 2014.

[4] François Armand and Michel Gien. A practical look at micro-kernels and virtual machine monitors. In *6th IEEE Consumer Communications and Networking Conference*, CCNC 2009, pages 1–7, Piscataway, New Jersey, USA, January 2009. IEEE.

[5] François Armand, Gilles Muller, Julia Laetitia Lawall, and Jean Berniolles. Automating the port of Linux to the VirtualLogix hypervisor using semantic patches. In *4th European Congress ERTS Embedded Real Time Software*, ERTS 2008, pages 1–7, January 2008.

[6] B-Labs Ltd. Codezero project overview, 2012. URL `https://web.archive.org/web/20120101123359/http://www.l4dev.org/codezero_overview`.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Operating Systems Review*, 37:164–177, October 2003. ISSN 0163-5980.

[8] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. The VMware mobile virtualization platform: is that a hypervisor in your pocket? *SIGOPS Operating Systems Review*, 44:124–135, December 2010. ISSN 0163-5980.

[9] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track*, USENIX '05, pages 41–46, Berkeley, CA, USA, 2005. USENIX Association.

[10] Prashanth Bungale. ARM virtualization: CPU & MMU issues, December 2010. URL `https://labs.vmware.com/download/68`.

[11] Prashanth P. Bungale and Chi-Keung Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual Execution Environments*, VEE '07, pages 137–147, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1.

[12] Jiunn-Yeu Chen, Bor-Yeh Shen, Quan-Huei Ou, Wuu Yang, and Wei-Chung Hsu. Effective code discovery for arm/thumb mixed isa binaries in a static binary translator. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, pages 19:1–19:10, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4799-1400-5.

[13] Cristina Cifuentes and Vishv M. Malhotra. Binary translation: static, dynamic, retargetable? In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 340–349. IEEE, November 1996.

[14] CORDIS. Report on the EC Workshop on Virtualisation, Consultation Workshop "Virtualisation in Computing", September 2009. URL `ftp://ftp.cordis.europa.eu/pub/fp7/ict/docs/computing/report-on-the-ec-workshop-on-virtualisation_en.pdf`.

[15] Christoffer Dall and Jason Nieh. KVM for ARM. In *Proceedings of the 12th Annual Linux Symposium*, pages 45–56, July 2010.

[16] Christoffer Dall and Jason Nieh. KVM/ARM: Experiences building the Linux ARM hypervisor. Technical Report CUCS-010-13, Columbia University, Department of Computer Science, April 2013.

[17] Christoffer Dall and Jason Nieh. KVM/ARM: The design and implementation of the Linux ARM hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5.

[18] Scott W. Devine, Lawrence S. Rogel, Prashant P. Bungale, and Gerald A. Fry. Virtualization with in-place translation, December 2009. URL http://www.google.com/patents/US20090300645. US Patent App. 12/466,343; original assignee: VMware Inc.

[19] Jiun-Hung Ding, Chang-Jung Lin, Ping-Hao Chang, Chieh-Hao Tsang, Wei-Chung Hsu, and Yeh-Ching Chung. ARMvisor: System virtualization for ARM. In *Proceedings of the Linux Symposium*, pages 93–107, July 2012.

[20] Marc Duranton, Sami Yehia, Bjorn De Sutter, Koen De Bosschere, Albert Cohen, Babak Falsafi, Georgi Gaydadjiev, Manolis Katevenis, Jonas Maebe, Harm Munk, Nacho Navarro, Alex Ramirez, Olivier Temam, and Mateo Valero. The HiPEAC vision, 2010. URL http://www.hipeac.net/roadmap.

[21] Daniel R. Ferstay. Fast secure virtualization for the ARM platform. Master's thesis, The University of British Columbia, Faculty of Graduate Studies (Computer Science), June 2006.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7803-7315-4. doi: 10.1109/WWC.2001.15. URL http://dx.doi.org/10.1109/WWC.2001.15.

[23] Kim Hazelwood and Artur Klauser. A dynamic binary instrumentation engine for the arm architecture. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 261–270, New York, NY, USA, 2006. ACM. ISBN 1-59593-543-6. doi: 10.1145/1176760.1176793. URL http://doi.acm.org/10.1145/1176760.1176793.

[24] Thomas Heinz and Reinhard Wilhelm. Towards device emulation code generation. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '09, pages 109–118, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-356-3.

[25] Gernot Heiser. The Motorola Evoke QA4—a case study in mobile virtualization. Technology white paper, Open Kernel Labs, July 2009.

[26] Gernot Heiser and Ben Leslie. The okl4 microvisor: convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, APSys '10, pages 19–24, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0195-4.

[27] Jason D. Hiser, Daniel Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7. doi: 10.1109/CGO.2007.10. URL http://dx.doi.org/10.1109/CGO.2007.10.

[28] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[29] Perry L. Hung. Varmosa: just-in-time binary translation of operating system kernels. Master's thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 2009.

[30] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones. In *5th IEEE Consumer Communications and Networking Conference*, CCNC 2008, pages 257–261, Piscataway, New Jersey, USA, January 2008. IEEE. ISBN 978-1-4244-1457-4.

[31] IBM. PowerVM Lx86 for x86 Linux applications, July 2011. URL http://www.ibm.com/developerworks/linux/lx86/index.html.

[32] Hiroaki Inoue, Akihisa Ikeno, Masaki Kondo, Junji Sakai, and Masato Edahiro. VIRTUS: a new processor virtualization architecture for security-oriented next-generation mobile terminals. In *Proceedings of the 43rd annual Design Automation Conference*, DAC '06, pages 484–489, New York, NY, USA, 2006. ACM. ISBN 1-59593-381-6.

[33] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 34–42, New York, NY, USA, 1991. ACM. ISBN 0-89791-394-9. doi: 10.1145/115952.115957. URL `http://doi.acm.org/10.1145/115952.115957`.

[34] Robert Kaiser and Stephan Wagner. The PikeOS concept – history and design. Whitepaper, SYSGO AG, January 2008. URL `http://www.sysgo.com/nc/news-events/document-center/whitepapers/pikeos-history-and-design-jan-2008/`.

[35] Sung-Min Lee, Sang-Bum Suh, and Jong-Deok Choi. Fine-grained I/O access control based on Xen virtualization for 3G/4G mobile devices. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 108–113, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0002-5.

[36] Joshua LeVasseur, Volkmar Uhlig, Yaowei Yang, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: soft layering for virtual machines. In *13th Asia-Pacific Computer Systems Architecture Conference*, ACSAC 2008, pages 1–9, Los Alamitos, California, USA, August 2008. IEEE Computer Society. ISBN 978-1-4244-2682-9.

[37] Larry McVoy and Carl Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1268299.1268322`.

[38] Ryan W. Moore, José A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser. Addressing the challenges of DBT for the ARM architecture. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '09, pages 147–156, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-356-3.

[39] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. Formal virtualization requirements for the arm architecture. *Journal of Systems Architecture*, 59(3):144–154, March 2013. ISSN 1383-7621.

[40] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17:412–421, July 1974. ISSN 0001-0782.

[41] Red Bend Software. vLogix Mobile for mobile virtualization, 2014. URL `https://www.redbend.com/en/products-solutions/mobile-virtualization/vlogix-mobile-for-mobile-vitrualization`.

[42] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Operating Systems Review*, 42: 95–103, July 2008. ISSN 0163-5980.

[43] Alexey Smirnov. KVM-ARM hypervisor on Marvell Armada-XP board. Talk at Cloud Computing Research Center for Mobile Applications (CCMA) Low Power Workshop (Taipei), June 2012. URL `ftp://220.135.227.11/Low-Power%20Workshop%202012%20June%204th%20PDF/Low-Power%20Workshop%202012%20June%204th%20PDF/07-CCMA-workshop-taipei.pdf`.

[44] Alexey Smirnov, Mikhail Zhidko, Yingshiuan Pan, Po-Jui Tsao, Kuang-Chih Liu, and Tzi-Cker Chiueh. Evaluation of a server-grade software-only arm hypervisor. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 855–862, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5028-2.

[45] Brad Smith. ARM and Intel battle over the mobile chip's future. *Computer*, 41(5):15–18, May 2008. ISSN 0018-9162.

[46] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 2005. ISBN 1558609105.

[47] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth P. Bungale. Hdtrans: An open source, low-level dynamic instrumentation system. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 175–185, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-8. doi: 10.1145/1134760.1220166. URL `http://doi.acm.org/10.1145/1134760.1220166`.

[48] SYSGO AG. PikeOS comes with full virtualization for Renesas R-CAR H2, June 2014. URL `http://www.sysgo.com/news-events/press/press/details/article/pikeos-comes-with-full-virtualization-for-renesas-r-car-h2/`.

[49] TRANGO Virtual Processors. Virtualization for mobile, 2009. URL `https://web.archive.org/web/20090103050359/http://www.trango-vp.com/markets/mobile_handset/usecases.php`.

[50] Prashant Varanasi and Gernot Heiser. Hardware-supported virtualization on ARM. In *Proceedings of the 2nd ACM SIGOPS Asia-Pacific Workshop on Systems (Shangai, China)*, APSys 2011, pages 11:1–11:5, New York, NY, USA, July 2011. ACM.

[51] VirtualLogix. VLX for mobile handsets, 2009. URL `https://web.archive.org/web/20090308134358/http://www.virtuallogix.com/products/vlx-for-mobile-handsets.html`.

[52] Youfeng Wu, Shiliang Hu, Edson Borin, and Cheng Wang. A HW/SW co-designed heterogeneous multi-core virtual machine for energy-efficient general purpose computing. In *9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2011, pages 236–245, Piscataway, New Jersey, USA, April 2011. IEEE Computer Society. ISBN 978-1-61284-355-1.

[53] Xen Project. Xen ARM with virtualization extensions whitepaper, April 2014. URL `http://wiki.xen.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper`.

[54] Xvisor. eXtensible Versatile hypervISOR. URL `http://xhypervisor.org`.