MANCHESTER
1824
The University of Manchester

The University of Manchester Research

HyperMAMBO-X64: Using Virtualization to Support High-performance Transparent Binary Translation

DOI:
10.1145/3050748.3050756

Document Version
Accepted author manuscript

Link to publication record in Manchester Research Explorer

Citation for published version (APA):
d'Antras, A., Gorgovan, C., Garside, J., Goodacre, J., & Luján, M. (2017). HyperMAMBO-X64: Using Virtualization to Support High-performance Transparent Binary Translation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (pp. 228-241). (VEE '17). Association for Computing Machinery. https://doi.org/10.1145/3050748.3050756

Published in:
Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester's Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.

Download date:26. Aug. 2022

HyperMAMBO-X64: Using Virtualization to Support High-Performance Transparent Binary Translation

Amanieu d'Antras Cosmin Gorgovan Jim Garside John Goodacre Mikel Luján

School of Computer Science, University of Manchester

{bdantras,cgorgovan,jgarside,jgoodacre,mikel}@cs.manchester.ac.uk

Abstract

Current computer architectures — ARM, MIPS, PowerPC, SPARC, x86 — have evolved from a 32-bit architecture to a 64-bit one. Computer architects often consider whether it could be possible to eliminate hardware support for a subset of the instruction set as to reduce hardware complexity, which could improve performance, reduce power usage and accelerate processor development. This paper considers the scenario where we want to eliminate 32-bit hardware support from the ARMv8 architecture.

Dynamic binary translation can be used for this purpose and generally comes in one of two forms: application-level translators that translate a single user mode process on top of a native operating system, and system-level translators that translate an entire operating system and all its processes.

Application-level translators can have good performance but is not totally transparent; system-level translators may be 100% compatible but performance suffers. HyperMAMBO-X64 uses a new approach that gets the best of both worlds, being able to run the translator as an application under the hypervisor but still react to the behavior of guest operating systems. It works with complete transparency with regards to the virtualized system whilst delivering performance close to that provided by hardware execution.

A key factor in the low overhead of HyperMAMBO-X64 is its deep integration with the virtualization and memory management features of ARMv8. These are exploited to support caching of translations across multiple address spaces while ensuring that translated code remains consistent with the source instructions it is based on. We show how these attributes are achieved without sacrificing either performance or accuracy.

1. Introduction

ARM [33] is a general purpose architecture which is widely used in both embedded systems and consumer devices such as phones, tablets and TVs. While ARM has traditionally been a 32-bit architecture, the ARMv8 version of the architecture [23] introduced a new 64-bit execution mode and instruction set, called *AArch64*. This 64-bit ISA has double the number of general-purpose registers as the previous architecture and extends them to 64 bits, as well as extending the address space width to 64 bits.

While the 64-bit instruction set has many benefits, there is a large ecosystem of existing 32-bit applications which need to be able to run on ARMv8 systems. Most of the current generation of ARMv8 processors is capable of running legacy 32-bit ARM code directly in *AArch32* mode, but maintaining this support comes at a cost in hardware complexity, power usage and development time. For example, Cavium does not include hardware support for AArch32 in their ThunderX processors for this reason.

One solution to this issue is to use dynamic binary translation to translate AArch32 code into AArch64 code. A Dynamic Binary Translator (DBT) generally comes in one of two forms: application-level translators which translate a single user mode process running under a native operating system, and system-level translators which translate an entire operating system and all its processes. While the former have been able to achieve performance levels approaching that of native execution, they suffers from transparency issues: a translated 32-bit process will still appear as a 64-bit process to the operating system, and tools such as debuggers will see the state of the translator rather than that of the translated process. System-level translators avoid these issues since all processes are running natively from the point of view of the translated OS, but these tend to have lower performance than similar application-level translators.

A significant portion of the overhead of system-level translators comes from the need to emulate the Memory Management Unit (MMU) of the target architecture. This requires mapping the guest OS page table into the format of the host architecture and keeping these mappings consistent when the guest modifies its page tables. Application-level

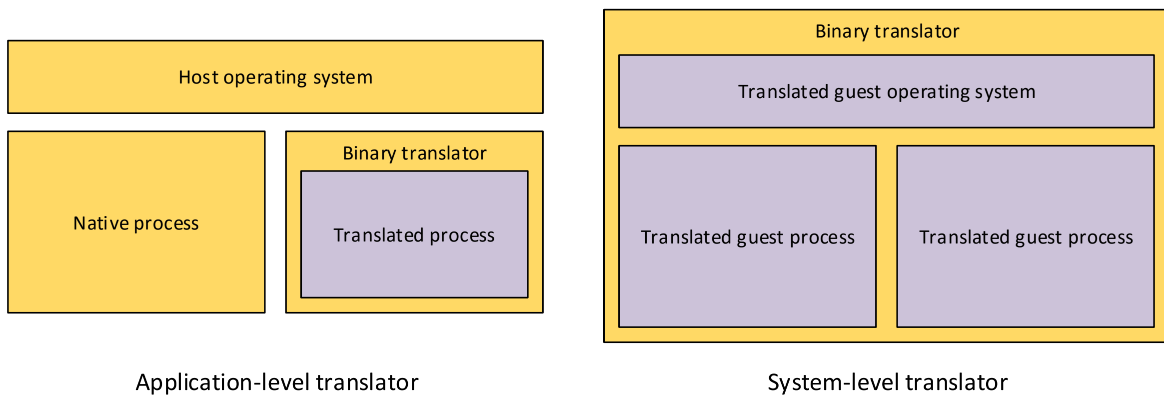

Figure 1: Overview of application-level and system-level translators

translators do not suffer from this overhead since page tables are managed by the host OS using the native MMU.

We propose HyperMAMBO-X64, a new type of DBT which is a hybrid of these two existing types, preserving the best attributes of each. HyperMAMBO-X64 extends an existing hypervisor to allow an AArch64 guest operating system to run AArch32 user mode processes even when the underlying processor only supports AArch64. This is achieved by having the hypervisor trap attempts by the guest OS to switch to AArch32 user mode and running the AArch32 code under a DBT. The DBT returns control to the guest OS once an exception (syscall, page fault, interrupt) occurs by simulating an exception coming from AArch32 mode. This process is completely transparent to the guest OS: from its point of view, the user process was executing natively in AArch32 mode. Yet, since the page tables are entirely controlled by the guest OS which runs natively, HyperMAMBO-X64 can achieve similar levels of performance to application-level translators.

A key challenge in the implementation of HyperMAMBO-X64 is keeping the translated code generated by the DBT consistent with any changes to the source AArch32 instructions. These modifications can come in the form of page table modifications, such as loading or unloading a shared library, or direct modifications to the underlying code, such as in a JIT compiler. HyperMAMBO-X64 handles these by exploiting several features of the ARMv8 architecture and virtualization extensions. We associate each translated code fragment with a user-mode process in the guest using the address space identifier (ASID) tags which are used by the TLB hardware. Modifications to the address space of a process are detected by trapping all TLB flush instructions to the hypervisor, which can then invalidate any translations affected by the changed virtual memory mappings. Finally, memory pages from which code has been translated are write-protected by the hypervisor to detect any modifications.

We built a prototype of HyperMAMBO-X64 on top of the Linux Kernel Virtual Machine (KVM) [17] hypervisor and evaluated its performance by running SPEC CPU2006 and several microbenchmarks. Our results on SPEC CPU2006 show that HyperMAMBO-X64 is able to match the performance of MAMBO-X64 [18, 19], an equivalent application-level DBT which also translates from AArch32 to AArch64. We measured a geometric mean performance *improvement* of about 1 % by running the AArch32 version of SPEC CPU2006 under HyperMAMBO-X64 compared to running it natively on the ARMv8 processor.

Some existing system-level translators use techniques similar to those used by HyperMAMBO-X64 to maintain code cache consistency. Such systems include MagiXen [13] and PinOS [11] which both translate x86 operating systems under the Xen hypervisor. A significant source of overhead in these systems comes from the need to emulate the page tables used by the guest operating system and detect changes to virtual memory mappings which would affect translated code. HyperMAMBO-X64 is able to avoid this overhead by running the guest operating system natively and exploiting ARM hardware virtualization features to track page table modifications.

The rest of this paper is organized as follows. Section 2 presents an overview of binary translation technology and the ARM architecture. Section 3 describes the design and implementation of the HyperMAMBO-X64 system. Section 4 presents our performance results on a selection of benchmarks. Section 5 summarizes some related works. Section 6 discusses several ways in which HyperMAMBO-X64 can be extended and Section 7 concludes the paper.

2. Background

This section reviews the basic concepts of binary translation and the ARM virtualization extensions.

2.1 Binary translation

Binary translation is a technology which allows a program to be transparently translated, instrumented or modified at the machine code level. It has numerous applications, such as dynamic instrumentation [27, 34], program analysis [32, 43], virtualization [1, 42] and instruction set translation [7]. A binary translator does not need access to the source code of a program, which makes it particularly useful in cases where source code is not available or is not portable enough to be simply recompiled. In the context of this paper, we specifically refer to *cross-ISA* binary translation, which involves reading a sequence of instructions for a *guest ISA* and translating them into an equivalent code sequence for a *host ISA*.

A Dynamic Binary Translator (DBT) translates code only as it is about to be executed rather than ahead of time. Rather than translating instructions individually, a DBT usually translates instructions in blocks, called *fragments*. Since some code, such as loop and function bodies, is likely to be executed many times, it is advantageous to preserve translated fragments so that they can be used again, instead of re-translating each time they are encountered. Rather than modifying the program code, translated fragments are stored in a *code cache*, separate from the original instructions.

DBTs can generally be split into two categories, shown in Figure 1, depending on the type of environment that they inhabit:

Application-level translators These translators work at the level of a single user-mode process, running an application compiled for a guest ISA on top of an operating system for the host ISA. In addition to translating all the instructions executed by the user-mode process, such a DBT also needs to translate the operating system Application Binary Interface (ABI), which can have significant variations from one ISA to another. This is usually done by intercepting all interactions between the translated application and the host OS, such as system calls and signals, and translating them from the format of the guest ABI to that of the host ABI. Examples of DBTs in this category are QEMU [7], Aries [44], IA-32 EL [5], FX!32 [14, 25], Rosetta [2], MAMBO-X64 [18, 19] and StarDBT [40].

System-level translators These translators work at the level of a complete system and, effectively, simulate a virtual machine running on a foreign architecture. These systems tend to be more complex than application-level translators because they need to be able to translate a larger portion of the guest instruction set. Whereas an application-level translator only needs to support user-mode, unprivileged instructions, a system-level translator must support the full guest ISA including all privileged instructions and related operations. An important part of this is efficiently simulating the guest ISA's virtual memory architecture, which involves translating page tables from one format to another and correctly handling page table modifications. Examples of DBTs in this category are MagiXen [13], Transitive QuickTransit [38], QEMU [7], Transmeta's Code Morphing Software (CMS) [20] and Nvidia's Project Denver [9].

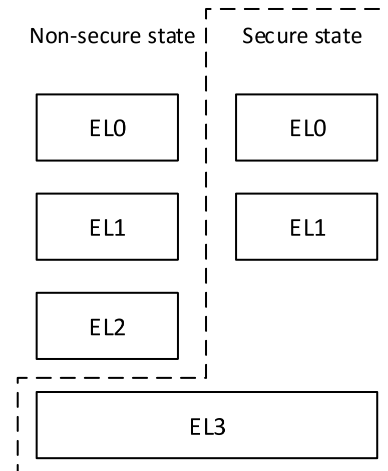

Figure 2: ARMv8 exception levels

2.2 ARMv8 virtualization extensions

The traditional ARM architecture is not classically virtualizable [30] because it contains several sensitive instructions that have observably different behavior depending on the current privilege level [29]. While there have been several attempts to support virtualization for the ARM architecture through hardware modifications [8], binary rewriting [35] or paravirtualization [21], these have not seen widespread use. ARM introduced an optional virtualization extension in ARMv7 which makes the ARM architecture classically virtualizable through the introduction of a hypervisor mode which executes at a higher privilege level than the existing privileged execution modes.

This virtualization capability was carried over to ARMv8, which also streamlined the various ARM execution modes. Figure 2 shows the four execution modes supported by ARMv8, called *exception levels* and numbered from EL0 to EL3:

- EL3 is the most privileged mode in ARMv8, called the "secure monitor" mode, and is part of the ARM TrustZone extension. This mode allows switching between the "secure world" and "non-secure world". TrustZone works by only allowing software access to secure RAM and secure peripherals when the processor is running in EL3 or in secure EL1/EL0. TrustZone is designed for

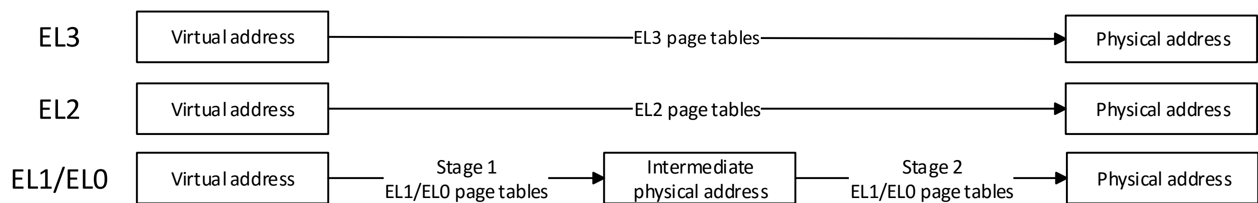

Figure 3: ARMv8 virtual memory address translation for different exception levels

specialized applications such as digital rights management and is outside the scope of this paper.

- EL2 is an execution mode designed for hypervisors: it supports an extensive set of configuration registers that allow it to trap certain classes of privileged or sensitive instructions to EL2 for special handling. These registers also allow configuring the exception level at which various exceptions are handled. This can be used to handle hardware interrupts in the hypervisor while letting the guest kernel handle system calls directly.
- EL1 is a privileged execution mode typically used by operating system kernels. On a system without virtualization extensions this would be the level which manages hardware peripherals directly, but inside a virtual machine it will manage virtual peripherals that are emulated by the hypervisor instead. EL1 also supports many system registers to configure various aspects of how user-mode processes execute in EL0.
- EL0 is the least privileged execution mode, which is intended for the execution of normal user-mode processes. This mode has no access to privileged instructions for operations such as page table and TLB management instructions, which means that it must perform system calls to EL1 or above for such operations.

Transitions between exception levels are only possible through exceptions (interrupts, system calls, page fault, etc.) and the exception return (`ERET`) instruction. All exception levels except EL0 define an *exception vector* which allows them to handle exceptions coming from the current exception level or any level below it. Similarly, the `ERET` instruction is allowed to switch to an exception level equal to or below the current level. The specific exception level at which a particular exception is handled is determined by special configuration registers that are only accessible to higher exception levels.

Architectural support for the legacy 32-bit ARM instruction set is implemented by allowing each exception level to run in either the 32-bit AArch32 mode or the 64-bit AArch64 mode. A transition between AArch32 and AArch64 is only possible through an exception or exception return, with two restrictions:

1. Only EL0 and EL1 support AArch32 mode. EL2 and EL3 must run in AArch64 mode.
2. If an exception level is running in AArch32 mode then all exception levels below it must also run in AArch32 mode. This means that while a 64-bit OS can run 32-bit user mode processes and a 64-bit hypervisor can run 32-bit virtual machines, it is not possible for a 32-bit OS to run a 64-bit user mode process.

The separation between exception levels is further supported by the ARMv8 virtual memory architecture, which supports specialized address translation mechanisms depending on the current exception level, as shown in Figure 3. EL2 and EL3 each has its own page table base register, which is used by the processor when executing code in one of those exception levels. Code executing at EL0 and EL1 share the same set of page tables but use a more complicated address translation system which involves two sets of page tables. Stage 1 page tables controlled by EL1 are used to transform virtual addresses into *intermediate physical addresses* (IPAs), while stage 2 page tables controlled by EL2 are used to transform IPAs into physical addresses. This system allows a hypervisor to control the physical memory used by a guest operating system transparently, giving the guest kernel the illusion that it has full access to its physical memory.

To avoid the need to perform a full TLB flush when context switching, the ARM architecture has support for *TLB tagging*. This involves associating two pieces of information with each TLB entry: a 16-bit *address space identifier* (ASID) and a 16-bit *virtual machine identifier* (VMID). The ASID is set by the kernel in EL1 when switching from one user-mode process to another by changing the stage 1 page tables. The VMID is set by the hypervisor in EL2 when switching from one virtual machine to another by changing the stage 2 page tables. This system effectively associates each set of stage 1 page tables with an ASID and each set of stage 2 page tables with a VMID.

TLB invalidation is performed using privileged instructions which come in several variants: a TLB flush can be directed to either remove only TLB entries relating to a specific virtual address or to remove TLB entries for all virtual

addresses. A flush can be further restricted to remove only TLB entries associated with a specific ASID or VMID. The ARM architecture requires that ASIDs and VMIDs be consistent across all processors in a system, which allows TLB flushes to be broadcast across processors.

We exploit the TLB features of the ARM architecture to keep track of the different user-mode processes in a virtual machine and handle code cache invalidation efficiently.

3. HyperMAMBO-X64

We propose using dynamic binary translation to translate AArch32 instructions into AArch64 code, which would open a path for future ARMv8 processors to remove hardware support for the legacy 32-bit instruction set while retaining the ability to run AArch32 applications. We developed a binary translation system, called HyperMAMBO-X64, which integrates with a hypervisor to allow a virtual machine to run AArch32 user mode processes transparently under an AArch64 kernel even when the underlying processor does not support AArch32 mode.

3.1 Proposed approach

As described in Section 2.1, binary translators generally fit into one of two categories, application-level translators and system-level translators, each of which has benefits and disadvantages:

System-level translators are the most flexible since they emulate a full virtual machine, including a full operating system. This allows a single translator to run any guest OS without needing specialized support. However this flexibility comes at a significant cost in performance, in particular due to the need to handle virtual memory address translation within the guest. This requires either translating guest page tables to the host page table format [12] or performing guest page table walks in software and caching the results in a software TLB [37].

While application-level translators are limited to translating a single user mode process, they do not suffer from many of the disadvantages of system-level translators because they work purely in a virtual address space managed by the host OS. An application-level translator can also make assumptions based on the OS ABI, such as determining which memory locations are read-only[1], and optimizing the generated code based on those assumptions. Another advantage is the ability to recognize memory locations that are mapped from an on-disk file and using this information to support persistent code caches [10, 31] which allow faster start-up and can be shared among multiple processes. The main disadvantage of this type of translators is that they are not fully *transparent*. For example, in the case of AArch32 to AArch64 translation, a translated process would still appear as a 64-bit process to the operating system, and debuggers attached to that process would be debugging the translator itself rather than the translated process.

HyperMAMBO-X64 is a hybrid of these two types of translator: like a system-level translator, it controls a guest operating system from a hypervisor running at EL2, but it only translates AArch32 code running at EL0 as an application-level translator.

The basic principle of HyperMAMBO-X64 is to allow 64-bit guest kernels and 64-bit user-mode processes to run natively on the processor in AArch64 mode, while trapping attempts by the 64-bit kernel to switch to AArch32 user mode. When such an attempt is detected, HyperMAMBO-X64 will run the 32-bit process using binary translation until an exception (such as a system call) occurs, at which point HyperMAMBO-X64 will return to the guest kernel. All of this is done transparently: from the point of view of the guest kernel, the user process was running natively in AArch32 mode.

The binary translator part of HyperMAMBO-X64 is based on MAMBO-X64 [18, 19], an application-level translator designed to translate AArch32 Linux programs into AArch64 code. We have adapted the code of MAMBO-X64 to work in a hypervisor environment without any dependency on either the host or guest OS.

The main disadvantage of our approach compared to a full system-level translator is that we require the guest kernel to run in AArch64 mode. However, this problem is not a significant drawback because most AArch64 kernels, such as Linux, have strong support for running AArch32 user mode applications. This in turn makes it easy to replace an AArch32 kernel with an AArch64 one since no other changes are required to the system: all existing AArch32 applications will still be able to run on the new kernel.

Similarly, a disadvantage of our approach compared to application-level translators is its inability to recognize memory mapped files in a translated process since that information is only known to the guest operating system. However there exist other persistent code caching techniques which do not require this information and, instead, keep a cache of translated code indexed by a hash of the code rather than the module it was loaded from [41], albeit at a cost in performance.

In addition to providing a platform for running AArch32 programs on a processor which only supports AArch64, HyperMAMBO-X64 can be used to support more exotic systems:

- ARM's big.LITTLE architecture [3] combines a cluster of high-performance "big" cores with a cluster of low-power "LITTLE" cores. This allows for higher performance and lower power consumption than similar homogeneous architectures [16]. While both clusters typically support the same ISA to allow an operating system to migrate processes from one cluster to another trans-

[1] Simple page table permissions are not a sufficient guarantee that data at a certain address is constant due to the possibility of writable aliases of that memory.

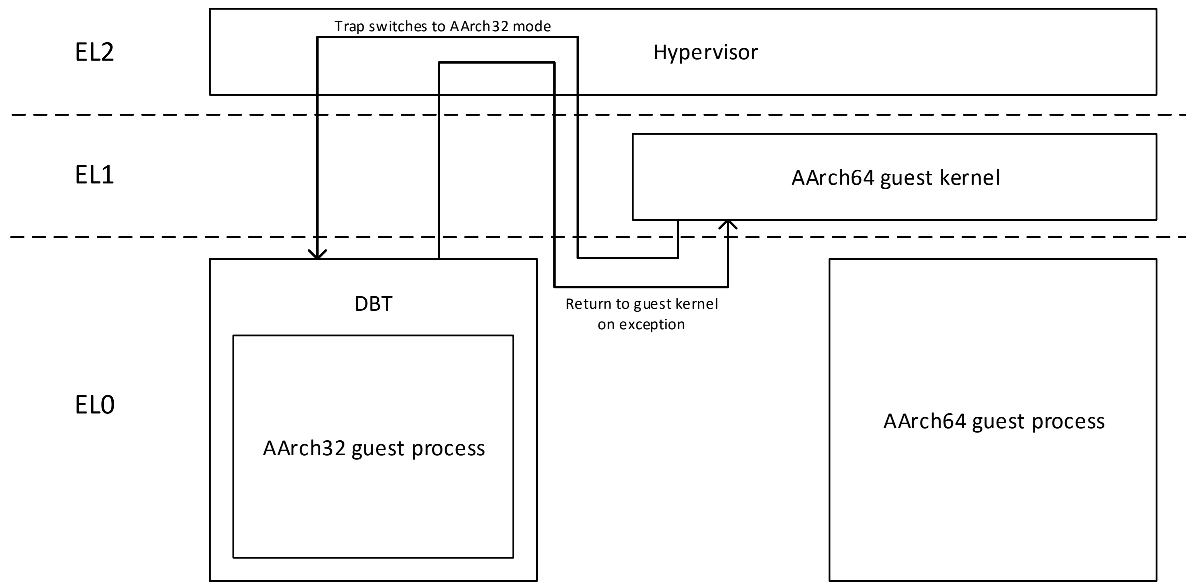

Figure 4: Overall architecture of HyperMAMBO-X64

parently, HyperMAMBO-X64 would allow relaxing this restriction. For example, HyperMAMBO-X64 would allow a “LITTLE” core to eliminate hardware support for AArch32 and reduce its power usage, while still allowing an operating system to freely migrate AArch32 tasks between the two core clusters. HyperMAMBO-X64 would then only perform translation on the “LITTLE” cores while running AArch32 code natively on the “big” cores.

- ARM-based servers are a growing market and the availability of hardware virtualization is a key factor driving this growth. While 64-bit ARM servers are starting to see widespread use, many need to run legacy AArch32 applications. The need to keep supporting AArch32 applications is a barrier to the adoption of AArch64-only processors, but this barrier can be eliminated by using HyperMAMBO-X64 to assist the migration of virtual machines to a physical server with AArch64-only processors. HyperMAMBO-X64 can even be used to support live migration of a virtual machine to an AArch64-only processor, as long as the virtual machine is running an AArch64 kernel.

3.2 Architecture

Figure 4 gives an overview of the different components comprising HyperMAMBO-X64. The basic principle is simple: the hypervisor traps attempts by a guest kernel to switch to AArch32 user mode and injects a binary translator into the address space of the 32-bit process. Control is then transferred to the DBT which will translate and execute the AArch32 code in that process.

The DBT and the translated code it generates run in EL0 under the direct control of the hypervisor in EL2. This is necessary to ensure that memory accesses performed by the translated code use the correct set of permissions and that any permission faults are detected correctly.

Execution of the translated process continues until an exception occurs. This can be a synchronous exception caused by the translated program itself, such as a system call or a page fault, or an asynchronous exception caused by a virtual interrupt from the hypervisor. In either case, control needs to be returned to the guest OS so that it can handle the exception as if it came directly from AArch32 mode.

Upon regaining control, the guest OS expects to see the register state of the underlying AArch32 process rather than the AArch64 register state of the translated code. HyperMAMBO-X64 reuses the signal handling mechanisms of MAMBO-X64 to recover the AArch32 register state when an exception occurs:

- Some exceptions are detected at translation time, such as system calls and undefined instructions. In those cases, specialized context recovery code can be compiled directly into the translated fragment.
- Runtime faults such as data aborts are handled by maintaining metadata for all potentially fault-generating instructions, such as load and store instructions. For each fragment, HyperMAMBO-X64 builds a table containing the addresses of these instructions and information on how to recover the AArch32 register context if that instruction generates a fault.

- Virtual interrupts are generated by the hypervisor to notify the guest OS of certain events such as virtual device interrupts. Keeping metadata for all translated instructions is impractical since such interrupts can occur at any point in the translated code. HyperMAMBO-X64 therefore uses a different strategy: after an interrupt is caught, the interrupted code is resumed with interrupts disabled until it reaches the end of the current fragment. The AArch32 context can then be recovered from the *fragment metadata* used to link fragments together.

Control is returned to the guest kernel through a hypervisor call which takes an AArch32 register context as a parameter. The hypervisor will restore the page tables to their original state and simulate an exception entry in the guest OS, which will make the guest kernel see an exception coming from an AArch32 process.

3.3 Memory management

In addition to the usual RAM and memory-mapped virtual devices usually present in a virtual machine, HyperMAMBO-X64 includes an area of RAM reserved for use by the DBT in the guest physical address space which is separate from the main RAM used by the guest OS. Each virtual machine managed by HyperMAMBO-X64 has a separate instance of this memory area, into which the DBT image is loaded when the virtual machine is created, and which holds all the runtime data managed by the DBT, including its code cache.

A key feature of HyperMAMBO-X64 is its complete transparency with regards to the guest OS: at no point does HyperMAMBO-X64 modify the contents of the RAM used by the guest OS, except through the actions of a translated AArch32 process. This presents an issue for injecting the DBT into the address space of the target process since it must be done without modifying the page tables of the guest OS. HyperMAMBO-X64 instead uses a *shadow top-level page table* in DBT RAM which contains the virtual memory mappings used while running a process under the DBT.

When the hypervisor starts running a process under the DBT, it will initialize the shadow top-level page table to contain the mappings shown in Figure 5 and then set the guest page table base register to point to it. The virtual memory map of an AArch32 process running under the HyperMAMBO-X64 DBT has four main components:

32-bit process pages The page table entries for the lowest 4 GB of the address space are copied directly from the page tables set up by the guest OS[2]. Copying only the page table entries referring to the lowest 4 GB of the address space is sufficient because the AArch32 process accesses memory using 32-bit pointers which restricts it to the lowest 4 GB of the 64-bit virtual address space. This portion of the address space is remapped every time the DBT switches to running a different user process so that it always contains the mappings for the process currently being translated.

[2] In practice, only the entries in the top-level page table need to be copied since the lower-level page tables can be used directly.

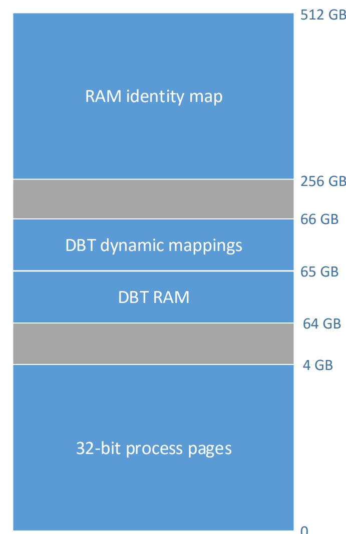

Figure 5: Virtual memory map of a process running under the HyperMAMBO-X64 DBT

DBT RAM The DBT reserved memory is mapped directly into the address space of the translated process. This memory area contains the DBT code and data, as well as the translated code fragments and their associated metadata.

Dynamic DBT mappings A portion of the address space is reserved for dynamic mapping of certain data structures used by the DBT. These typically consist of data structures that require memory protection features. One example of such is the return address stack [18] used for optimizing function returns in the translated code, which requires guard pages to catch stack overflows.

RAM identity map An identity map of the entire guest RAM is made available to the DBT for the purpose of performing page table walks in software. This is necessary to determine the access permissions for a particular memory address and, in particular, to determine whether a certain page has execute permission when translating code from it.

One of the key benefits of our binary translation model compared to a full system translator is that page tables are entirely managed by the guest OS, which avoids the need to perform expensive software TLB emulation. However this requires ensuring that the shadow page tables used by the DBT always match those set by the guest OS for the AArch32 process. In a virtual machine with only a single

virtual CPU, this is trivially handled by updating the shadow page table entries every time the hypervisor enters the DBT.

The situation is more complicated in a multi-processor virtual machine since the top-level page table of an AArch32 process may be modified by one processor while that process is running in the DBT on another processor. HyperMAMBO-X64 handles such cases by trapping guest execution of TLB invalidation instructions to the hypervisor, where it will update the shadow top-level page tables for any processes running under a DBT on another processor. This is safe since the ARM architecture requires a TLB flush to ensure that updated page table entries are picked up by all processors. We also take advantage of trapping guest TLB flush instructions for our code cache consistency algorithm which is described in Section 3.4.

3.4 Code cache consistency

A key aspect of a DBT is maintaining consistency between translated code fragments and the original instructions they were translated from: if the original instructions are modified and the instruction cache is flushed appropriately then the underlying architecture guarantees that the new code will be executed, and this must be reflected in the code cache of a DBT by invalidating all relevant translated fragments when such a modification occurs.

In an application-level translator such as MAMBO-X64, this problem is easily solved: there is only a single address space to deal with and we can keep track of any changes by intercepting system calls. In Linux for example, there are only 2 types of system calls which can affect translated code: those which modify virtual memory mappings (`mmap`, `mprotect`, `munmap`, etc.) and those which perform instruction cache invalidation on behalf of the application[3] to support self-modifying code and runtime code generation (`cacheflush`).

This approach is not viable in HyperMAMBO-X64 because it has no knowledge of the underlying guest OS and its system call interface. The guest OS is free to perform an instruction cache invalidation or page table modification affecting the translated process at any point. Instead, HyperMAMBO-X64 uses a three-tiered approach to ensure that translated code remains consistent with the instructions it is sourced from.

ASID-based address space management A guest OS may have multiple AArch32 processes running concurrently, each with its own address space. HyperMAMBO-X64 is able to distinguish the different address spaces by reading the ASID value from the system registers. The ASID is a 16-bit value set by the guest kernel to identify the current address space. It is used by the hardware to tag addresses in the TLB and avoid TLB flushes on context switches.

[3] Instruction cache flushing is a privileged operation in AArch32, thus requiring a system call to perform from user mode.

HyperMAMBO-X64 takes advantage of this ARM architecture feature by tagging every translated code fragment with the ASID it originated from, and uses this tag when looking up a code fragment to execute. This allows HyperMAMBO-X64 to support two or more user processes with different code at the same virtual address without requiring a full code cache flush when switching to a different address space.

TLB invalidation tracking HyperMAMBO-X64 also needs to keep track of changes within a particular address space: the guest kernel can modify the page table entries of an AArch32 process at any time, even if that process is concurrently executing on a different virtual CPU. HyperMAMBO-X64 exploits the fact that any such modification requires a TLB flush and traps the execution of any TLB flush instruction by the guest kernel to the hypervisor. The hypervisor can then invoke a callback in the DBT to invalidate any code fragments that were based on the pages affected by the TLB flush. Once the DBT invalidation is complete, the hypervisor will perform the TLB flush on behalf of the guest OS and then resume execution of the guest OS.

Switching from the guest OS to the DBT via the hypervisor and back for every TLB invalidation has significant overhead, especially considering that the majority of TLB invalidations are due to memory allocation and deallocation for data rather than code, so we implemented several optimizations to reduce this overhead. As it translates code, the DBT tracks the set of virtual memory addresses from which instructions are read for translation.These addresses are tracked at a page granularity, tagged with the ASID of the process they belongs to. When the translator reads instructions from a page for the first time, it performs a hypervisor call to register the virtual address and ASID of that page, which indicates to the hypervisor that the DBT has fragments which are based on that page.

Since the hypervisor only needs to notify the DBT about TLB invalidations which affect a previously registered virtual address and ASID combination, it can filter out TLB invalidations which do not affect the DBT by using a hash table lookup in the trap handler. Calling into the DBT to perform an invalidation can thus be avoided if the lookup finds that the TLB invalidation does not affect any virtual addresses registered by the DBT.

Code page write protection Even when the virtual memory mappings of an AArch32 process are not modified, the contents of the underlying page can be modified, invalidating any translated code derived from it. This can happen when code is modified by a JIT compiler or simply because the guest OS is reusing a page that previously contained code for another purpose. While the ARM architecture requires an instruction cache flush in such cases, simply trapping all instruction cache flushes to the hypervisor, as is done for TLB flushes, is not viable for several reasons:

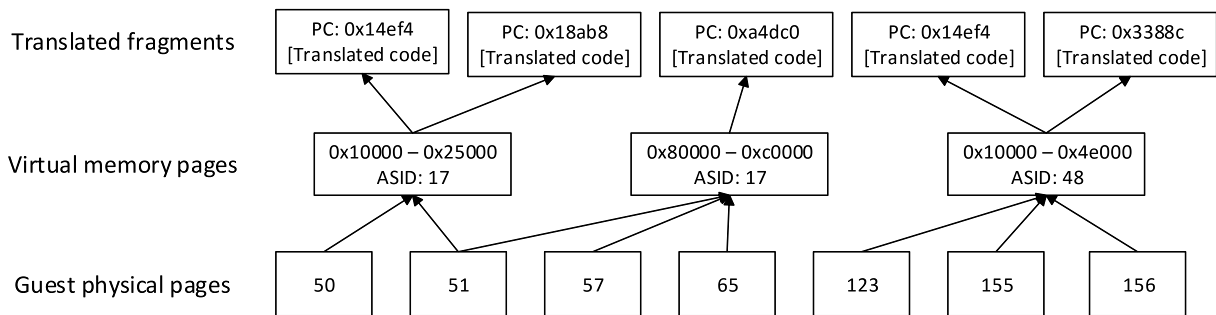

Figure 6: HyperMAMBO-X64 data structures for tracking code cache invalidation

- Unlike TLB flushes, ARMv8 does not provide a way for a hypervisor, which uses EL2 page tables, to perform an instruction cache flush on behalf of a guest kernel which uses EL1/EL0 page tables.
- There are many situations in which the guest OS needs to flush the entire instruction cache, which would require the DBT to flush all translated code for all address spaces.
- Translated fragments may be sourced from data as well as instructions, but the former may be modified without an instruction cache invalidation. One example of this is the branch table translation used by MAMBO-X64 [18]: an entry of the source branch table may be modified without an instruction cache flush, yet the translated code must take this modification into account the next time the branch table is executed.

HyperMAMBO-X64 instead makes use of the hypervisor-managed stage-2 page tables to detect code modifications: when the DBT registers a virtual address for TLB invalidation tracking, the hypervisor will also write-protect the underlying guest physical address for that page. If the guest attempts to write to a protected page, the hypervisor will notify the DBT so that it can invalidate any affected code fragments. Once the invalidation is complete, the hypervisor will remove the write-protection on the page and remove it from the set of watched pages since all translated fragments based on that page have been invalidated.

Figure 6 shows how these three mechanisms fit together. Each code fragment (top row) is translated from a sequence of source instructions which are located within a range of virtual addresses in a given address space (middle row). In turn, these virtual addresses are backed by a set of of physical pages (bottom row). If the data on one of these physical pages or the page mappings themselves are modified then all affected fragments are invalidated. Switching between different address spaces, such as during a context switch, does not cause any invalidations since virtual addresses are ASID-tagged.

3.5 Implementation

A prototype implementation of HyperMAMBO-X64 was built on top of the Linux Kernel Virtual Machine (KVM) [17] hypervisor. However the general concept is portable to other AArch64 hypervisors such as Xen [6] or Xvisor [28]. Another possibility for consumer devices such as smartphones, which do not need to run more than one OS, is to implement HyperMAMBO-X64 as part of a minimal hypervisor which only performs binary translation while allowing the guest OS full access to the underlying hardware.

One significant issue that we encountered while implementing HyperMAMBO-X64 is that there is no direct way to trap mode switches from an AArch64 guest kernel to AArch32 user mode in current ARMv8 processors. In our prototype, we worked around this issue by performing a small modification to the guest kernel: the `ERET` instruction responsible for perform an exception return into AArch32 mode was replaced with a `HVC` hypercall instruction. We anticipate that in an AArch64-only processor this instruction would generate an "Illegal Mode" exception when trying to switch to the non-existent AArch32 mode, which could be caught by the hypervisor.

4. Evaluation

In this section we evaluate the performance of HyperMAMBO-X64 and how it compares to a similar application-level translator. We use a set of microbenchmarks and the SPEC CPU2006 benchmark suite.

Because the ARMv8 processors used in these experiments are capable of running AArch32 code directly, all benchmarks are executed natively on the same processor and the results are used as a baseline for the experiments. All other results are normalized to this baseline, showing the relative performance of the DBT compared to native execution.

We also run the same benchmarks under MAMBO-X64, an application-level translator which also translates code from AArch32 to AArch64, which HyperMAMBO-X64 extends. Since HyperMAMBO-X64 and MAMBO-X64 share

the same DBT engine, they produce similar translated code. The differences appear at the boundary between the translated application and the operating system.

To ensure consistent results, the benchmarks executed in the three configurations (Native, MAMBO-X64 and HyperMAMBO-X64) all use the same statically linked AArch32 binaries.

Our test system is an AppliedMicro X-Gene X-C1 development kit with 8 X-Gene processor cores running at 2.4 GHz. Each core has a 32 kB L1 data cache, a 32 kB L1 instruction cache, a 256 kB L2 cache shared between each pair of cores and an 8 MB L3 cache. The machine comes with 16 GB of RAM and runs Debian Unstable with Linux kernel version 4.6.

4.1 Microbenchmarks

We started by running some microbenchmarks which stress particular aspects of the implementation of HyperMAMBO-X64. Four different benchmarks were run, and their results are shown in Table 1.

Integer This benchmark simply increments an integer in memory one billion times. It aims to measure the overhead HyperMAMBO-X64's handling of interrupts that would otherwise be transparent to MAMBO-X64 or a native application, such as timer interrupts. Because the guest OS needs to see the AArch32 register state of the translated process upon receiving the interrupt, HyperMAMBO-X64 needs to recover this state every time a virtual interrupt occurs, unlike MAMBO-X64 which only needs to do this when an asynchronous OS signal occurs. However, the results show no measurable difference in the three tested configuration. This is due to interrupts being such a relatively rare occurrence that any performance impact in their handling is negligible.

Syscall This benchmark measures the overhead of invoking a system call by calling the 'getppid' system call in a loop ten million times. This system call performs very little work and effectively measures just the overhead of switching into and out of the kernel. Both MAMBO-X64 and HyperMAMBO-X64 suffer in this respect because they need to perform internal bookkeeping operations before performing the system call. HyperMAMBO-X64 additionally suffers from the need to go through the hypervisor when switching into and out of the DBT.

Page fault This benchmark allocates 2 GB of virtual memory using `mmap` and then touches every 4 kB page by writing one byte into each. The `mmap` call will initially map every page to a copy-on-write zero page. Every write to such a page will trigger a page fault which the OS will handle by allocating a new writable page. As with interrupts, this process is transparent to native executables and MAMBO-X64. HyperMAMBO-X64 however must catch the fault and recover the AArch32 register state at the faulting instruction so that it can be presented to the guest OS for fault handling.

Benchmark	Native	MAMBO-X64	HyperMAMBO-X64
Integer	2.92	2.92	2.92
Syscall	1.00	8.39	10.63
Page fault	1.94	1.96	1.96
Signal	1.67	6.19	4.65

Table 1: Microbenchmark results in the three tested configurations. All results are in seconds.

However the results show that there is negligible difference in performance in the three tested configurations, due to the cost of the page fault in the OS dwarfing any handling by MAMBO-X64.

Signal This benchmark measures the overhead of signal handling by registering a signal handler for `SIGSEGV` and then dereferencing a null pointer one million times. Each dereference causes a page fault which the guest OS reflects back to the user process as a synchronous signal. The signal handler simply skips the offending instruction and allows the program to continue. Although HyperMAMBO-X64 and MAMBO-X64 both use the same algorithm for recovering an AArch32 register state, MAMBO-X64 also needs to emulate the Linux signal handling interface, which requires additional system calls to set the signal mask and leads to it having a higher overhead compared to HyperMAMBO-X64.

4.2 SPEC CPU2006

To evaluate the performance of HyperMAMBO-X64 with complex applications, we ran the SPEC CPU2006 [36] benchmark suite under HyperMAMBO-X64, MAMBO-X64 and natively. Figure 7 shows the results of these experiments. These show that, overall, both HyperMAMBO-X64 and MAMBO-X64 are able to deliver a performance level comparable to and sometimes even exceeding that of the processor's hardware support for AArch32 code. The geometric mean average of the results show that HyperMAMBO-X64 and MAMBO-X64 are 1.1 % and 1.0 % faster than native execution respectively.

Both systems are able to run many 32-bit benchmarks faster than if they were run natively on the processor. This is due to a combination of several factors:

- MAMBO-X64 takes advantage of the more flexible AArch64 instruction encodings to translate certain AArch32 instruction sequences into a single AArch64 instruction.
- Previous research in Dynamo [4] has shown that effective trace generation in a DBT can improve runtime performance compared to native execution.
- We have observed that on certain combinations of benchmarks and microarchitectures, such as the *libquantum* benchmark on X-Gene, the AArch32 code generated by GCC causes processor pipeline stalls which do not occur in the AArch64 translated code.

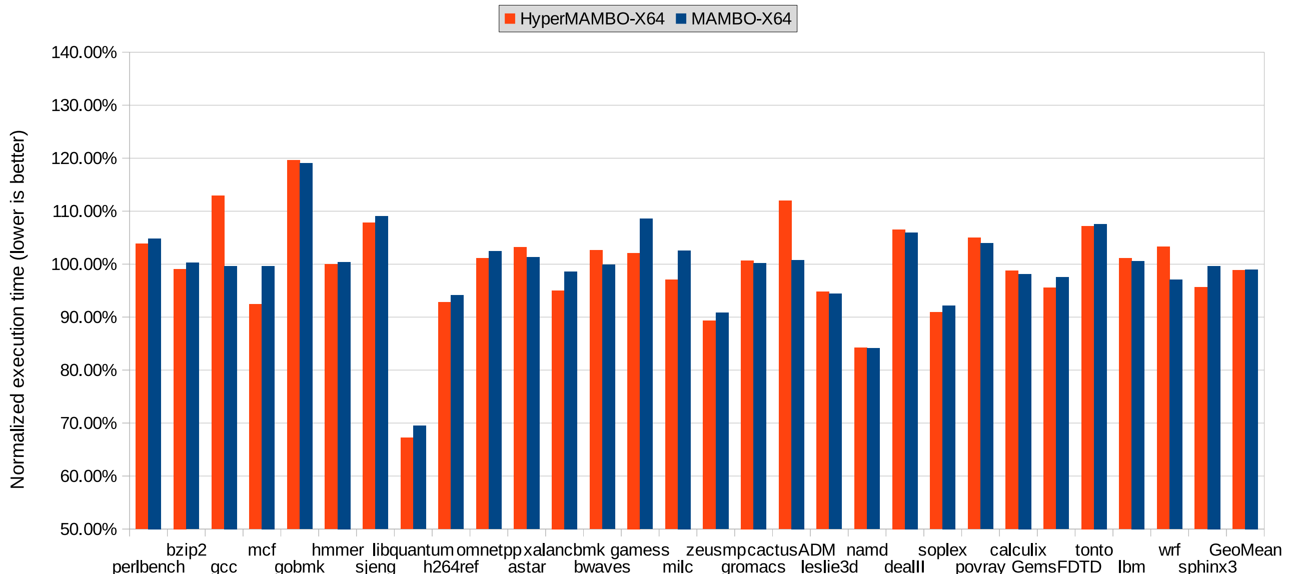

Figure 7: Performance of SPEC CPU2006 under HyperMAMBO-X64 and MAMBO-X64. Performance numbers are relative to the benchmark running natively in AArch32 mode.

Note that a few benchmarks have an overhead that is up to 13 % higher under HyperMAMBO-X64 than under MAMBO-X64. This particularly affects the *gcc* and *cactusADM* benchmarks, while also affecting the *wrf* benchmark to a lesser extent.

Analysis of these benchmarks showed that the performance loss was indirectly related to the way HyperMAMBO-X64 handles page faults. HyperMAMBO-X64 and MAMBO-X64 both use a variant of the Next Executing Tail [22] algorithm to generate traces, which are large single-entry multiple-exit fragments. Traces are built by finding hot basic blocks and recording an execution path through these blocks, which are then combined into a single fragment. Page faults interfere with the recording of an execution path and cause traces to terminate prematurely. This in turn results in a greater number of small traces, thus limiting their effectiveness.

5. Related work

MagiXen [13] is probably the closest work to ours, which also integrates a DBT with a hypervisor, in this case to translate a 32-bit x86 operating system on an Itanium system using the Xen hypervisor. Like HyperMAMBO-X64, MagiXen reuses the core of an existing application-level translator (IA-32 EL) as the DBT, however MagiXen differs in that it is closer to a full system-level translator. A limitation of MagiXen is that it only supports running paravirtualized guest operating systems, which means that the guest runs in user mode and does not make use of privileged instructions. Despite this, the performance of MagiXen still suffers compared to native execution due to the need to translate page tables in the hypervisor: although the guest OS is paravirtualized, its page tables are still in the x86 format. Additionally, x86 does not support tagged TLBs and ASIDs, which means that a full TLB flush is required on every context switch. This TLB flush must necessarily invalidate all of the translated code for the current process.

PinOS [11] is an extension of the Pin [26] dynamic instrumentation framework, which it adapts to instrument an entire operating system. Like HyperMAMBO-X64, it builds on top of existing hardware virtualization platforms (Xen for PinOS, KVM for HyperMAMBO-X64) to support transparent instrumentation of unmodified operating systems. While both face similar issues with regards to detecting modifications of code pages, HyperMAMBO-X64 exploits ARM architectural features to detect such situations while PinOS requires runtime checks for page table modifications. This is reflected in the overall performance of the systems: while HyperMAMBO-X64 is able to achieve near-native performance, applications under PinOS typically suffer from a slowdown on the order of 50x.

QEMU [7] is a DBT which supports a large number of architectures both as host ISAs and as guest ISAs. A key feature of QEMU is its ability to run both as a system-level translator and as an application-level translator. However it performs virtual address translation in software when running as a system-level translator, which impacts its performance.

Nvidia's Project Denver [9] and Transmeta's Crusoe [20] are two processors which use a DBT to translate code for a source architecture (ARM for Denver, x86 for Crusoe) into the processor's internal VLIW instruction set. While this

puts them in the category of system-level translators, they do not suffer from the overheads of page table translation since they include specialized hardware support.

Finally, there have been many instances of application-level translators used to assist an architecture transition. Examples include HP Aries [44] (PA-RISC to IA-64), IA-32 EL [5] (x86 to IA-64), FX!32 [14, 25] (x86 to Alpha) and Rosetta [2] (PowerPC to x86).

6. Discussion

The evaluation has not covered every stress point of a DBT, such as the handling dynamically generated code, memory usage or startup overhead. The following discussion indicates how these situations would be handled.

6.1 Dynamically generated code

A disadvantage of the use of page permissions to detect code modifications is their coarse granularity. This is not a problem for typical compiled applications since code pages are generally mapped with read-only permissions and do not contain any mutable data. However, the use of just-in-time (JIT) compilers, which involves frequent modifications of code pages, is becoming increasingly common to accelerate the execution of scripting languages such as Javascript. The performance of JIT compilers could suffer under the basic HyperMAMBO-X64 system when generating a high number of page faults due to code page write protection in the hypervisor.

One approach to reducing this overhead would be to adapt the parallel mapping technique developed by Hawkins et al. for DynamoRIO [24]. This involves creating two "views" of the RAM in the guest physical address space. The first view is used by the guest OS and most translated code. The second view is a "mirror" which is identical except that writes to code pages in this view are not trapped to the hypervisor.

HyperMAMBO-X64 can then identify memory write instructions which cause frequent hypervisor traps and replace them with instrumented writes. The instrumented write can use a fast hash table lookup to check if the target address points to a virtual page in the current process from which code has been translated and continue with a normal write if not. If the check passes then the DBT invalidates any code fragments derived from the instructions at the target address and performs the write in the mirror RAM to avoid a hypervisor trap.

6.2 Persistent code caching through virtualized non-volatile memory

A 'traditional' file system *copies* its contents into RAM pages; however it is increasingly feasible to implement large portions of the filestore in RAM. If this has been done it makes little sense to load a file by copying from one part of the RAM to another. The idea of virtual persistent memory has previously been used in Intel's Clear Containers [39] and involves mapping an entire virtual disk image directly as guest physical memory. If the guest OS supports it, such as through Linux's DAX subsystem [15], pages from the disk can be mapped directly into the address space of a guest process, thus entirely bypassing the disk cache in the guest OS.

HyperMAMBO-X64 can exploit this feature to create persistent code caches because it can associate translated code fragments directly with a page of the virtual disk. This key piece of information allows translated code to be written to a cache file and to persist across reboots of the virtual machine: once the pages are mapped into a process running under the DBT, the translated code can be 'loaded' immediately from the cache. This cache can even be shared across multiple virtual machines, for example if they share a read-only virtual disk containing the guest OS and applications.

7. Conclusions

We have proposed and evaluated HyperMAMBO-X64, a new type of Dynamic Binary Translator which is a hybrid of existing types of translators and preserves the best attributes of each. HyperMAMBO-X64 extends an existing hypervisor to allow an AArch64 guest operating system to run AArch32 user mode processes even when the underlying processor only supports AArch64. This is achieved by having the hypervisor trap attempts by the guest OS to switch to AArch32 user mode and running any AArch32 code under a DBT. The DBT returns control to the guest OS once an exception (syscall, page fault, interrupt) occurs by simulating an exception coming from AArch32 mode. This process is completely transparent to the guest OS: from its point of view, the user process was executing natively in AArch32 mode. Yet since the page tables are entirely controlled by the guest OS which runs natively, HyperMAMBO-X64 can achieve similar levels of performance as application-level translators.

A key challenge in the implementation HyperMAMBO-X64 is keeping the translated code generated by the DBT consistent with any changes to the source instructions. HyperMAMBO-X64 solves this challenge by exploiting several features of the ARMv8 architecture and virtualization extensions. Each translated code fragment is associated with a user-mode process in the virtual machine using the address space identifier (ASID) tags which are used by the TLB hardware. Modifications to the address space of a process are detected by trapping all TLB flush instructions to the hypervisor, which can then invalidate any translations affected by the changed virtual memory mappings. Finally, memory pages from which code has been translated are write-protected by the hypervisor to detect any modifications.

The evaluation using microbenchmarks and SPEC CPU2006 shows that HyperMAMBO-X64 introduces negligible performance overhead when compared with MAMBO-X64, a

similar application-level DBT for ARMv8, while reaping the transparency benefits of system-level translators.

In addition to its applicability to virtual machine migration to new, AArch64-only processors, HyperMAMBO-X64 can also be used to support specialized situations. One such example is supporting ARM "big.LITTLE" single-ISA heterogeneous systems where HyperMAMBO-X64 will allow a "LITTLE" core to eliminate hardware support for AArch32 and reduce its power usage, while still allowing an operating system to freely migrate AArch32 tasks between the clusters.

Acknowledgments

This work was supported by UK EPSRC grants DOME EP/J016330/1 and PAMELA EP/K008730/1. Luján is supported by a Royal Society University Research Fellowship.

References

[1] K. Adams and O. 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 2006*, pages 2–13. ACM, 2006. doi: 10.1145/1168857.1168860.

[2] Apple. Apple — Rosetta, 2006. URL `https://www.apple.com/rosetta/`. [Archived at `http://web.archive.org/web/20060113055505/http://www.apple.com/rosetta/`].

[3] ARM. big.LITTLE technology: The future of mobile, 2013. URL `https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf`. (Visited on 13/07/2016).

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, 2000. doi: 10.1145/349299.349303.

[5] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 191–204. ACM/IEEE Computer Society, 2003. doi: 10.1109/MICRO.2003.1253195.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003*, pages 164–177. ACM, 2003. doi: 10.1145/945445.945462.

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 41–46. USENIX, 2005. URL `http://www.usenix.org/events/usenix05/tech/freenix/bellard.html`.

[8] R. Bhardwaj, P. Reames, R. Greenspan, V. S. Nori, and E. Ucan. A Choices hypervisor on the ARM architecture. *Department of Computer Science, University of Illinois at Urbana-Champaign*, 2006. CS523 Course Project Report.

[9] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman. Denver: Nvidia's first 64-bit ARM processor. *IEEE Micro*, 35(2): 46–55, 2015. doi: 10.1109/MM.2015.12.

[10] D. Bruening and V. Kiriansky. Process-shared and persistent code caches. In *Proceedings of the 4th International Conference on Virtual Execution Environments, VEE 2008*, pages 61–70. ACM, 2008. doi: 10.1145/1346256.1346265.

[11] P. P. Bungale and C. Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007*, pages 137–147. ACM, 2007. doi: 10.1145/1254810.1254830.

[12] C. Chang, J. Wu, W. Hsu, P. Liu, and P. Yew. Efficient memory virtualization for cross-ISA system mode emulation. In *10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14*, pages 117–128. ACM, 2014. doi: 10.1145/2576195.2576201.

[13] M. Chapman, D. J. Magenheimer, and P. Ranganathan. Magixen: Combining binary translation and virtualization. Technical report, Technical Report HPL-2007-77, Hewlett-Packard Laboratories, 2007.

[14] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. FX! 32: A profile-directed binary translator. *IEEE Micro*, (2):56–64, 1998.

[15] J. Corbet. Supporting filesystems in persistent memory, 2014. URL `https://lwn.net/Articles/610174/`.

[16] K. V. Craeynest, A. Jaleel, L. Eeckhout, P. Narváez, and J. S. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *39th International Symposium on Computer Architecture (ISCA 2012)*, pages 213–224. IEEE Computer Society, 2012. doi: 10.1109/ISCA.2012.6237019.

[17] C. Dall and J. Nieh. KVM/ARM: the design and implementation of the Linux ARM hypervisor. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14*, pages 333–348. ACM, 2014. doi: 10.1145/2541940.2541946.

[18] A. d'Antras, C. Gorgovan, J. D. Garside, and M. Luján. Optimizing indirect branches in dynamic binary translators. *ACM Transactions on Architecture and Code Optimization*, 13(1): 7, 2016. doi: 10.1145/2866573.

[19] A. d'Antras, C. Gorgovan, J. D. Garside, and M. Luján. Low overhead dynamic binary translation on ARM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*. ACM, 2017.

[20] J. C. Dehnert, B. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003)*, pages 15–24. IEEE Computer Society, 2003. doi: 10.1109/CGO.2003.1191529.

[21] J.-H. Ding, C.-J. Lin, P.-H. Chang, C.-H. Tsang, W.-C. Hsu, and Y.-C. Chung. ARMvisor: System virtualization for ARM.

In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 93–107, 2012.

[22] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211. ACM Press, 2000. doi: 10.1145/356989.357008.

[23] R. Grisenthwaite. ARMv8 Technology Preview, 2011.

[24] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao. Optimizing binary translation of dynamically generated code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015*, pages 68–78. IEEE Computer Society, 2015. doi: 10.1109/CGO.2015.7054188.

[25] R. J. Hookway and M. A. Herdeg. DIGITAL fx!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1), 1997. URL `http://www.hpl.hp.com/hpjournal/dtj/vol9num1/vol9num1art1.pdf`.

[26] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 190–200. ACM, 2005. doi: 10.1145/1065010.1065034.

[27] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 4th Conference on Computing Frontiers*, pages 143–152. ACM, 2007. doi: 10.1145/1242531.1242554.

[28] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi. Embedded hypervisor xvisor: A comparative analysis. In *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015*, pages 682–691. IEEE Computer Society, 2015. doi: 10.1109/PDP.2015.108.

[29] N. Penneman, D. Kudinskas, A. Rawsthorne, B. D. Sutter, and K. D. Bosschere. Formal virtualization requirements for the ARM architecture. *Journal of Systems Architecture - Embedded Systems Design*, 59(3):144–154, 2013. doi: 10.1016/j.sysarc.2013.02.003.

[30] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974. doi: 10.1145/361011.361073.

[31] V. J. Reddi, D. Connors, R. Cohn, and M. D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *Fifth International Symposium on Code Generation and Optimization (CGO 2007)*, pages 74–88. IEEE Computer Society, 2007. doi: 10.1109/CGO.2007.29.

[32] Y. Sato, Y. Inoguchi, and T. Nakamura. On-the-fly detection of precise loop nests across procedures on a dynamic binary translation system. In *Proceedings of the 8th Conference on Computing Frontiers*, page 25. ACM, 2011. doi: 10.1145/2016604.2016634.

[33] D. Seal. *ARM Architecture Reference Manual*. Pearson Education, 2001.

[34] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 17–30. USENIX, 2005. URL `http://www.usenix.org/events/usenix05/tech/general/seward.html`.

[35] A. Smirnov, M. Zhidko, Y. Pan, P. Tsao, K. Liu, and T. Chiueh. Evaluation of a server-grade software-only ARM hypervisor. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 855–862. IEEE, 2013. doi: 10.1109/CLOUD.2013.71.

[36] Standard Performance Evaluation Corporation. SPEC CPU2006. `http://www.spec.org/cpu2006/`.

[37] X. Tong, T. Koju, M. Kawahito, and A. Moshovos. Optimizing memory translation emulation in full system emulators. *ACM Transactions on Architecture and Code Optimization*, 11(4):60:1–60:24, 2014. doi: 10.1145/2686034.

[38] Transitive. Transitive, 2008. URL `http://www.transitive.com`. [Archived at `https://web.archive.org/web/20080914184751/http://www.transitive.com`].

[39] A. van de Ven. An introduction to clear containers, 2015. URL `https://lwn.net/Articles/644675/`.

[40] C. Wang, S. Hu, H. Kim, S. R. Nair, M. B. Jr., Z. Ying, and Y. Wu. StarDBT: An efficient multi-platform dynamic binary translation system. In *Advances in Computer Systems Architecture, 12th Asia-Pacific Conference, ACSAC 2007, Proceedings*, volume 4697 of *Lecture Notes in Computer Science*, pages 4–15. Springer, 2007. doi: 10.1007/978-3-540-74309-5_3.

[41] W. Wang, P. Yew, A. Zhai, and S. McCamant. A general persistent code caching framework for dynamic binary translation (DBT). In *2016 USENIX Annual Technical Conference, USENIX ATC 2016*, pages 591–603. USENIX Association, 2016. URL `https://www.usenix.org/conference/atc16/technical-sessions/presentation/wang`.

[42] J. Watson. Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1, 2008.

[43] Q. Zhao, D. Koh, S. Raza, D. Bruening, W. Wong, and S. P. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th International Conference on Virtual Execution Environments, VEE 2011*, pages 27–38. ACM, 2011. doi: 10.1145/1952682.1952688.

[44] C. Zheng and C. L. Thompson. PA-RISC to IA-64: transparent execution, no recompilation. *IEEE Computer*, 33(3):47–52, 2000. doi: 10.1109/2.825695.