Low Overhead Dynamic Binary Translation for ARM

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering.

2016

Bernard Amanieu d'Antras

School of Computer Science

Contents

This thesis contains 35122 words.

List of Figures

List of Tables

Abstract

LOW OVERHEAD DYNAMIC BINARY TRANSLATION FOR ARM

Bernard Amanieu d'Antras

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2016

Driven by Moore's Law, many computer architectures — ARM, x86, MIPS, PowerPC, SPARC — have evolved from 32-bit to 64-bit. To support existing applications, these have all kept support for a 32-bit compatibility mode. However, this comes at a cost in hardware complexity, power consumption and development time.

Dynamic binary translation — recompiling binaries into the new instruction set at runtime — can be used instead of specific hardware for this purpose. While this approach has previously been used to assist architecture transition, these translators have all traded-off performance and *transparency*, a measure of how accurately they emulate the 32-bit environment.

This thesis addresses ARM's transition from AArch32 to AArch64 through MAMBO-X64, a dynamic binary translator developed to support this transition. A range of novel optimizations were devised to improve translation performance while maintaining strict transparency. This follows a common theme of exploiting existing hardware features such as hardware return prediction, virtual memory and virtualization extensions to offset translation overheads. HyperMAMBO-X64 — a variant of MAMBO-X64 integrated in a hypervisor — was also developed to support system-level translation while remaining transparent to guest operating systems.

Results demonstrate that the cost of binary translation is reduced, delivering performance competitive with the manufacturer's hardware. Performance in several benchmarks even *exceeds* that from the integrated compatibility mode. Thus MAMBO-X64 not only provides a means for architectural upgrade, but also an *alternative* to the expense of the legacy support currently employed.

Declaration

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

Copyright

Acknowledgments

I am immensely grateful towards Prof. Mikel Luján, my supervisor, for his extensive support and guidance during my years as a PhD student. It was Mikel who first guided me towards binary translation as a research topic, and in hindsight it is obvious that he could not have made a better decision. I am similarly thankful to Dr. Jim Garside, my co-supervisor, who has always kept his door open for impromptu discussions, both technical and mundane, as well as his uncanny (but tremendously useful!) ability to always find something to improve in any paper that crosses his path.

Both of my supervisors have provided me with excellent and detailed feedback throughout the process of writing my thesis, for which I thank them, as well as Jonathan Heathcote and Dr. Andy Nisbet for providing helpful reviews of my thesis and papers.

I would like to extend special thanks to Cosmin Gorgovan and Guillermo Callaghan for the lively discussions on the finer points of dynamic binary translation. Though we may not have always agreed, these discussions were always mutually beneficial to our respective projects. And, of course, I thank the many members of the Advanced Processor Technologies group for filling the past few years of my life with fun, which more than compensated for the evanescent Manchester sunshine. I am also grateful to the School of Computer Science for the funding that it has provided for my research.

After nearly fifteen years, I extend a grateful thought to Nicolas Ténart, my secondary school mathematics teacher, who first taught me the joys of programming and started me down the path to this thesis.

Last but not least, I would like to thank my family for their patience, support and carefully orchestrated nodding as they endured a lifetime's quota of my rants on race conditions and obscure ARM instructions.

Chapter 1

Introduction

Computer architectures have always evolved following hardware and software technology trends over time, driven by Moore's Law [Moo65]. This evolution generally takes the form of adding new instructions to accelerate certain algorithms or implementing architectural features such as virtual memory. However, much like biological evolution, this process results in many features which "seemed like a good idea at the time" but turned out to be poorly adapted to a later software ecosystem.

For example, in early microprocessors (c. 1970) it was common to support Binary Coded Decimal (BCD) operations, something which is rarely wanted directly now. Such operations could be abandoned, but this renders old code potentially inoperable and is generally unacceptable. Thus they must be supported in some way.

Two major 'mainstream' processor architectures — Intel x86 and ARM — illustrate two contrasting approaches to this: Intel maintains full backwards compatibility with all instructions since the 8086 by emulating legacy instructions using low-level microcode programmes embedded in the processor. ARM has, instead, opted to support legacy instructions directly in hardware, while

simultaneously deprecating and removing old features (such as 26-bit addressing, Jazelle and ThumbEE) from newer versions of the ARM architecture. While removing features was acceptable when ARM processors were primarily used in embedded systems with tightly bound hardware and software, this is no longer the case now that ARM is a widespread, general-purpose architecture.

These considerations extend to architecture transitions: current computer architectures — ARM, MIPS, PowerPC, SPARC, x86 — have all evolved from a 32-bit architecture to a 64-bit one. Again, x86 and ARM take two contrasting approaches to this: on one hand x86-64 provides a mostly straightforward extension of the 32-bit instruction set to 64 bits, carrying over most of the legacy 8086 instructions to the new execution mode. On the other hand ARM exploited this opportunity to discard much of the 'baggage' it had accumulated over seven versions and many variants of the architecture, thus producing a 'clean' instruction set which, in turn, potentially allows for simpler and more efficient processor designs.

1.1 Binary translation

The desire to simplify and clean up the hardware architecture and the requirement to support software making use of legacy features can be bridged through the use of *binary translation*.

Binary translation is a technology which allows a program to be translated and modified transparently at the machine code level. It has numerous applications, such as dynamic instrumentation [MCGP07, SN05], program analysis [SIN11, ZKR+11], virtualization [AA06, Wat08] and instruction set translation [Bel05, DGB+03]. 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.

Binary translation has previously been used successfully to assist architecture transitions: the best known example is Rosetta [App06], which was used by Apple to transit their platform from PowerPC to x86, based on technology originally developed at the University of Manchester [Tra08]. IA-32 EL [BDE+03] and HP Aries [ZT00] both supported the transition to the IA-64 architecture from x86 and PA-RISC respectively. FX!32 [HH97, CHH+98] was similarly used to help migrate x86 applications to the Alpha architecture. Binary translation has also been used to allow execution of code from existing instruction sets on a Very Large Instruction Word (VLIW) architecture, such as Nvidia Denver (ARM on VLIW) [BBTV15] and Transmeta Crusoe (x86 on VLIW) [DGB+03].

While translation may be attempted statically (i.e. off-line by generating a new, translated binary) there has been a trend back to self-modifying code — particularly the use of Just-In-Time (JIT) compilers — which renders this process inefficacious. Instead, *dynamic binary translation* — at runtime — must be used for complete compatibility.

Binary translation typically suffers from some performance overhead compared to recompiling an application from its source code. The sources of overhead can be broadly categorized as follows:

Architectural mismatch While, fundamentally, the binary translation process is quite straightforward — an `ADD` instruction in the source architecture can be translated into an equivalent `ADD` instruction in the host architecture — complications can arise when the exact semantics of these instructions differ, such as the exact set of condition flags modified by such an `ADD` instruction (e.g. carry flag, overflow flag). While a source

compiler only needs to translate the semantics of the source code and is thus free to discard most of these flags, e.g. only using one of them for a conditional branch, a binary translator works at a lower level and must emulate all aspects of the host architecture accurately. This type of overhead generally comes in the form of additional instructions required to simulate all the effects of the source instruction, such as calculating the correct values of all the condition flags.

Environment overhead In addition to the above, generating code to work within the constraints of a binary translation environment brings its own challenges. For example, processor registers from the source architecture need to be mapped to those of the host architecture and, if the latter resource is insufficient, source register values should be spilled to memory. Additionally, any operations involving the program counter, such as branches, require special handling since the translated code may be located at a different address from the source instructions.

Translation overhead Finally, for dynamic translators, the process of translation itself occupies a portion of the execution time. Unlike the previous two sources of overhead, which are evenly distributed over the runtime of an application, translation overhead mainly occurs during application startup when no code has been translated yet. This overhead becomes insignificant for long-running applications as it is amortized over the entire runtime of the application, providing the translated code can be retained.

1.2 Contributions

There has been incremental development of binary translation over the past few decades, however the overheads outlined above still imply that the translated code runs slower than it could with hardware support.

In particular, while existing research has developed optimizations to deal with the overheads of binary translation, many of these trade off *transparency* in favor of performance. In other words, such optimizations make assumptions about the behavior of a program (e.g. "the program never accesses invalid memory addresses" or "condition flags do not need to be preserved across function calls"), and cause the translated program to behave incorrectly if these assumptions are violated.

This thesis enables the elimination of hardware support for a legacy instruction set in new processors by describing how to create dynamic binary translators which rival the performance of direct hardware support. This is achieved through a series of novel optimizations which address various sources of performance overhead involved in binary translation, all while maintaining the transparency of the translation.

The focus of this thesis is on the ARM's transition to a 64-bit architecture. The ARMv8 architecture introduced AArch64, a 64-bit execution mode with a new instruction set, in addition to the existing AArch32 32-bit instruction set. Due to the need to support the large number of existing AArch32 applications, current hardware implementations of ARMv8 processors support *both* AArch32 and AArch64. However, this situation is problematic since such support comes at a cost in hardware complexity, power usage and development time.

MAMBO-X64, a dynamic binary translator which executes 32-bit ARM binaries using only the AArch64 instruction set, was developed as a solution to this problem. It opens a path for future processors to drop hardware support

for the legacy 32-bit instruction set while retaining the ability to run AArch32 applications at realistic speeds. MAMBO-X64 is a mature piece of technology which is able to run complex Linux and Android applications and has already been licensed to at least one company.

The performance of MAMBO-X64 is competitive with that of the hardware support in current ARMv8 processors: 32-bit builds of many benchmarks — from suites such as SPEC CPU2006 and PARSEC — run faster under MAMBO-X64 than natively on the processor. This is attributable to a number of new techniques which have been developed to address the worst remaining inefficiencies of pre-existing translators.

It was reported some time ago that **indirect branch handling** was the biggest source of performance overhead in a binary translator [KS03, HWH+07], a problem which still had remained largely unsolved. An indirect branch is a branch instruction with a target which is only known at execution time and which can vary from one execution to the next, a common example being a subroutine return. Unlike direct branches, which have a known target at translation time, an indirect branch requires translating a source program counter address to a translated program counter address every time the branch is executed. This translation can impose a serious runtime penalty if it is not handled efficiently.

Chapter 3 describes three novel techniques for translating such branches. The first, *hardware-assisted function returns*, uses a software return address stack to predict the targets of function returns, making use of several novel optimizations while also exploiting hardware return address prediction. The second, *branch table inference*, is an algorithm for detecting and translating branch tables into equivalent structures for the host architecture. The remaining indirect branches are handled using a *fast atomic hash table*. This translates

indirect branches using a single shared hash table which avoids expensive synchronization in performance-critical lookup code. This chapter is based on the paper *Optimizing Indirect Branches in Dynamic Binary Translators* [dGGL16] which was published in *Transactions on Architecture and Code Optimization*.

Chapter 4 describes the other new optimization principles used by MAMBO-X64 to achieve high performance without sacrificing accuracy. The most significant one, after indirect branch optimizations, is *ReTrace*, the trace generation algorithm used by MAMBO-X64 which improves the layout of translated code by further exploiting hardware return address prediction. MAMBO-X64 also supports a wide range of optimizations, including an efficient system for mapping AArch32 floating-point registers to AArch64 registers dynamically and a speculative optimization to improve the performance of certain ARM addressing modes.

Supporting legacy code through binary translation requires more than just translating one instruction set to another: software is designed to work within an **environment**, and the software-visible interfaces of this environment must be translated as well. In the case of a user-level binary translator like MAMBO-X64, this consists primarily of OS interfaces such as *system calls* and *signals*. While system call translation is fairly straightforward, signals pose particular challenges since some can occur at arbitrary points in the execution of a program. Section 4.3 describes how MAMBO-X64 supports accurate delivery of synchronous and asynchronous OS signals without sacrificing performance.

Dynamic binary translation generally comes in one of two forms: application-level translators, which translate a single user mode process on top of a native operating system, and system-level translators which translate an entire operating system and all its processes. Application-level translators can have good

performance but are not totally transparent; system-level translators may be 100 % compatible but performance typically suffers.

Chapter 5 presents HyperMAMBO-X64, which 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 the caching of translations across multiple address spaces while ensuring that translated code remains consistent with the source instructions it is based on. These attributes are achieved without sacrificing either performance or accuracy.

Together this set of contributions has accelerated MAMBO-X64 such that it delivers execution performance rivaling that of a hardware implementation. This includes the translation time, which means that binary translation is not only feasible but also becomes an attractive option for future backward compatibility and will allow silicon companies such as ARM to reduce their development and verification overheads in future silicon implementations.

Chapter 2

Dynamic Binary Translation

A static binary translator translates the entirety of the program object code ahead of time. This is convenient because it may take time for optimization without impacting runtime. However such translators are not always practical in the general case due to the *code-discovery problem* [HM80]: since it is not always possible to determine which memory locations contain instructions as opposed to inline or pre-loaded data, all addresses need to be treated as potential branch targets to ensure full transparency. In modern systems this problem is worse in that not all the code that will be executed is present ahead of time, such as when a program imports shared libraries or generates new instructions using a Just-In-Time compiler. A Dynamic Binary Translator (DBT) translates code only as it is about to be executed, which avoids these issues but comes at a cost in overall runtime because code discovery and translation time is included.

A significant disadvantage of dynamic binary translators is that, while they are able to achieve good performance when executing translated code, they suffer from poor startup times because of the need to translate new code when an application begins. This has led to new types of DBTs which integrate

some aspects of static translation to accelerate application startup times. This comes either in the form of ahead-of-time translation which generates translated code by pre-processing an application binary, or in the form of persistent code caches [BK08, RCCS07] which retain translated code across multiple invocations of an application. Both of these methods provide a pre-existing base of translated code which is available immediately on application startup, while dynamic translation is still available to handle any remaining untranslated code.

2.1 Code caches

Translating a block of code is more efficient than translating single instructions in many ways, hence DBTs usually translate sequences of instructions as blocks, called *code fragments*. Fragments can have many forms, depending on the design of the DBT, the most common of which is the *basic block*. A basic block corresponds to a linear sequence of source instructions with a single entry point and a single exit point. While this approach has the advantage of simplicity, it can suffer from poor performance due to the need for branches between basic blocks. Many DBTs therefore also use of some form of *superblocks* which still have a single entry point but can have multiple exits. Larger fragments also allow a DBT more opportunities for optimizations such as dead code elimination and constant propagation.

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.

A naïve DBT would schedule the execution of each fragment as it is needed by jumping to the start of the fragment and having the fragment return control to the DBT once it has finished running. A fragment typically ends when the translation has encountered a branch instruction, at which point the DBT must select the next fragment to execute. This approach is impractical because of the high overhead of context switching between fragments and the DBT, and the frequent changing of fragments in typical programs[1].

Instead, a branch in one fragment can be *linked* to a different fragment by having the translated branch transfer control to the target fragment directly. Because source and translated fragments are not, necessarily, identical in size, the address from the source cannot be used directly: some address translation is necessary.

For *direct branches*, where the branch target is a constant encoded in the instruction itself, this translation can be performed statically at translation time, resulting in a single branch instruction in the translated code that points to the fragment for the target address. If a fragment has not yet been translated for the branch target, the translated branch can point to an *exit stub*, which returns control to the DBT while passing the current program counter value so that the DBT knows what to execute next. When the target fragment is translated, any branch instructions in other fragments that pointed to the exit stub are patched and redirected to point to the newly translated fragment.

Indirect branches need to be handled differently because their target is only known at execution time and can vary from one execution to the next. This requires its own dynamic translation and imposes a serious runtime penalty. Translation is typically implemented by using a hashed translation table to find the translated fragment for a given target address at runtime. While this

[1] A typical basic block fragment has fewer than a dozen instructions.

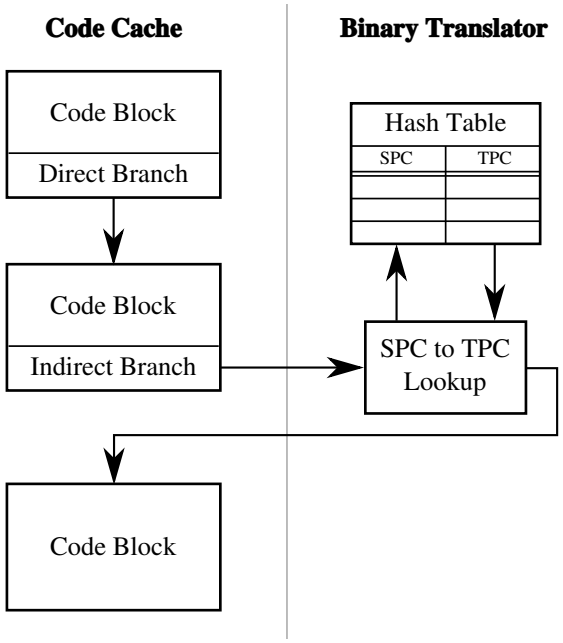


Figure 2.1: Direct and indirect branch handling in a DBT.

approach solves the translation problem, it is still many times slower than a native indirect branch, which only consists of a single instruction. It also interacts poorly with hardware branch prediction mechanisms that are optimized for native code, often resulting in unnecessary branch mispredictions.

Figure 2.1 shows how fragment in a code cache are linked through direct and indirect branches. While direct branches can be linked directly, every time an indirect branch is executed in a DBT, a Source Program Counter (SPC) value must be mapped to a Translated Program Counter (TPC) value, which is then branched to. This figure illustrates how the translation of an indirect branch can be much more costly than that of a direct branch.

2.2 Multi-threading

Multi-threaded applications pose additional challenges to a DBT. In particular, threads may share the source object code and different threads could 'collide' in deciding to perform translations. These concerns are particularly relevant for the design of the code cache, for which there exist two major models: *thread-private* code caches and *thread-shared* code caches.

Thread-private code caches This model is the simplest to implement since it involves each thread having a separate code cache. It also enables several optimizations that exploit the fact that a code cache is only used by a single thread, such as self-modifying code or embedding pointers to thread-local data directly in the translated code. The main disadvantage of thread-private code caches is that multi-threaded applications suffer from increased memory usage and high overhead on thread creation to fill the code cache, particularly for server applications which may create hundreds of threads. Additionally, the need for synchronization is not completely eliminated since it is still necessary to handle cases where one thread needs to invalidate a fragment in the code cache of another thread.

Thread-shared code caches While this model is more complicated to implement, researchers have demonstrated [BKGB06, HLC09] that it scales significantly better than thread-private code caches on multi-threaded applications. In this model, all threads share the same code cache, which means that translated code cannot 'hard-code' pointers to thread-local data and must use some form of indirection instead. Executing code in such a code cache does not require any synchronization and code translation can be performed concurrently in multiple threads. Synchronization is only required when adding or remov-

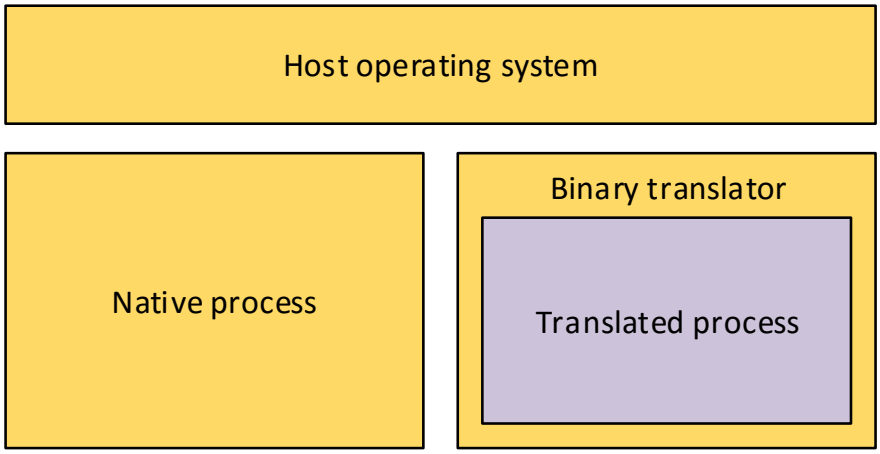
ing a fragment from the code cache, which is a rare operation compared to the execution of translated code.

A key issue with thread-shared code caches is the handling of fragment deletion: a fragment cannot be deleted from the code cache immediately since there may be other threads concurrently executing that fragment. Instead, the fragment must first be *unlinked*, which removes all branch instructions pointing to the fragment and ensures that it cannot be re-entered. The fragment can then be fully deleted once all other threads are known to have exited the code cache at least once since the fragment was unlinked, as this indicates that other threads can no longer be executing the deleted fragment.

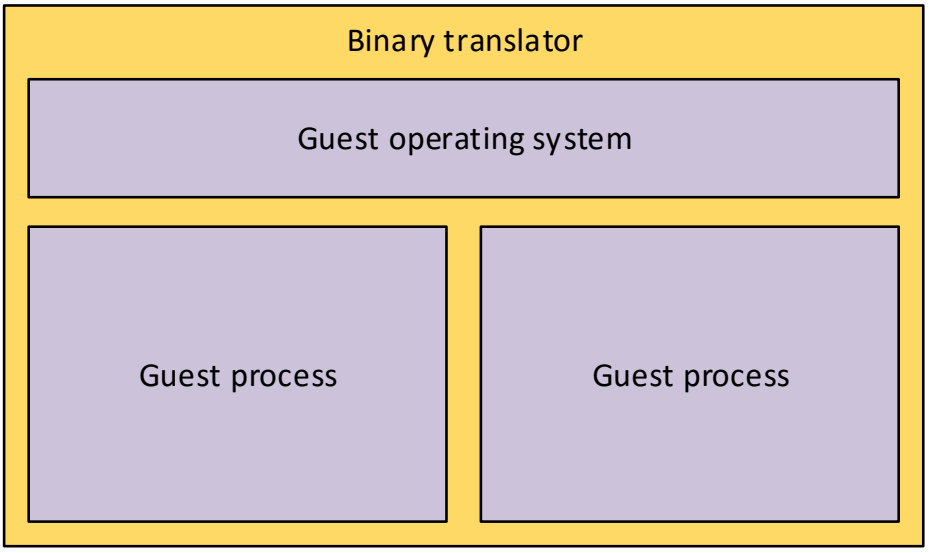
2.3 Environment

DBTs can generally be split into two categories, shown in Figure 2.2, 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 [Bel05], Aries [ZT00], IA-32 EL [BDE+03], FX!32 [HH97, CHH+98], Rosetta [App06] and StarDBT [WHK+07].



(a) Application-level translator



(b) System-level translator

Figure 2.2: Overview of application-level and system-level translators.

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 [CMR07], Transitive QuickTransit [Tra08], QEMU [Bel05], Transmeta's Code Morphing Software (CMS) [DGB+03] and Nvidia's Project Denver [BBTV15].

While the first part of this thesis focuses on application-level translators, the topic of system-level translators will be revisited in Chapter 5.

2.4 Transparency

Transparency is a measure of how accurately a DBT emulates a target environment. As a general rule, the more transparent a DBT is, the more details of the target environment it emulates which, in turn, allows a wider range of applications to function under the DBT successfully. Conversely, increased transparency also comes at a cost in performance, since additional time needs to be spent emulating these details.

A perfectly transparent DBT is functionally indistinguishable from the target environment in every way from the point of view of the translated application, down to simulating details such as instruction timings. Such a level of

transparency comes at a large performance cost, typically at least an order of magnitude slower than the simpler behavioral transparency, which is an unacceptably high cost in most situations.

A lesser form of transparency which is more commonly used by DBTs is *behavioral transparency*, which involves only supporting features defined by the ABI of the target environment. A DBT can exploit this by assuming that applications never perform any operations which the ABI considers to have undefined behavior. Examples of undefined behaviors in ABIs include using undefined instruction encodings, writing data below the stack pointer and not flushing the instruction cache after modifying instructions.

Despite this relative freedom, transparency requirements still have subtle implications for the code generated by a DBT, for example:

- An application's memory, including its code, should not be modified by a DBT. This is necessary to ensure that applications which inspect their own code memory see the original instructions, hence the use of a code cache by DBTs instead of modifying application memory. This restriction is further emphasized by the need to support applications that perform self-modification of their own code.

- Faulting memory accesses must be precisely emulated since they can be caught by an application fault handler which has the ability to inspect the full processor register state at the fault point. Some applications make use of this information to handle faults themselves, after which they resume execution with a modified register state. Supporting this means that a DBT must be able to recover the original values of all registers every time a potentially faulting instruction is executed.

- Even if memory accesses do not cause a fault, they may have side effects if their addresses refer to memory-mapped I/O regions. This means that a DBT cannot consider loads and stores to be free of side effects, which limits its ability to reorder, merge or elide such instructions.

2.5 MAMBO-X64

ARM [Sea01] is a general purpose architecture 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 [Gri11] 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, while also increasing the size of the floating-point/SIMD register bank.

One of the key factors driving the design of AArch64 is the desire to return to the RISC philosophy by cleaning up all the "clutter" that has accumulated in the 32-bit ARM architecture through its various revisions, which have led, among other things, to a convoluted and variable-width instruction encoding. The new instruction set is therefore a better fit for ARM's overall strategy by enabling smaller and lower-power core implementations.

While AArch64 has many benefits, there is a large ecosystem of existing 32-bit applications which need to be able to run on ARMv8 systems. 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:

- The AArch32 execution mode supports two instruction sets: a fixed-width 32-bit instruction set (ARM) and a variable-length 16/32-bit instruction set (Thumb-2). This increases the complexity and power usage

of the instruction decode unit compared to supporting just a single, fixed-width instruction set. AArch64 instructions are always 32 bits long and of regular format, which allows a simpler decoder design. On typical ARM processors, the instruction decode unit alone can account for over 10 % of the overall CPU power consumption [NVI13].

- AArch32 contains legacy instructions that have not been carried forward to AArch64. Many of these instructions are intended for specialized DSP workloads and rarely appear in compiler-generated code. Supporting these instructions requires additional complexity in the ALUs.

- Having to support two instruction sets in hardware can more than double the cost of hardware verification due to the possible interactions between instruction sets. This, in turn, increases the development time of a processor; it also increases the chance of a defect making its way into released hardware, fixing which may require the creation of a new processor revision.

MAMBO-X64 is a DBT developed by the author at the University of Manchester which translates AArch32 Linux programs into AArch64 code. It is implemented as a process-level virtual machine: a separate binary translator instance is started for each 32-bit process, while the operating system kernel and 64-bit processes run natively on the processor. The objective is to support the running of legacy AArch32 code without the need for specific hardware support, preferably at speeds competitive with hardware execution. No contemporary DBT which achieves a similar goal has been described.

When tested on existing ARMv8 systems which support both AArch32 and AArch64 in hardware, a 32-bit build of SPEC CPU2006 [Cor] ran on average 1 % *faster* under MAMBO-X64 compared to running the same 32-bit binary na-

tively on the processor. Particular benchmarks were also measured to run up to 38 % faster under MAMBO-X64 than natively, although a few other benchmarks suffer from a performance degradation of up to 19 %.

MAMBO-X64 is structured as three components, shown in Figure 2.3:

Binary translator The binary translator is an operating system-independent module which performs the translation of AArch32 instructions into AArch64 code.

System emulator The system emulator handles all interactions with the operating system, such as system calls and signals, and translates them between the 32-bit and 64-bit Linux ABIs.

Support library The support library provides OS-specific utilities such as memory management and synchronization primitives to the binary translator and system emulator.

This arrangement isolates OS-specific code from the binary translator, which makes it easier to port MAMBO-X64 to other operating systems.

2.5.1 Binary translator

The binary translator component has the same basic structure as a typical DBT: it works by scanning sequences of AArch32 instructions on demand and converting them into AArch64 code fragments, stored in a thread-shared code cache. Each fragment is either a single-entry, single-exit *basic block* or a single-entry, multiple-exit *trace* formed by combining multiple basic blocks.

A key focus in the design of MAMBO-X64 was efficient handling of indirect branches in translated code. For this purpose, three new techniques for translating indirect branches, each applying to a different class of indirect branch, have been developed. These techniques are discussed in detail in Chapter 3.

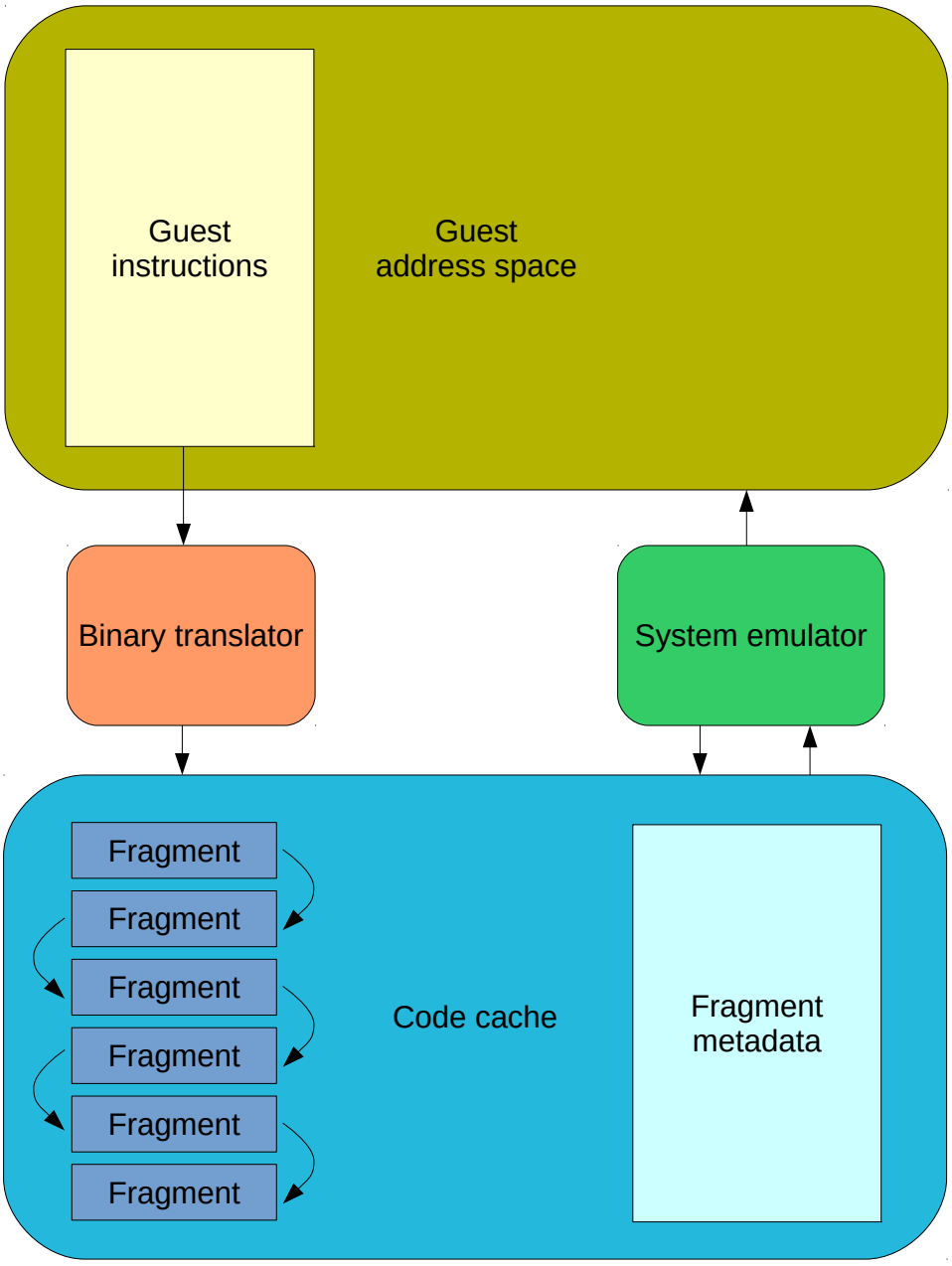


Figure 2.3: Overview of the different components in MAMBO-X64. The binary translator translates sequences of instructions from the guest address space into fragments in the code cache. The system emulator handles system interactions such as system calls and signals, and manages the guest address space. The code cache also contains various metadata associated with the translated fragments.

MAMBO-X64 also leverages a wide range of optimizations to improve the performance of the translated code. Specific techniques developed during the author's research are described in detail in Chapter 4. These optimizations include:

- An efficient scheme for mapping AArch32 floating-point registers to AArch64 registers dynamically.
- A method for efficiently translating AArch32 load/store addressing modes into AArch64 by speculatively assuming that address calculations do not overflow.
- A novel trace compilation algorithm that leverages hardware return address prediction to improve performance.

MAMBO-X64 is able to precisely emulate the full AArch32 instruction set, which includes both the traditional ARM instruction set as well as the newer Thumb instruction set. The binary translator was extensively tested through both manually written test suites and randomly generated instruction sequences, each time ensuring that the tests run identically whether translated or run natively.

2.5.2 System emulator

The system emulator has three main functions: managing the address space of the translated program, translating system call parameters and handling signals. MAMBO-X64 takes advantage of the 64-bit address space by allocating a 4 GB 'private' address space for the translated application. The program image is loaded into this address space on startup and all memory accesses performed by the application are restricted to this address space since the original code

uses 32-bit memory addresses. This layout, shown in Figure 2.4, isolates the application from the DBT and ensures that it is impossible for faulty applications to affect the operation of the DBT.

The Linux system ABI for AArch64 differs from that for AArch32 in several ways, such as the size and layout of data types used in system calls and the layout of the stack frame when a signal handler is called. MAMBO-X64 therefore needs to emulate the AArch32 Linux ABI by translating the AArch32 system calls generated by the translated program into a format that can be handled by the host kernel. However, Linux exposes a large number of system calls and is constantly evolving[2], which makes it impractical to create and maintain ABI translation wrappers for each of them. Such wrappers are even more impractical for multiplexed system calls, such as `ioctl`, which exposes thousands of device-specific sub-functions.

This complexity can be avoided by reusing the built-in compatibility layer in the AArch64 kernel. This layer is used to support running native AArch32 applications and provides system call wrappers which translate 32-bit system calls into their 64-bit equivalent. MAMBO-X64 intercepts some system calls and handles them internally, such as those used for virtual memory management and signal handling, and forwards the remaining ones to the compatibility layer in the host kernel.

MAMBO-X64 also intercepts all *signals* delivered to the translated program using a master signal handler, which then handles the delivery of the signal to the application. Signal handling in DBTs is complicated because they can occur at any point while executing translated code and require a view of the untranslated register state at the interrupted point to be given to the application signal handler. MAMBO-X64 uses a scheme involving fragment unlinking and signal

[2]At the time of writing, Linux (version 4.5) has 387 different system calls.

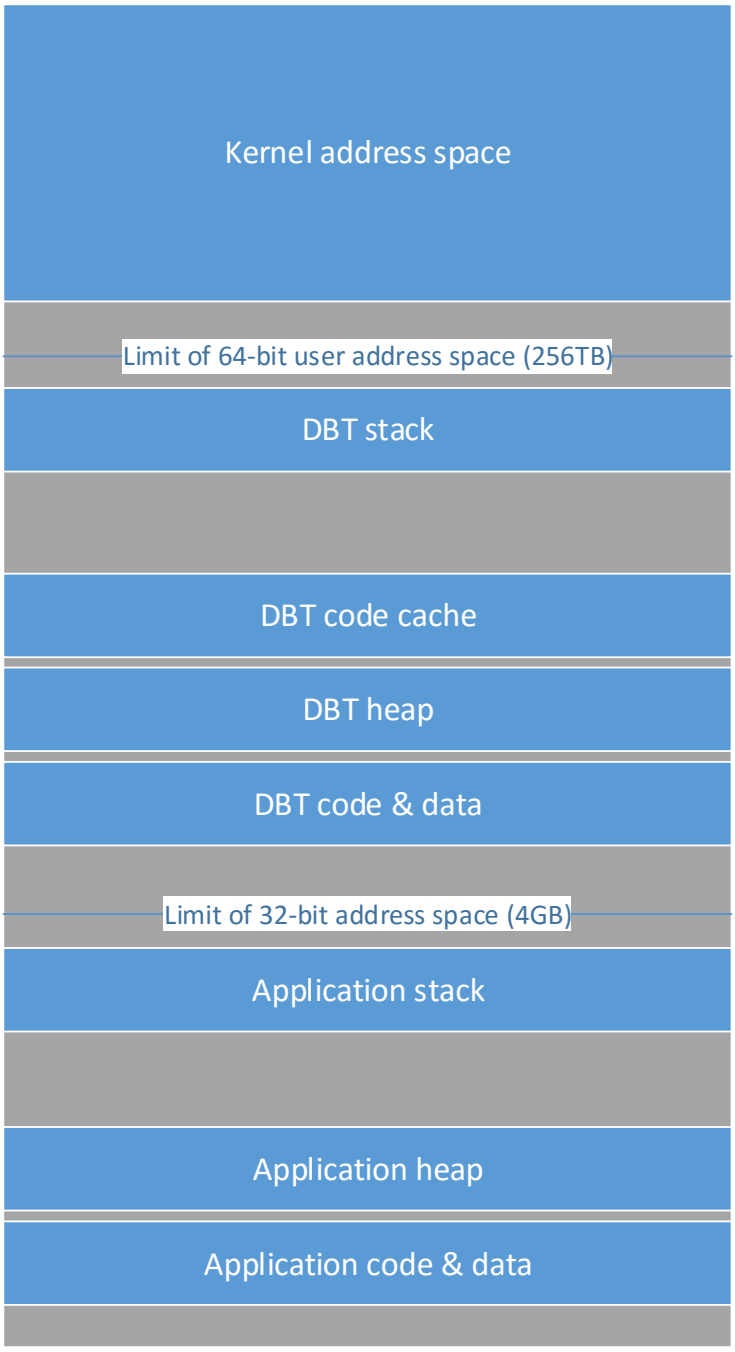


Figure 2.4: Address space layout of an application running under MAMBO-X64.

masking to achieve race-free and efficient signal delivery to the application, which is described in Section 4.3.

2.6 Summary

This chapter has presented an overview of the various concepts underpinning binary translation, such as code caches and transparency. It has also presented MAMBO-X64, the DBT developed by the author, which translates code from AArch32 to AArch64 with the aim of eliminating the need for AArch32 hardware support in new processors. This is made possible through the various optimization techniques implemented by MAMBO-X64, which are presented in the following chapters.

Chapter 3

Optimizing indirect branches in dynamic binary translators

In a typical program, indirect branches mainly come from three sources, as shown in Figure 3.1:

Branch tables Branch tables are an efficient way to branch to many targets by using an array of code addresses in memory.

Function returns Because a function may be called from many places, functions must use an indirect branch to return to their caller.

Function pointers Function pointers and virtual functions are used to dispatch execution to different functions dynamically.

When generated by a compiler, each of these classes has distinctive assembly code signatures which a DBT can detect. This allows the DBT to perform specialized optimizations depending on the branch type. This chapter presents three novel techniques for translating indirect branches which handle each type of indirect branch efficiently.

```
caller:
...
BL callee ; Branch to callee and set link register
          ; to the address of the next instruction
...

callee:
...
BX LR     ; Branch to address in link register (return)
```

(a) Function call and return

```
CMP R0, #3                ; Compare against limit
BHI default               ; Branch to default case if higher
ADR R1, table             ; Get base address of the table
LDR PC, [R1, R0, LSL #2] ; Load from (table + index << 2)
                          ; into the program counter

table:
.word case0
.word case1
.word case2
.word case3
```

(b) Branch table

```
caller:
...
LDR R0, [R1] ; Load a pointer from [R1]
BLX R0       ; Branch to address in R0 and set link register
             ; to the address of the next instruction
...

callee:
...
BX LR        ; Branch to address in link register (return)
```

(c) Function pointer call

Figure 3.1: Indirect branch types generated by GCC when compiling for AArch32.

The first, *hardware-assisted function returns* (Section 3.1), uses a stack of translated return addresses to predict the target of function returns, thus avoiding the need for a hash table lookup. While return address stacks have previously been used in some DBTs [HK06] to predict function returns, they have not always resulted in performance improvements due to the increased number of memory operations and poor interactions with indirect branch prediction hardware. Hardware-assisted function returns are designed to work with the return address predictor of the host processor, while also including optimizations to eliminate many return address stack operations and handle return address mispredictions efficiently.

The second, *branch table inference* (Section 3.2), is a pattern-matching algorithm to detect branch tables during translation and generate a corresponding table in the code cache. Although ad-hoc branch table detection has been explored in DBTs [PG10], the proposed inference provides a systematic way of detecting many variants of branch tables and extracting the bounds of the table directly from the source instructions instead of guessing it.

A hash table is still necessary to handle the remaining indirect branches that are not covered by the previously mentioned techniques. However, most existing hash tables used for indirect branch translation are not designed to work with multiple threads, and require either duplicating the hash table for each thread or introducing expensive synchronization mechanisms. This chapter presents *fast atomic hash tables* (Section 3.3) which take advantage of cheap 64-bit atomic loads and stores to provide a thread-shared hash table that matches the performance of single-threaded hash tables.

These techniques were implemented in MAMBO-X64 early in its development, which allowed it to reach an average performance overhead of only 10 %. This overhead is measured by running a 32-bit build of SPEC CPU2006

on an ARMv8 system both natively and under MAMBO-X64, and measuring their relative performance.

In Section 3.4, these techniques are evaluated on an ARM Cortex-A57 system using the SPEC CPU2006 benchmark suite. The results show that the hardware-assisted function return optimization has the highest impact on performance, with an average overhead reduction of 40 % and up to 90 % on some benchmarks compared to hash table lookups. Branch table inference has a significant effect on benchmarks which make frequent use of branch tables, reducing DBT overhead by up to 40 % in those benchmarks. Finally, fast atomic hash tables are shown to reduce DBT overhead by 40 % compared to existing thread-shared hash table designs, also matching the performance of other indirect branch handling techniques while consuming significantly less memory.

3.1 Hardware-assisted function returns

Research has shown that function returns are by far the most common type of indirect branch [SKC+04]. Function returns are different from other indirect branches in that they usually target the instruction following a previously executed call instruction[1]. In some cases, a function may not return to the address it was called from, but this is atypical, only occurring in exceptional cases such as during stack unwinding after an exception is thrown or if the return address of a function has been modified.

Hardware-assisted function returns take advantage of this property in two ways, first by tracking the addresses of executed call instructions in a software return address stack and secondly by laying out the translated code in a way

[1] Exceptions to this include functions which perform stack unwinding, such as with C++ exception or the C `setjmp` and `longjmp` functions.

that can take advantage of hardware return address prediction logic built into modern processors. While software return address stacks have been used previously in DBTs to optimize function returns [HK06, HH97], this technique extends them by efficiently handling stack overflows and underflows using memory protection hardware and by avoiding the return address stack entirely in certain cases (e.g. 'leaf' functions).

3.1.1 Software return address stack

Hardware-assisted function returns work by maintaining a software *Return Address Stack* (RAS) in memory which tracks previously executed call instructions. Each thread is allocated its own RAS, and the current position in the stack is tracked by dedicated RAS pointer register. To account for the possibility of mispredicting returns, each RAS entry comprises a pair of values: the Source Program Counter (SPC) of the expected return address and its corresponding Translated Program Counter (TPC). An entry is pushed onto the RAS by a translated call instruction and an entry is popped from the RAS by a translated return instruction. The resulting RAS entries therefore mirror the call stack of the program, as shown in Figure 3.2.

Translating a call instruction is simple since all it needs to do is push the SPC and TPC of the assumed call return target, which is the instruction immediately following the call instruction. Translating a return instruction is more complicated due to the need to handle potential mispredictions, and requires four operations:

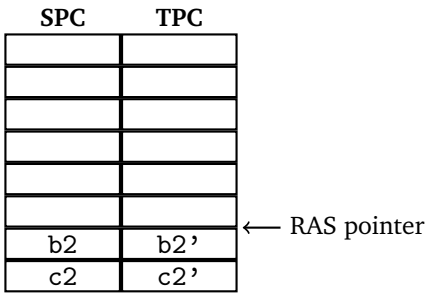
1. An entry containing a SPC and TPC pair is popped from the return address stack.

```
void a() {
    // currently executing
}

void b() {
    a();
b2: // return target after call
}

void c() {
    b();
c2: // return target after call
}
```

(a) Source code



(b) Return address stack contents

Figure 3.2: Return address stack contents while executing nested function calls.

2. The SPC is compared with the program-visible return address used by the return instruction.

3. If the values match then control branches to the TPC in the entry.

4. If the values do not match then control is returned to the DBT so that it can determine the target of the return by translating the SPC.

3.1.2 Hardware return address prediction

Most modern processors include a return address prediction mechanism in hardware to predict the targets of function returns. This specifically detects 'call' and 'return' instructions and passes them to the branch prediction system. Unfortunately this is not used in most DBTs because code before and after the call is treated separately for translation purposes, so they might not place the target of a return immediately after the matching call instruction in the code cache, which is required to exploit the hardware predictor.

Hardware-assisted function returns exploit hardware return prediction by including the return target in the same block as the call instruction. This is done by not regarding a call as the end of a basic block, thus ensuring that the return target is located immediately after the translated call instruction. Translated code can then use native call and return instructions, which take advantage of any return address predictor.

Figure 3.3 shows how a function call and return are translated in MAMBO-X64. The source `BL` instruction is translated into a constant move to set the source link register and a `BL` (call) instruction to branch to the translated function. The code takes advantage of the BL instruction to generate the TPC address in the link register. In the translated function, the source link register and translated link register are both saved to the return address stack. The `BX LR` (return) instruction is translated into a return address stack pop and compare. If the comparison succeeds then a `RET` instruction is used to branch to the translated address from the stack. The `RET` instruction allows the processor to use its return address predictor for this branch. Because the return address stack contains the link register value generated by the `BL` instruction, the processor will predict the target of the return correctly, thus avoiding any penalty from pipeline flushing in the hardware.

3.1.3 Return address stack elision

Rather than pushing an entry to the RAS at the translated call instruction in the caller block, the SPC and TPC of the return target are passed to the callee block in registers. The SPC is passed in the application-visible link register for the source architecture (`R14` on AArch32), while the TPC is passed in the link register for the host architecture (`X30` on AArch64). This TPC value is hidden

```
orig_caller:
BL function     ; branch and set LR
orig_ret_target:
...             ; rest of code

orig_function:
...             ; contents of function
BX LR           ; return using LR (R14)
```

(a) Original AArch32 code

```
translated_caller:
MOV W14, #orig_ret_target      ; calculate return SPC
BL translated_function         ; branch and set TPC
translated_ret_target:
...                            ; rest of code

translated_function:
STP X14, LR, [ras_ptr], #16    ; push SPC and TPC
...                            ; contents of function
...                            ; possibly spread over
...                            ; multiple blocks
LDP X16, LR, [ras_ptr, #-16]! ; pop SPC and TPC
SUB W16, W16, W14              ; compare SPC with LR
CBNZ W16, return_mispredict    ; handle mispredicts
RET LR                         ; return using TPC
```

(b) Translated AArch64 code

Figure 3.3: Translated function call and return in MAMBO-X64. `ras_ptr` is a register that holds the return address stack pointer, `X16` is a scratch register and `W14` contains the AArch32 link register.

from the target application and remains valid as long as the application link register is not modified.

The relationship between these two values is broken when the application link register is modified, either by explicitly overwriting the link register with a different value or implicitly through a call instruction which overwrites this register. In this situation, the SPC and TPC pair needs to be pushed to the RAS just before the register holding the SPC is modified. When a function returns by performing an indirect branch to the address in the source link register, a RAS pop can be avoided if the link register is known to not have been modified since the last executed call instruction. In this situation the host link register already contains the correct TPC address to return to and can be branched to directly.

Consider, as an example, the code in Figure 3.4 which consists of a function with two execution paths. In the first path, the source link register is not modified, which means that a function return in this execution path can be translated to a single branch to the address contained in the host link register, effectively matching the performance of a native function return. This can be done blindly since the DBT statically knows that the host link register holds the TPC for the source link register SPC value. In the second path, the relationship between the source and host link registers is broken when the source link register is modified by a subroutine call, which means that these values need to be preserved in the RAS rather than in registers.

The relationship between the host and source link registers is maintained across block boundaries by creating two variants of every block: a *normal* variant and a *callee* variant. The latter variant has the property that, on entry, both registers will contain valid values. Each variant is generated on demand since in practice most blocks only ever use a single variant: for the SPEC2006 bench-

```
function1:
CMP R0, #0    ; if R0 == 0
BEQ R0, ret   ; skip the call

PUSH {LR}     ; save link reg
BL function2  ; call function2
POP {LR}      ; restore link reg
B ret         ; branch to end

ret:
BX LR         ; return to caller

function2:
BX LR         ; return to caller
```

(a) Original AArch32 code

```
function1-callee:
CBZ W0, ret-callee                (*)
STR W14, [X13, #-4]!              (*)
STP X14, LR, [ras_ptr], #16       (*)
MOV W14, #return_target
BL function2-callee
LDR W14, [X13], #4
LDP X16, LR, [ras_ptr, #-16]!
SUB W16, W16, W14
CBNZ W16, return_mispredict
B ret-normal

ret-normal:
RET LR

ret-callee:
RET LR                            (*)

function2-callee:
RET LR                            (*)
```

(b) Translated AArch64 code

Figure 3.4: Example code showing RAS elision in MAMBO-X64. For instructions marked with `(*)`, `W14` contains the return target SPC and `LR` contains the return target TPC. This property is preserved across branches by making them target callee blocks instead of normal blocks, but is lost when the AArch32 link register (`W14`) is modified, such as by the `BL` instruction, at which point the SPC and TPC values must be saved to the RAS.

marks, on average only 1.5 % of all blocks needed to have both a normal and callee variant generated.

Three rules determine which variant of a block is targeted when a branch instruction is translated:

1. If the instruction is a call then it will always target a callee variant.
2. If the instruction is a non-call branch, the block containing the branch is a callee variant and the source link register has not been modified since the start of the block then the branch will target a callee variant.
3. In all other cases, the branch will target a normal variant.

For indirect branches, two separate hash tables are used, one for normal variants and one for callee variants. Although the target address of an indirect branch can't be determined at translation time, the target variant can be determined because indirect calls (`BLX` on AArch32) can be distinguished from other indirect branch types. The indirect branch is then translated to use one of the two hash tables depending on the variant which needs to be targeted.

3.1.4 Overflow and underflow handling

Because the RAS is allocated as a block of memory of fixed size, it can *overflow* it if a function call is executed when the stack is full. Similarly, it is possible to *underflow* the RAS by attempting to return from a function when the RAS is empty. The former usually occurs when searching through a deep tree structure recursively, while the latter usually occurs when returning from such recursion. A DBT must handle both of these situations to maintain transparency since they could otherwise potentially result in incorrect code execution.

Many RAS implementations handle overflows and underflows by adding bound-checking instructions to the RAS push and pop operations, but this

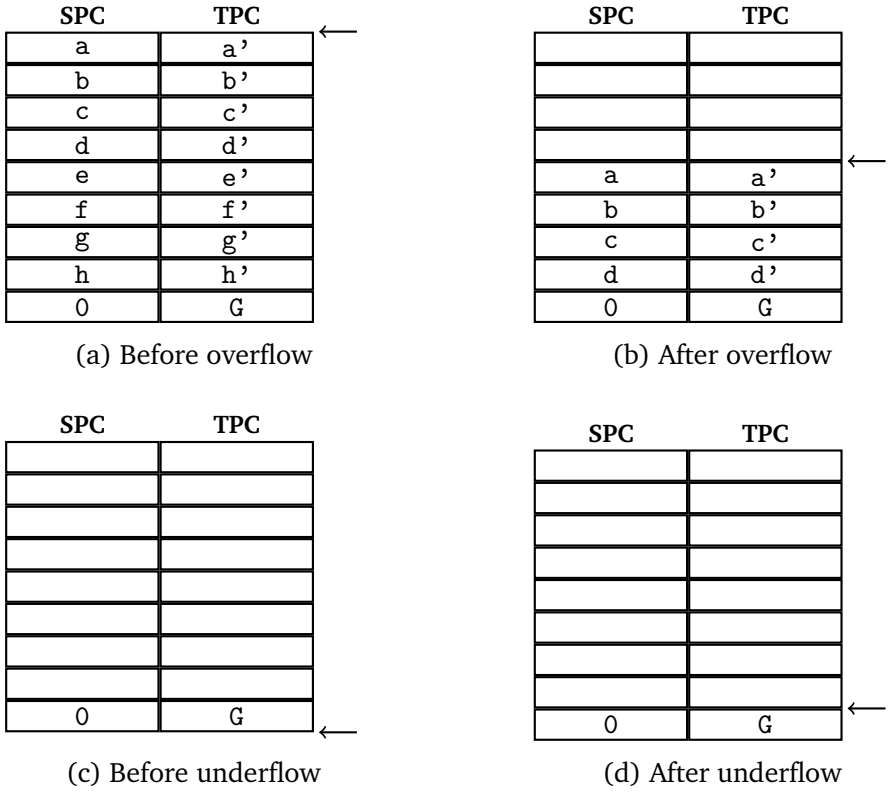
comes at a significant cost due to the additional instructions required multiplied by the high frequency of calls and returns in many programs. A better approach is to use memory protection hardware to trap overflows by allocating a *guard page* at the end of the RAS. Underflows are caught using a *guard entry* that is reserved at the bottom of the stack. Figure 3.5 shows an example of how RAS overflows and underflow are handled.

When the RAS is full and a push is attempted, the write to the guard page will trigger a page fault. The fault handler will shift the RAS contents down: the top half of the stack is copied to the bottom half and the RAS pointer register is adjusted to point to the new top of the stack. Although this discards the bottom half of the RAS entries, which are the least recently used, correct execution is not affected because the RAS is only used as a prediction mechanism. After the stack contents are moved, the push instruction is restarted with the adjusted RAS pointer, which will cause it to push a value into the newly freed space successfully.

To catch underflows, a guard entry is reserved at the bottom of the RAS. This entry contains an SPC address of 0 and a TPC address pointing to a stub that returns to the DBT, so that control returns to the DBT to handle the unlikely event of a return instruction jumping to address zero due to a software error correctly. Once control is returned to the DBT, the current RAS pointer is checked and adjusted to ensure that it always points above the guard entry. A misprediction is unavoidable at this point because there is no prediction information available in the RAS.

3.1.5 Misprediction handling

There are two reasons that can cause a function return to be mispredicted: either the function did not return to its matching call instruction or the pre-



(a) Before overflow

(b) After overflow

(c) Before underflow

(d) After underflow

Figure 3.5: Overflow and underflow handling for the return address stack. On overflow, the contents of the RAS are shifted down, the RAS pointer is adjusted and the push instruction is restarted. On underflow the RAS pointer is moved above the guard entry so that the guard is not overwritten by a later push, and the misprediction is handled by the DBT.

dicted return address was lost due to a RAS overflow and subsequent underflow. These situations can be discerned by checking whether the entry that was just popped from the RAS is the guard entry. Since these situations have different causes, they are best handled separately.

Return misprediction A genuine return misprediction can occur for a variety of reasons, such as stack unwinding when an exception is thrown, calling the C `longjmp` function, switching stacks when invoking a coroutine or simply having a function modify its return address. In many of these cases, program execution will continue normally at an earlier point in the call stack, so it is beneficial to avoid further mispredictions by *unwinding* the RAS to an earlier point. Unwinding is done by scanning the RAS from the top down until an entry matching the current SPC return target is found, and adjusting the RAS pointer to remove that entry and all others above it from the RAS. If a matching entry could not be found then the RAS is left unmodified so that its contents are still available for a later attempt at unwinding.

RAS underflow A misprediction due to a RAS underflow can occur when the call depth of a program exceeds the size of the RAS, causing the RAS to overflow and lose some entries. This is more common than genuine mispredictions and can happen in algorithms which make heavy use of recursion, such as when searching through a very deep tree structure. In this situation it is possible to take advantage of the fact that these algorithms tend to only call a limited set of functions recursively: a small hash table is used to predict these function returns, which contains SPC and TPC addresses of returns that were previously mispredicted due to a RAS underflow. The hash table allows function returns to be predicted even when the relevant return address stack entry

has been lost due to an overflow. This fallback avoids the need to perform an expensive context switch back to the DBT, which can cost hundreds of cycles.

Note that while a return address misprediction can affect program performance, it will never lead to incorrect code execution. This is guaranteed by always tracking the SPC address for each predicted TPC return address and checking that it matches the intended return target.

3.1.6 Unlinking

Hardware-assisted function returns have been designed to work in a *thread-shared code cache* model [BKGB06, HLC09] where the same translated code is shared among multiple threads because this model has been shown to scale significantly better than thread-private code caches. One complication with this model is that block invalidation is more complicated: a block which needs to be deleted because its source assembly instructions have been modified may still have other threads concurrently executing it. This is solved by using *lazy deletion*: all incoming links to the block are removed so that is becomes unreachable and it is freed once all live threads have returned to the DBT at least once since the block was unlinked since this indicates that other threads can no longer be executing the deleted fragment.

There are three ways by which a block can be reached: direct branches from one block to the next, the indirect branch lookup table and a thread's return address stack. Direct branches can be unlinked by having each block maintain metadata about which other blocks have incoming direct branches to it. The branch instructions can then be patched to point to an *exit stub* which returns control to the DBT while passing the current program counter value so that the DBT knows what to execute next. Removing an entry from a thread-shared indirect branch lookup table is discussed in Section 3.3.1.

Removing entries from the RAS of all live threads is more complicated because the RAS is a thread-private structure, which makes it impossible to modify from another thread safely. Using a lock or atomic operations to achieve this is unacceptable as it would slow down normal RAS operations which are performance-critical. Leaving the RAS unmodified is also unacceptable because it violates memory consistency: code returning into an invalidated function will expect to be executing the newly written code.

Rather than attempting to modify the RAS of all live threads, the invalidated block can be patched so that each instruction located immediately after a call is replaced with a branch to an exit stub. This modification is safe to perform while other threads are concurrently executing code in that block because each thread will either see the new code and return to the DBT or it will see the old code and continue execution. Since a thread needs to execute a synchronizing instruction (`ISB` on ARM, any branch on x86) to guarantee that it will execute newly generated code, any thread that is expecting to see the new code will always execute the patched branch and return to the DBT. Once a thread has returned to the DBT, a simple scan of its RAS will remove any entries pointing to invalidated blocks. Therefore, once all threads have returned to the DBT at least once since the block was invalidated, the block can be safely freed since it is not pointed to by the RAS of any thread.

3.2 Branch table inference

Branch tables are used to support multi-way branches efficiently by loading a target code address from a table in memory. C compilers commonly generate branch tables for large `switch` statements. As a whole, this structure is a multi-

```
switch (val) {
    case 0: ...
    case 1: ...
    case 2: ...
    case 3: ...
    default: ...
}
```

(a) Original C code

```
CMP R0, #3
BHI default
ADR R1, table
LSL R0, R0, #2
LDR PC, [R0, R1]
table:
.word case0
.word case1
.word case2
.word case3
```

(b) AArch32 branch table

```
CMP X0, #3
B.HI translated_default
ADR X16, table
LDR X16, [X16, X0, LSL #3]
BR X16
table:
.quad translated_case0
.quad translated_case1
.quad translated_case2
.quad translated_case3
```

(c) AArch64 branch table

Figure 3.6: AArch32 branch table generated by Clang/LLVM for a switch statement, and an AArch64 translation of that branch table. Note that when executed by a processor, the LDR instruction expands to the same micro-operations as the LDR/BR sequence, so the translation introduces negligible performance overhead.

way direct branch since its targets are fixed at a compile time, but, in detail, it makes use of an indirect branch instruction to perform the actual jump.

While a naïve DBT encountering such a structure would simply translate the indirect branch into a hash table lookup, a more advanced DBT could take a holistic view of the structure and optimize it accordingly. Branch table inference is an algorithm to discover (Section 3.2.1) and translate (Section 3.2.2) branch tables. An example of this optimization is shown in Figure 3.6.

3.2.1 Detecting branch tables

Code implementing a branch table comprises several operations:

1. An *index register* is compared to a constant value, which is the number of entries in the table.

2. If the index register value is greater than or equal to the number of entries in the table, control jumps to a default case handler. Otherwise, the index register is known to be within the bounds of the table.

3. The address of the table is loaded into a register, usually as a PC-relative constant.

4. The address of the table entry that should be loaded is calculated by multiplying the index register by the address size (4 bytes in AArch32) using a shift and adding it to the table base.

5. A 32-bit value is loaded from the table using the calculated address.

6. The target address is jumped to using an indirect branch instruction.

While the specific instruction sequence used may vary depending on the architecture, compiler and even compiler options, it always consists of the same operations. In particular, depending on the instruction set, more than one of the listed steps may be performed by a single instruction. For example, the AArch32 branch table code generated by LLVM, shown in Figure 3.6, combines the last three operations into a single instruction.

One significant variation is when branch tables are compiled as position-independent code: in this case, rather than containing absolute target addresses, the table entries contain offsets from a known position, such as the table address or the address of the branch instruction. In such a table the loaded value would be added to base address before being branched to.

Branch table inference can recognize all these variations by scanning the instructions in a block in reverse order once an indirect branch instruction is encountered. During the scan, several conditions are checked:

1. The target of the indirect branch is the result of a word-sized load instruction, optionally added to a constant base address.

2. The address operand of load instruction is the sum of two values A and B.

3. *A* is a constant value known at translation time. This is the base address of the table in memory.

4. *B* is the value of a register *R* multiplied by the word size using a shift (2 places in the case of AArch32).

5. Before the load, there is an exit branch if the condition code indicated a 'greater than or equal' result.

6. The condition used by the branch is generated by comparing *R* to a constant value. This constant value is the size of the table.

While this algorithm will detect most compiler-generated branch tables, such as those generated by LLVM, some compilers use non-standard branch table structures on some architectures, which follow a different pattern. One particular example of this is the branch tables generated by GCC for AArch32 code, which is shown in Figure 3.7. The generated code makes use of several features specific to the ARM architecture to make the branch table more efficient. The branch table code generated in this case can be recognized by simply looking for certain hard-coded instruction sequences.

If a very obscure instruction sequence is used, the table may not be recognized and optimized, but it will still be accommodated by the default indirect branch translation mechanism.

3.2.2 Translating branch tables

Once a branch table code sequence has been identified, the following information needs to be extracted from the sequence so that it can be translated:

- The branch table type: fixed or position-independent.

```
CMP R0, #3
LDRLS PC, [PC, R0, LSL #2]
B default
.word case0
.word case1
.word case2
.word case3
```

(a) Fixed branch table

```
CMP R0, #3
ADDLS PC, PC, R0, LSL #2
B default
B case0
B case1
B case2
B case3
```

(b) Position-independent branch table

Figure 3.7: AArch32 branch tables generated by GCC, which do not match generic branch table patterns. These exploits several features of the AArch32 instruction set: almost all instructions can be predicated, the PC register reads as a value 8 bytes past the current instruction, and writing to the PC register causes an indirect branch.

- The index register (R) and its upper limit, which determines the table size.
- The branch table base address (A).
- For position-independent tables, the indirect branch base address.

Once the address and size of a table are known, all possible targets can be found at translation time, so there is no need to perform a SPC to TPC translation every time the branch is executed. A new branch table is generated which contains the addresses of translated blocks, shadowing each target of the original branch table. Initially, the table only contains pointers to exit stubs that return control to the DBT to translate a block, but these are gradually replaced by the TPC addresses of the blocks as they are translated.

With branch table inference, a translated branch table performs exactly the same operations as a native branch table: a compare, a load and a branch. This results in branch table inference completely eliminating any DBT overhead for this type of indirect branch. Another benefit is that branch table targets are eliminated from the set of indirect branch targets which need to be considered for generic indirect branch lookup. The latter typically uses a hash table lookup

for SPC to TPC translation, which has poor performance when the target of the indirect branch varies a lot from one execution to the next, as is often the case with branch tables.

To ensure that the copy of the branch table in the code cache remains consistent if the branch table entries are modified, this optimization is only performed when the pages containing the branch table are mapped with read-only permissions. This covers all compiler-generated branch tables since these are located in the code or read-only data segments of the executable. Attempts to change the permissions of pages containing a branch table to read-write are caught and the code cache block containing the branch table is invalidated.

3.3 Fast atomic hash tables

When an indirect branch is encountered in translated code, the SPC target of the branch must be translated into a TPC address to continue execution. The standard method for doing this is to use a hash table to store a mapping of SPC to TPC addresses. In a *thread-shared code cache* model, which has been shown to scale significantly better than thread-private code caches [BKGB06, HLC09], this hash table is shared among all running threads. Synchronization of hash table accesses is complicated by the strict performance requirements of indirect branch lookup: because these lookups can occur very frequently, adding any kind of locking for synchronization comes at an unacceptable performance cost.

3.3.1 Hash table operations

Fast atomic hash tables are based on open-addressing hash tables with linear probing. To avoid the need for locks while reading, this technique makes use

of the fact that aligned 64-bit loads and stores are guaranteed to be atomic on many architectures. This is true for all 64-bit architectures and even some 32-bit architectures, such as ARMv7 with the Large Physical Address Extension (LPAE). To take advantage of these instructions, each hash table entry, consisting of an SPC and TPC pair, is packed into a 64-bit value, which allows them to be loaded or stored together atomically.

The hash table supports four operations:

Lookup Hash table lookup performance is by far the most critical since it is the most common operation: typical programs will have billions of hash table reads for each hash table write. The hash table lookup algorithm, shown in Figure 3.8, is simple: it loads entries from the hash table one at a time using a 64-bit atomic load, stopping only when an entry with a matching SPC or an empty entry is reached. The need for bound-checking or wrap-around is eliminated by simply adding a terminating empty entry after the end of the table. For correct execution, the lookup algorithm requires that, at any time, all hash table entries must contain either a valid SPC and TPC pair or be empty. This is guaranteed by having all hash table modifications use 64-bit atomic stores. This algorithm can be efficiently implemented, as shown in Figure 3.9: the implementation in MAMBO-X64 requires only 10 instructions, of which only 8 are executed if a matching entry is found on the first iteration. These instructions are inlined directly into the translated block to avoid call overhead and allow the processor's indirect branch predictor to track each translated indirect branch separately.

Insertion Inserting an entry into the hash table requires only a single 64-bit atomic store to add the entry, but it also requires holding a lock to prevent other threads from modifying the table concurrently. This lock is acceptable because

Input: SPC address of the block that should be executed next.

$Index = \texttt{Hash}(SPC)$ **mod** $HashTableSize$;
repeat
    $Entry = \texttt{AtomicLoad64}(HashTableBase, Index)$;
    $Entry_{SPC}, Entry_{TPC} = \texttt{Unpack}(Entry)$;
    if $\texttt{IsEmpty}(Entry_{SPC})$ **then**
        BranchToDBT();
    end
    $Index = Index + 1$;
until $Entry_{SPC} == SPC$;
$\texttt{BranchTo}(Entry_{TPC})$;

Figure 3.8: Indirect branch lookup algorithm with fast atomic hash tables.

```
AND W18, mask, W15          ; mask the SPC to obtain the table offset
ADD X18, base, X18          ; get a pointer to the hash table entry

loop:
LDR X16, [X18], #8          ; read the entry with a 64b atomic load
SUB W17, W16, W15           ; compare low bits of the entry with SPC
CBZ W17, found              ; break out of the loop if they match
CBNZ W16, loop              ; loop while the entry is not empty
B indirect_branch_miss      ; return to DBT to handle misses

found:
ADRP X17, code_cache_base   ; get the base address of the code cache
ADD X17, X17, X16, LSR #32  ; extract TPC from high bits of the entry
BR X17                      ; branch to TPC
```

Figure 3.9: AArch64 implementation of the indirect branch lookup algorithm. On entry, `W15` holds the SPC target of the branch. `table_mask` and `table_base` are registers that hold the hash table mask and hash table base respectively. All other registers are scratch registers.

unlike lookups, hash table additions and removals are relatively rare in typical programs, and using a lock while writing avoids the need for memory barriers in lookup code. A thread, concurrently reading the table, will either see the new entry or an empty entry. Potential races due to two threads attempting to insert the same entry into the table are resolved once the hash table lock is taken by checking if a matching entry already exists.

Removal An entry needs to be removed from the hash table when a block is invalidated, such as when the instructions it is sourced from have been modified. Entry removal is similar to insertion in that it also requires holding the hash table lock but differs in its effect on concurrent lookups. While the target entry can simply be replaced with a *poisoned* entry, which will always be skipped by lookups, this causes lookup times to grow over time as the number of poisoned entries increases. A better solution is to shift the entries after the target backwards, thus keeping lookup times low. In practice, this shifting usually only consists of one or two swap operations. The shifting can induce spurious failures in concurrent lookups since a lookup might miss its intended target as it is shifted past. This will result in a lookup failure and a return to the DBT, but the lookup can then be resolved by searching the hash table again while holding its lock to prevent concurrent writes. This case rarely occurs in practice and therefore has an insignificant impact on performance.

Growth Growing the hash table is necessary to allow a DBT to scale to a wide range of applications. Since other threads may still be reading the table, it cannot be freed immediately. Resizing instead creates a new, larger table into which the entries of the previous table are copied. Just before a thread starts executing translated code, it will copy a pointer to the latest version of the indirect branch hash table into thread-local storage — which is used for

lookups — and increment the table's reference count. Once it returns to the DBT, the thread will decrement the reference count and free the table if it reaches zero, since that means that no more threads are using the table.

3.3.2 SPC and TPC packing

For fast atomic hash tables to work, both an SPC and a TPC address must be packed into a 64-bit value. There are several ways of achieving this, depending on two factors: the pointer size of the *source* architecture, which determines the size of the SPC, and the pointer size of the *host* architecture, which determines the size of the TPC.

The simplest case is when both the host and the source use 32-bit pointers, in which case they can simply be appended to form a 64-bit value. For a 64-bit host emulating a 32-bit source, the TPC can be turned into a *code cache offset* from the start of the code cache, which can then fit in 32 bits, with the limitation that the code cache cannot exceed 4 GB.

When both the host and the source are 64-bit, the situation is more complicated. While the TPC can be compressed as a code cache offset, the SPC needs to be fully represented to ensure transparency. Fortunately, most 64-bit architectures do not support a full 64-bit address space yet, instead only using a subset of those bits for virtual addresses. For example, the default configuration of Linux on AArch64 restricts the virtual address space of a process to 39 bits, which leaves 25 bits for a code cache offset, allowing a maximum code cache size of 32 MB. The TPC can then be reconstructed by simply adding the 25 bit offset to the starting address of the code cache.

If the source architecture uses more bits for virtual addresses than can fit with this scheme, a DBT can still artificially reduce the address space of a process by controlling memory allocation system calls such as `mmap`. By restricting

all virtual memory allocations with an upper address limit, the DBT can guarantee that all valid SPC addresses can fit in a limited number of bits.

The case of a 32-bit host architecture emulating a 64-bit source architecture is not discussed here because it is rarely used.

3.4 Evaluation

In this section the performance of the indirect branch handling techniques is evaluated. Because the optimizations described in this chapter are designed to reduce the runtime overhead of running a program under a DBT, performance results are described in terms of "overhead reduction". Overhead is defined as the additional time a benchmark takes when running under a DBT compared to running the 32-bit program natively on the CPU, and overhead reduction is the percentage by which this overhead is reduced when the optimization is applied. For example, if an optimization brings the performance overhead relative to native execution from 10 % to 7 % then it is considered to reduce DBT overhead by 30 %.

3.4.1 Experimental setup

All experiments were conducted on a Juno ARM Development Board with two Cortex-A57 cores running at 1.1 GHz and four Cortex-A53 cores running at 850 MHz. The board comes with 8 GB of RAM and runs Debian with Linux kernel version 3.17. To keep results consistent, all experiments were run on one of the Cortex-A57 cores, which has 48 kB of L1 instruction cache, 32 kB of L1 data cache and 2 MB of shared L2 cache.

The performance of the three techniques was analyzed using the SPEC CPU2006 [Cor] benchmark suite. All benchmarks were compiled with GCC

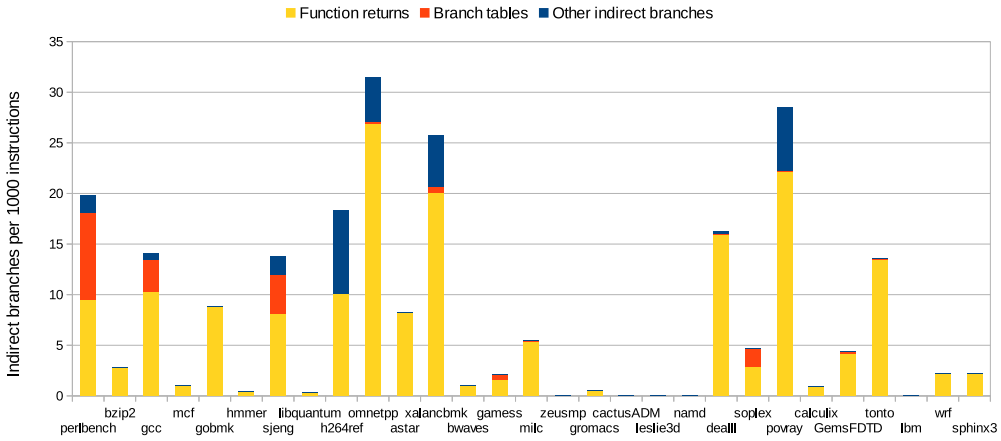


Figure 3.10: Dynamic distribution of indirect branch types in SPEC CPU2006.

4.9.1, optimization level -O2 and targeting 32-bit and 64-bit ARMv8-A. Figure 3.10 shows the distribution of indirect branch types in the SPEC CPU2006 benchmarks as a fraction of the total dynamic instruction count.

Indirect branches account for fewer than 3 % of dynamic instructions executed, but they are a significant source of overhead for DBTs because they need to be translated into a lookup routine that translates an SPC address to a TPC address. While a native indirect branch instruction (BR) only requires a single cycle to execute on a Cortex-A57, a DBT's indirect branch lookup code usually requires 10-20 cycles. This explains why so few instructions can have such a significant impact on DBT performance.

These results also show that in all the benchmarks the majority of indirect branches are function returns, which makes effective handling of returns an important factor in DBT performance. In contrast, intensive use of branch tables and other indirect branches is limited to only a few benchmarks and optimization thereof is thus expected to have less impact on performance.

Because the ARMv8 processors used in these experiments are capable of running AArch32 code directly, all benchmarks were executed natively on the

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.

3.4.2 MAMBO-X64

To evaluate the three techniques, they were implemented on MAMBO-X64. The version of MAMBO-X64 used in these experiments was an early one, which did not yet include the optimizations described in Chapter 4.

Figure 3.11 shows the performance of the benchmarks translated from AArch32 to AArch64 compared to executing the AArch32 code directly. Three different translation methods are shown:

- The benchmarks can be translated using QEMU [Bel05], a generic DBT which supports translating programs among many architectures.
- The benchmarks can be recompiled to AArch64 from source.
- The benchmarks can be translated using MAMBO-X64.

While recompiling the code for AArch64 results in the best performance in most benchmarks, this requires that the source code be available and portable to the new architecture, which may not always be the case. Additionally, pointers in AArch64 use two times more space than on AArch32, which can degrade performance due to increased memory usage and cache pressure despite the additional registers and new instructions in AArch64.

QEMU supports a large number of architectures, therefore it does not use many architecture-specific optimizations and emulates all floating-point operations in software. Together, these design choices cause QEMU's performance to suffer significantly compared to native execution.

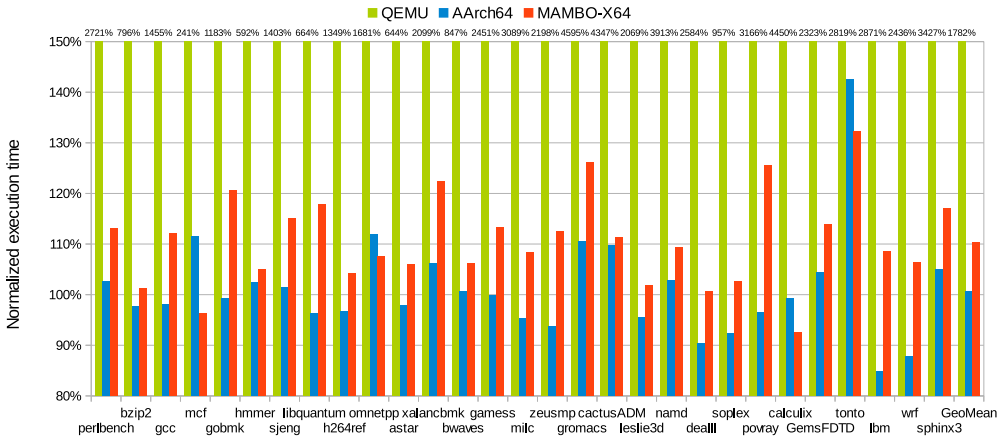


Figure 3.11: Performance of various systems on SPEC CPU2006. These are: QEMU translating AArch32 to AArch64, MAMBO-X64 translating AArch32 to AArch64 and recompiling the benchmarks for AArch64. Performance numbers are relative to the benchmark running natively in 32-bit mode. The results for QEMU do not fit within the graph and are instead shown as percentages above the graph.

MAMBO-X64 is specialized for AArch32 to AArch64 translation, which allows it to reach near-native performance: it is only 10 % slower than native execution on average, and even achieves faster speeds than native execution in some cases. This is achieved by taking advantage of the new features of the AArch64 instruction set which allow a wider range of immediate values to be encoded in instructions and allow certain operations to be translated into in fewer instructions.

3.4.3 Hardware-assisted function returns

Hardware-assisted function returns give a significant performance improvement on about half of the benchmarks, as shown in Figure 3.12. To understand the impact of hardware return address prediction on performance better, the benchmarks were run in four configurations:

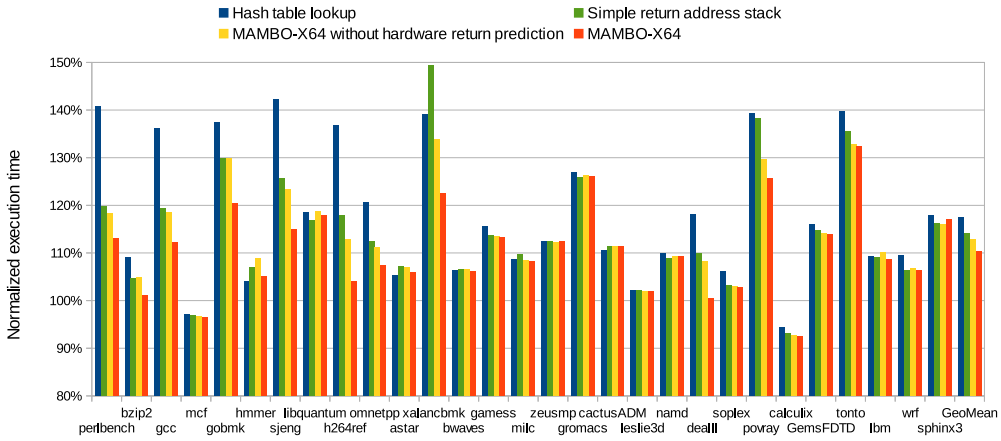


Figure 3.12: MAMBO-X64 performance on SPEC CPU2006 with different ways of handling function returns. Performance numbers are relative to the benchmark running natively in 32-bit mode.

Hash table lookup Function returns are handled using a hash table lookup, just like other indirect branches.

Simple return address stack A return address stack is used, similar to the one used by Pin on ARM [HK06]: it does not support RAS elision or support advanced handling of mispredictions.

MAMBO-X64 All of the techniques described in Section 3.1 are used to optimize function return handling.

MAMBO-X64 without hardware return prediction Similar to the previous, but with the hardware return address predictor inhibited by using `BR` instructions instead of `RET` instruction when translating function returns.

The results show that, while most of the performance improvement comes from the use of a return address stack instead of a hash table, taking advantage of hardware return prediction still accounts for about a third of the overall speedup. Over all of the SPEC CPU2006 benchmarks, the use of a return address stack reduces overhead by 27 % and the use of hardware return address

prediction reduces overhead by a further 14 %. Of the former 27 %, 18.5 % is due to the use of a RAS while the remaining 8.5 % comes from the RAS elision and misprediction handling optimizations. On benchmarks which make heavy use of function calls, such as *h264ref*, the difference is even more significant: using a return address stack reduces overhead by 65 % and hardware return address prediction reduces it by a further 23 %.

Figure 3.13 shows the hardware branch misprediction rate (fraction of branch instructions that were mispredicted) when running MAMBO-X64 under the same four configurations. Unfortunately, the Cortex-A57 does not use separate performance counters for direct and indirect branches, so only the combined branch misprediction rate is shown. These results match the previous ones, which shows that both aspects of hardware-assisted function returns improve DBT performance by reducing branch mispredictions. The additional mispredictions, when using a hash table instead of a RAS, are due to cases where the hardware fails to predict the direct branch in the hash table lookup loop in addition to the indirect branch after the loop.

The RAS can very accurately predict the targets of function returns: the majority of benchmarks do not have any RAS mispredictions. Some benchmarks (*perlbench*, *omnetpp* and *povray*) have RAS mispredictions due to the use of stack unwinding through C++ exceptions and the C `longjmp` function, but these are still extremely rare: the misprediction rate did not exceed one in ten million for any benchmark, which makes the overhead insignificant.

Another set of benchmarks (*perlbench*, *gcc*, *gobmk* and *xalancbmk*) suffer from RAS mispredictions due to RAS overflow. MAMBO-X64 allocates a single 4 kB page for the RAS, followed by a guard page. This allows the RAS to hold 512 entries, which is sufficient since few applications reach a call stack depth of over 500. When the RAS overflows, its contents are shifted down, which

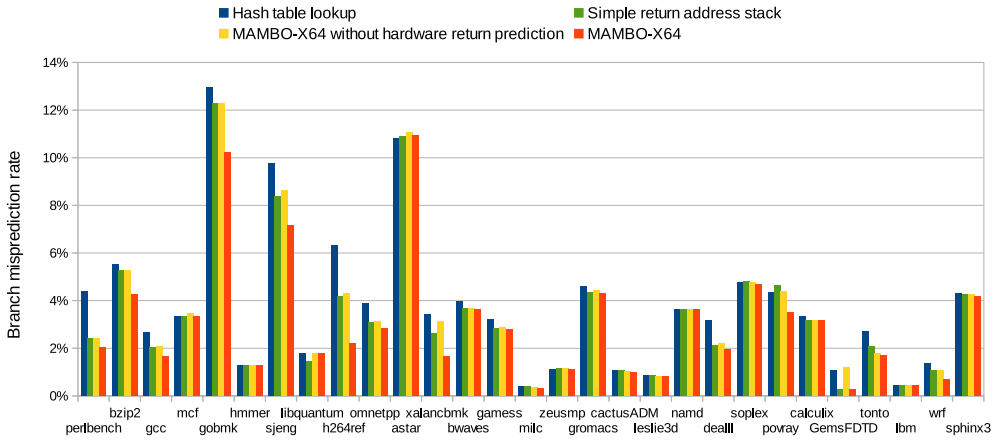


Figure 3.13: Hardware branch misprediction rate of MAMBO-X64 on SPEC CPU2006 with different ways of handling function returns. The numbers show the number of branch mispredictions divided by the total number of branches executed. Note that the misprediction rate includes both direct and indirect branches.

causes it to lose some of its entries. This is most visible in *xalancbmk*, which makes heavy use of recursive functions and overflows the RAS over 600000 times during its 25 minute runtime, causing a RAS misprediction rate of 0.3 % due to the RAS entries lost to the overflows. To avoid context-switching back to the DBT to handle each misprediction, MAMBO-X64 uses the hash table mechanism described in section 3.1.4 to handle overflow-related mispredictions efficiently. This significantly improves performance on *xalancbmk*, where a third of the DBT overhead was due to time spent handling the misprediction in the DBT.

A significant factor in the performance of MAMBO-X64 is the use of RAS elision, which was applied to 51 % of all function returns executed in the benchmarks. This optimization effectively eliminates the overhead of translated function returns compared to native execution. On two benchmarks which make heavy use of function calls and returns, *h264ref* and *dealII*, over 80 % of function returns were optimized with RAS elision.

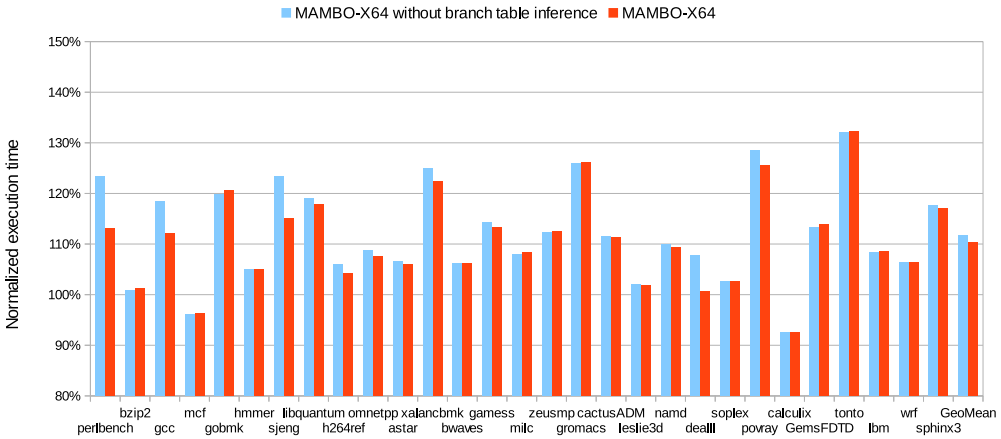


Figure 3.14: MAMBO-X64 performance on SPEC CPU2006 with and without branch table inference. Performance numbers are relative to the benchmark running natively in 32-bit mode.

3.4.4 Branch table inference

Figure 3.14 shows the impact of branch table inference on the benchmarks when run under MAMBO-X64. This optimization mainly affects benchmarks which make heavy use of branch tables: *perlbench* benefits the most because it is an interpreter structured around large `switch` statements, where branch table inference reduces DBT overhead by 40 %. Despite this, the overhead reduction over all of the benchmarks is only of 10 % because few benchmarks rely extensively on branch tables.

3.4.5 Fast atomic hash tables

To evaluate the performance of fast atomic hash tables, three alternative indirect branch handling mechanism were implemented on MAMBO-X64. The first, called “megatables”, is based on SPIRE [JYW+13] but is simplified by exploiting the fact that MAMBO-X64 runs in a 64-bit address space while the program it is translating only uses a 32-bit address space. Instead of using a

small hash table, a huge 16 GB table is allocated, which contains a 4-byte code cache offset for every possible 32-bit address. Handling an indirect branch then simply consists of a simple lookup with no need to handle masking or hash collisions. While the table consumes a large amount of virtual memory, the operating system will only allocate memory for pages which contain entries, while pages with no entries will simply be mapped to a common, zero-filled page. Table modifications are atomic because adding or removing an entry is simply an aligned 32-bit store. As with fast atomic hash tables, entries are added to the megatable lazily, only when an indirect branch lookup misses.

The second mechanism is a hash table that does not require 64-bit atomic operations, similar to the one used by DynamoRIO [BKGB06]. Like fast atomic hash tables, it also uses a single table shared among all threads, but only uses 32-bit loads and stores when accessing the hash table, writing to the SPC and TPC parts of a hash table entry separately. Because of this, backward shift deletion cannot be used to compact the hash table when an entry is deleted; instead the deleted entry must be "poisoned" by replacing the TPC of the entry with the address of a routine that returns control to the DBT. Once an entry has been poisoned, it cannot be reused since another thread may be concurrently reading that entry in a lookup. This causes the average number of entries scanned during a hash table lookup to increase as entries are added and removed, which can degrade performance. Another disadvantage of this method is that, because of the ARM architecture's weakly-ordered memory model, a memory barrier is needed between reading the SPC of an entry and reading its TPC. This memory barrier is part of the performance-critical lookup code and therefore has a significant effect on performance.

The last mechanism is thread-private hash tables, which uses a separate hash table for each thread and therefore does not require any synchronization

during lookups. This is the mechanism most commonly implemented in DBT due to its simplicity and because it works with thread-private code caches. Despite these advantages, it has been shown to not scale well to large numbers of threads [BKGB06, HLC09] since each thread requires its own table. Block invalidation is also more complicated because a block needs to be removed from the hash tables of all live threads although, unlike thread-shared hash tables, this can be done without needing a memory barrier in the lookup code.

Figure 3.15 shows the performance of MAMBO-X64 with fast atomic hash tables, non-atomic hash tables, thread-private hash tables and megatables. In most benchmarks, the performance of these techniques is similar, but some benchmarks which make heavy use of indirect branches, such as *h264ref*, suffer a very large performance degradation due to the use of a memory barrier in thread-shared hash tables. The other three techniques all have very close performance, but some benchmarks, such as *povray* or *dealII*, suffer from hash table collisions that megatables do not suffer from. These results show that fast atomic hash tables have the performance of thread-private hash tables while preserving the superior scalability of thread-shared hash tables.

While megatables and fast atomic hash tables have similar performance and both use a single shared table, the memory usage of megatables is much higher than fast atomic hash tables, as shown in Table 3.1. This is because megatables cause the operating system to allocate a full 4 kB page for every page of the table that contains entries, whereas the size of a hash table is proportional to the total number of entries in that table. This shows that fast atomic hash tables are competitive with existing indirect branch handling techniques while consuming significantly less memory.

Benchmark	Indirect branch targets	Hash table size (KB)	Megatable size (KB)	Ratio
perlbench	316	7.1	431.6	60.7
bzip2	32	1.0	52.0	52.0
gcc	748	14.5	771.2	53.1
mcf	38	1.0	52.0	52.0
gobmk	1021	21.6	347.9	16.1
hmmer	51	1.0	75.9	75.9
sjeng	43	1.0	68.0	68.0
libquantum	29	1.0	48.0	48.0
h264ref	67	1.3	94.4	72.0
omnetpp	497	9.0	388.0	43.1
astar	38	1.0	60.0	60.0
xalancbmk	1120	18.0	1208.0	67.1
bwaves	46	1.0	60.0	60.0
gamess	59	1.0	80.0	80.0
milc	40	1.0	56.0	56.0
zeusmp	48	1.0	52.0	52.0
gromacs	59	1.0	96.0	96.0
cactusADM	127	2.5	200.0	80.0
leslie3d	55	1.0	76.0	76.0
namd	53	1.5	112.0	74.7
dealII	193	5.0	280.0	56.0
soplex	182	4.5	233.9	52.0
povray	175	4.0	248.0	62.0
calculix	65	1.5	68.0	45.3
GemsFDTD	53	1.0	68.0	68.0
tonto	69	1.5	104.0	69.3
lbm	36	1.0	56.0	56.0
wrf	64	1.5	76.0	50.7
sphinx	49	1.0	68.0	68.0
Geometric mean ratio				58.7

Table 3.1: Memory usage of megatables compare to fast atomic hash tables on SPEC CPU2006. In both cases, indirect branches that are handled by branch table inference and hardware-assisted function returns are not included in the tables.

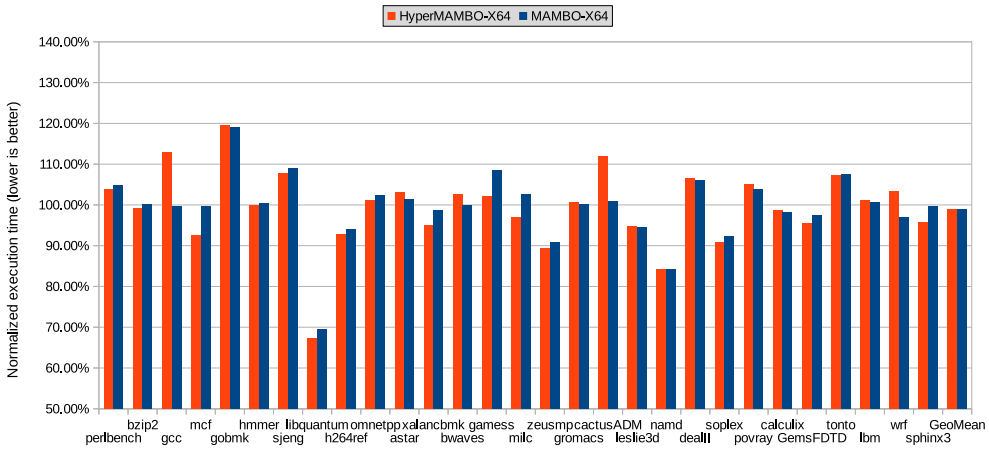


Figure 3.15: MAMBO-X64 performance on SPEC CPU2006 with various indirect branch handling techniques. Performance numbers are relative to the benchmark running natively in 32-bit mode.

3.5 Related work

Because of its significance, many approaches have been designed to reduce the overhead of indirect branches in DBTs. Some of these are generic and can be applied to all types of indirect branches while others only apply to a single type.

3.5.1 Indirect branch handling

The most common way that DBTs handle indirect branches is by using a hash table to map SPC addresses to TPC addresses. The lookup is typically done using a heavily optimized assembly code routine, which can be either called like a function or inlined directly inside a block. In some DBTs [BZA12], these routines also need to save and restore registers to provide scratch registers to work with.

DynamoRIO [BKGB06] implements support for thread-shared indirect branch hash tables, but these are disabled by default because they result in lower per-

formance than thread-private hash tables. This implementation is also specific to x86 and will not work on architectures with a weakly ordered memory model such as ARM without expensive memory barriers in the performance-critical lookup code (Section 3.4.5).

The Indirect Branch Translation Cache (IBTC) [SKC+04] mechanism uses per-branch hash tables instead of a global hash table shared by all indirect branches. While it can potentially offer better hit rates, this approach suffers from increased memory usage compared to a global hash table. It also suffers from the same issues as DynamoRIO's thread-shared hash tables on architectures with a weakly-ordered memory model.

A different approach to handling indirect branches is to use software prediction [LCM+05, BGA03]. This technique consists of comparing the branch target with a pre-defined SPC and branching to the corresponding TPC if the comparison succeeds. Multiple predicted targets can be checked this way, eventually falling back to a hash table lookup if none of the comparisons succeeds. Software Prediction with Target Updating (SPTU) [JYHC14b] is an improvement on this technique which updates the predicted targets according to their frequency. While this approach works well on architectures like x86, which can include an entire word-sized immediate operand with a compare instruction, it is less effective on RISC architectures such as ARM which only support a limited set of immediate operands. Additionally, these comparisons cannot be updated dynamically if the code cache is shared by multiple threads due to the possibility of race conditions.

HDTrans [SSB05] uses a technique called SIEVE which combines software prediction and hash tables. First, the branch target SPC is hashed and used to index a branch table. Each entry in the branch table leads to a chain of compare and branch instructions for all targets in that hash bucket. While

SIEVE requires fewer registers than a standard hash table lookup and can take advantage of hardware branch prediction, it suffers from the same issues as software prediction on RISC architectures.

SPc-Indexed REdirecting (SPIRE) [JYW+13] handles indirect branches by patching the instruction at the SPC in the original program to branch to the TPC, thus allowing an indirect branch to jump to the SPC directly, avoiding the need for address translation. However, an efficient implementation of this scheme is complicated and, as mentioned by the authors, incorrect code execution may occur in certain edge cases.

The fastBT DBT [PG10] uses shadow jump tables to optimize handling of branch tables, which are a subset of indirect branches. When an indirect branch is recognized as a branch table instruction, fastBT will create a shadow table in the code cache containing the TPC addresses of all the branch table targets. The main downside of this technique is that it sometimes ends up creating shadow tables that are too large or too small because it uses fixed size tables instead of inferring the size from the instructions.

Direct-TPC Tables (DTT) [JYHC14a] extend shadow jump tables to apply to all indirect branches which load an address from an array, such as virtual function calls. This is done by shadowing large blocks of program memory and treating them as large branch tables. An indirect branch that loads from an array can then be translated into a load from the shadow table, which will contain the TPC address for the entry. While this technique can offer performance improvements, maintaining transparency requires additional bound and alignment checking code at each indirect branch, which offsets the performance gains.

The Jump Target Lookup Table (JTLT) [KS03] proposed hardware extensions which adds a small hardware cache containing a mapping of SPC ad-

dresses to TPC addresses. When the processor executes an indirect branch instruction, the JTLT is used to find the TPC address for the SPC branch target. If a matching entry is found in the JTLT then the processor branches to the TPC address in the entry. Unfortunately this technique requires special instructions and therefore is not usable on existing architectures.

In summary, hash table-based techniques (DynamoRIO, IBTC) suffer from performance issues in a thread-shared code cache model. Techniques based on software prediction, including SPTU and SIEVE, do not work well with RISC architectures and are incompatible with thread-shared code caches. DTT and SPIRE both suffer from transparency issues, although in the case of DTT this can be corrected at a performance cost. Finally, JTLT requires specialized hardware support.

3.5.2 Function return handling

The results in this chapter (Section 3.4.1), as well as previous research [SKC+04], show that that function returns are by far the most common type of indirect branch. Function returns are different from most other indirect branches in that they almost always return to a matching call instruction. Many techniques have been designed to take advantage of this property to accelerate return handling in DBTs.

While return address stacks have been used in existing DBTs [HK06], they have used normal indirect branch instructions instead of return instructions and thus do not take advantage of hardware return prediction. They also do not implement the optimizations described in this chapter for eliding return address stack operations and for efficient handling of return address stack mispredictions.

An experimental version of DynamoRIO [Bru04] did attempt to combine a software return address stack with hardware return prediction, but the resulting performance was worse than that of using a hash table lookup. This was caused by excessive memory operations and because call return targets were not kept in the same block as the call instruction itself.

Pin [LCM+05] uses *function cloning* to create a different translated copy of a function for each site it is called from. Because each clone is specialized for a single caller, which is known at translation time, a function return can be translated to a compare and branch to the return target. While this provides an accurate prediction of the return address in most cases, it comes at a significant cost in memory, instruction cache locality and translation effort due to code duplication since a new clone needs to be generated for each site a function is called from.

Fast returns [SKC+04] consist of translating call instructions such that they generate the TPC of the return target instead of the SPC for the program-visible return address. This allows return instructions to avoid a SPC-to-TPC lookup because the return address is a TPC value instead of an SPC value. Moore et al. [MBC+09] proposed a variant of this called *checked fast returns* which adds a check to the return instruction to ensure the address is a valid TPC value. While these techniques offer near-native performance for translated return instructions, they can cause applications to malfunction if they attempt to read the return address of a function, for example when generating stack traces or when unwinding the stack.

The *return cache* [SSB05, PG10] is a different approach which works by using a direct mapped hash table which holds return target TPC addresses and is indexed by a hash of the return target SPC address. A return instruction is handled by simply branching to an entry in the return cache. Hash collisions

are handled at the return target by comparing the SPC used for the lookup with the SPC of the return target. The downsides of this approach are that hash collisions are not handled efficiently and that it interacts poorly with hardware branch prediction mechanisms.

The *dual-address return address stack* [KS03] is a technique that requires modifying the processor hardware. It works by extending the hardware return address predictor to track both SPC and TPC addresses. Since this technique requires hardware support, it is not usable on existing architectures.

In summary, fast returns suffer from transparency issues, the dual-address return address stack requires additional hardware and the remaining techniques have performance overheads.

3.6 Summary

This chapter has presented three novel techniques that improve the performance of indirect branches in DBTs. These techniques have been implemented in MAMBO-X64, and their performance impact was evaluated using the SPEC CPU2006 benchmarks. Together, these techniques allow MAMBO-X64 to achieve a very low performance overhead of only 10 % on average compared to native execution of 32-bit programs.

The first technique, *hardware-assisted function returns*, tracks the SPC and TPC addresses of executed call instructions in a software return address stack so that subsequent return instructions can use the last entry on the stack as a predicted branch target, thus avoiding the overhead of SPC to TPC translation. This extends previous work on return address stacks by combining it with a novel layout for translated code that allows the use of the hardware return address predictor in translated code, as well as optimizations to elide

return address stack operations and to better handle return address stack mispredictions. Of the three techniques presented in this chapter, this one has the highest impact when applied to SPEC CPU2006 running under MAMBO-X64, reducing DBT overhead by 40 % on average and by up to 90 % on some benchmarks. This approach has significant benefits over techniques based on hash tables due to the reduced the number of CPU branch mispredictions.

The second technique, *branch table inference*, is an algorithm for recognizing and translating certain code patterns that are used for branch tables. This allows branch tables to be translated as multi-way direct branches rather than as indirect branches by reading the source branch table and generating a corresponding branch table in the translated code. This optimization completely eliminates DBT overhead on branch tables, and significantly improves the performance of benchmarks which make extensive use of branch tables, achieving an overhead reduction of 40 % on some benchmarks. While detection of branch tables has previously been used in DBTs, branch table inference provides a systematic way of detecting many variants of branch tables and extracting the bounds of the table directly from the source instructions instead of guessing it.

The last technique, *fast atomic hash tables*, takes advantage of the fact that aligned 64-bit loads are guaranteed to be atomic on 64-bit architectures and some 32-bit architectures to perform fast indirect branch lookups on a thread-shared hash table. This avoids the need for memory barriers in performance-critical lookup code, which reduces DBT overhead by 40 % compared to a DBT that uses memory barriers. It matches the performance of thread-private hash tables while scaling much better to large numbers of threads. In experiments, it was shown to match the performance of SPIRE, an existing indirect branch handling technique, while consuming 50 times less memory.

While these techniques have been shown to be effective on MAMBO-X64 for translation of AArch32 executables to AArch64, they are also applicable on a wider range of architectures. Branch table inference can work on any architecture as long as a branch table instruction sequence can be recognized. Hardware-assisted function returns can also be generalized, although some additional overhead may be introduced by the need to have scratch registers. Finally, fast atomic hash tables can work on 64-bit architectures but has limitations on 32-bit architectures as described earlier.

Chapter 4

MAMBO-X64: Advanced optimizations and general design

Although indirect branches were the major remaining inefficiency in DBT development, there are a number of other aspects that may still be subject to improvement. This chapter addresses some of these issues and demonstrates that further optimizations can be made.

Section 4.1 describes several of the transformations used by MAMBO-X64 to translate AArch32 instructions into AArch64 code. Of particular note are the scheme for translating AArch32 floating point registers, which builds upon Pin's *register binding* [LCM+05] optimization to allocate registers across multiple fragments dynamically, and the novel *speculative address generation* optimization which efficiently handles integer overflows in address calculations for load/store instructions.

Section 4.2 describes ReTrace, a novel algorithm for trace compilation which takes advantage of hardware return address prediction. This is done by integrating the hardware-assisted function returns algorithm (described in Section 3.1) with the Next Executing Tail (NET) trace compilation scheme [DB00].

Section 4.3 describes the mechanisms used by MAMBO-X64 to translate operating system signals so that they can be handled by the translated application. Signals are particularly tricky for DBTs because they can interrupt the execution of translated code at arbitrary points, which makes handling them susceptible to race conditions. Additionally, the application may rely on highly specific architectural features such as register contents at the interruption point. While existing DBTs have attempted to address this problem in various ways, many of these suffer from race conditions and a lack of transparency. MAMBO-X64 combines several techniques to ensure that signals are delivered to the application correctly and with minimal performance impact.

These techniques collectively reduce the overhead of binary translation, and can even raise the performance above that which can be achieved through hardware support: on an X-Gene XC-1 system, an AArch32 build of SPEC CPU2006 runs 1 % faster under MAMBO-X64 than it does running natively on the same processor, even with translation time included.

These techniques also scale to multiple threads thanks to MAMBO-X64's thread-shared code cache architecture. The overhead, compared to native execution, of running the PARSEC benchmark suite on MAMBO-X64 on that same system is of 2.1 % for 1, 2 and 4 threads, and 4.9 % with 8 threads.

4.1 Translation process

Upon reaching a code address for which there is no translated code fragment, MAMBO-X64 will begin scanning the instructions of the source program until it reaches a control flow instruction[1]. As instructions are gathered, the DBT will also determine the set of input and output registers and condition flags for each

[1]This is typically a branch instruction, but it can also be a system call or other exception-generating instruction.

instruction, which are used in the later stages of the translation process. Once a control flow instruction is encountered the fragment ends, and a reverse pass is done through the instructions to determine register liveness and eliminate instructions with no live outputs.

After the instruction analysis pass has completed, MAMBO-X64 begins translating the block of AArch32 instructions into AArch64 code. While doing so, it also performs several optimizations to improve the generated code, some of which are shown in Table 4.1:

Instruction merging MAMBO-X64 can take advantage of the new instructions in AArch64 to translate sequences of AArch32 instruction into a single AArch64 instruction. For example, floating-point comparisons on AArch32 require two instructions, one to perform the comparison and one to load the result into the condition flags register. This same operation on AArch64 only requires a single instruction which performs both operations. MAMBO-X64 can recognize the AArch32 `VCMP` and `VMRS` pair of instructions and optimize it to a single AArch64 `FCMP` instruction.

Dead code elimination Some instructions can have more than one output, such as an instruction both writing to a register and updating some condition flags. In many cases, some of the condition flags are identifiably 'dead' (i.e. never used), in which case MAMBO-X64 can avoid computing them.

Code layout optimization Some AArch32 instructions have complex behavior that requires many AArch64 instructions to emulate accurately. A large portion of the complexity is due to the need to handle edge-cases which rarely occur in real applications, such as non-default rounding modes or overlong bit shifts (shifting by a value greater than the register

width). When translating these instructions, MAMBO-X64 moves these cold paths outside the main fragment code, which allows the hot paths to execute without needing to take branches.

Constant inlining A common way to load constants into a register in ARM code is by using a PC-relative load instruction. Since this instruction can only generate addresses within 4 kB of the PC, a compiler will mix constants into the code pages of a program. When MAMBO-X64 detects such a pattern, it will copy the constant into the code cache and translate the load into a native PC-relative load of that constant. This is superior to the naïve approach of loading the constant from the code pages of the original program: this optimization helps to reduce data cache and TLB pressure since the translated code no longer has to access the code pages of the original program.

In experiments, the SPEC CPU2006 benchmarks running under MAMBO-X64 executed on average 10 % more instructions than when running natively. However, as shown in Section 4.4, this increased instruction count only has minimal performance overhead. This is because many of these extra instructions do not introduce data dependency stalls and can thus be executed in parallel with other instructions.

4.1.1 Conditional execution

Whereas most architectures tend to only support conditional execution in the form of conditional branches, an unusual feature of AArch32 is the ability to predicate almost all instructions. While this feature enables compact hand-written assembly function, most compilers make limited use of predicated execution and primarily rely on conditional branches instead. This feature was

Original AArch32 code	Translated AArch64 code
`ADDS R0, R1, R2, LSL #2`	`ADDS W0, W1, W2, LSL #2`
`VCMP.F64 D0, D1` `VMRS APSR_nzcv, FPSCR`	`FCMP D0, D1`

`LDR R0, [PC, #data - .]`
 	
`data:`
`.word 0xabcd1234`
	`LDR W0, [PC, #code_cache_data - .]`
	
	`code_cache_data:`
	`.word 0xabcd1234`

`MOVEQ R0, R1` `MOVNE R0, R2`	`CSEL W0, W1, W2, EQ`

`MOV R0, R1, LSR R2`	`AND W17, W2, #0xe0`
	`MOV W0, #0`
	`CBNZ W17, .+8`
	`LSRV W0, W1, W2`

`VCVTR.U32.F64 S0, D1`
`VMOV R0, S0`
	`AND W16, W22, #0xc00000`
	`CBNZ W16, cold_path`
	`FCVTNU W0, D1`
	`continue:`
	`...`
	
	`cold_path:`
	`TBZ W16, #23, .+16`
	`TBNZ W16, #22, .+20`
	`FCVTMU W0, D1`
	`B continue`
	`FCVTPU W0, D1`
	`B continue`
	`FCVTZU W0, D1`
	`B continue`

Table 4.1: Examples of AArch32 instruction sequences translated by MAMBO-X64.

therefore not carried over to AArch64, which only supports three types of conditional instructions: conditional select (`CSEL` and `FCSEL`), conditional compare (`CCMP`) and conditional branch (`B.cc`).

A naïve translation is to simply jump over the translations of predicated instructions if their predicate is false. MAMBO-X64 performs a few optimizations on top of that, such as grouping instructions with identical predicates together and converting adjacent `MOV` instruction with the same destination register but opposite predicates into conditional selects. An example of the latter is shown in Table 4.1.

Conditional branches to other fragments present another difficulty: MAMBO-X64 allocates its code cache as a single 128 MB segment of virtual memory, which allows an AArch64 unconditional branch instruction to jump to any point in the code cache since that instruction has a range of ±128 MB. However, AArch64 conditional branch instructions instead have a range of only ±1 MB, which means that a conditional branch cannot jump to a different fragment directly if it is more than 1 MB away from the branch instruction in the code cache.

To avoid this problem, MAMBO-X64 uses a two stage branch linking scheme. First, the size of individual fragments is limited to 1 MB[2], which guarantees that a conditional branch within a fragment is able to target any instruction in that same fragment. Each conditional branch that jumps to a different fragment is then given an associated *long branch*, which is a single unconditional branch instruction located at the end of the current fragment. MAMBO-X64 will point the conditional branch to the target fragment directly if it is within the range of the conditional branch instruction. If that is not the case then MAMBO-X64

[2]In practice, most fragments contain fewer than a dozen instructions. The largest fragments observed in experiments only consisted of a few hundred instructions.

will point the conditional branch to its associated long branch, and then point the long branch to the target fragment.

A final complication is the handling of the `IT` (If-Then) instruction. This instruction predicates the next one to four instructions, creating an "IT block". Two rules of the ARM architecture allow MAMBO-X64 to determine the predicates of instructions in an IT block statically at translation time:

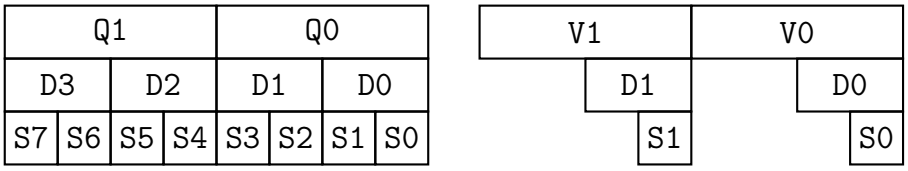
- A branch instruction is only allowed at the end of, or outside an IT block, and
- A branch instruction may not jump into the middle of an IT block.

If code that violates these rules is executed then behavior is considered *unpredictable*, which means that an implementation of the ARM architecture is free to perform any action. MAMBO-X64 takes advantage to these rules to assume that, in almost all cases, fragments do not begin in the middle of an IT block.

There is only one situation in which control can be transferred into the middle of an IT block: exception-generating instructions[3] are allowed in IT blocks, and an exception return from kernel mode is allowed to return into the middle of an IT block by restoring a hidden `ITSTATE` register which holds the state of the IT block when the exception occurred.

MAMBO-X64 handles this situation by statically determining the value of the `ITSTATE` register at each exception-generating instruction and recording it in the fragment metadata used for exception handling (see Section 4.3.1). When an exception occurs, this value is passed to the application signal handler as part of the register context. When the signal handler returns, the `ITSTATE`

[3]This includes the system call instruction, which the ARM architecture allows in an IT block, although this is not fully supported by Linux.



(a) AArch32 (b) AArch64

Figure 4.1: Floating-point register aliasing in AArch32 and AArch64.

value in the register context is used by MAMBO-X64 to create a special fragment which resumes execution with the correct set of predicates for the rest of the IT block.

4.1.2 Register allocation

Translation from AArch32 to AArch64 is interesting because it exposes the problem of both an expanding register bank — general purpose registers — and contracting register bank — floating-point and SIMD registers — as shown in Table 4.2. Because the AArch64 general purpose registers are a strict superset of the AArch32 ones, MAMBO-X64 uses a one-to-one mapping of each 32-bit AArch32 register into a 64-bit AArch64 register. The remaining AArch64 registers are used to hold various AArch32 flags and pointers to DBT data structures, or simply as scratch registers for emulating certain instructions.

However this approach does not work for floating-point registers because AArch32 has *more* floating-point registers than AArch64: an AArch32 program can — by virtue of treating some as single-precision and some as double-precision – use up to 48 floating-point registers (`D16` – `D31` and `S0` – `S31`), while an AArch64 program is limited to a maximum of 32 floating-point registers (`V0` – `V31`). MAMBO-X64 therefore keeps the AArch32 floating-point register state in memory and dynamically allocates registers from `V0` – `V31` to

Register	Description
R0 – R14	32-bit general-purpose registers
SP	32-bit stack pointer, alias for R13
LR	32-bit link register, alias for R14
PC	32-bit exposed program counter
S0 – S31	32-bit floating-point registers (overlaps with D0 – D15)
D0 – D31	64-bit floating-point/SIMD registers (overlaps with Q0 – Q15)
Q0 – Q15	128-bit SIMD registers

(a) AArch32 registers

Register	Description
X0 – X30	64-bit general-purpose registers
LR	64-bit link register, alias for X30
SP	64-bit stack pointer
XZR	64-bit zero register
V0 – V31	128-bit floating-point/SIMD registers

(b) AArch64 registers

Register	Usage in MAMBO-X64
X0 – X14	Mapped to R0 – R14
X15 – X18	Scratch registers
X19	APSR.Q flag
X20	APSR.C and APSR.V flags
X21	FPSCR.NZCV flags
X22	Shadow copy of the FPCR register
X23	APSR.GE flags
X24 – X27	Hash table parameters for indirect branch lookup
X28	Return address stack pointer used for return prediction
X30	Translated link register used for return prediction
SP	Pointer to DBT context on the stack
V0 – V31	Dynamically mapped to floating-point/SIMD registers

(c) AArch64 register usage in MAMBO-X64

Table 4.2: Comparison of AArch32 and AArch64 registers and how MAMBO-X64 uses them.

hold AArch32 floating-point/SIMD register values as they are needed, similarly to register allocation of variables by a compiler.

This dynamic allocation is further complicated by the register aliasing behavior of AArch32, as shown in Figure 4.1. For example, a write to `S0` or `S1` will modify the value of `D0` since these alias. MAMBO-X64 handles such a situation by invalidating any AArch64 register holding the value of `D0` before the write to `S0`/`S1`.

4.1.3 Dynamic register bindings

To avoid having to write modified floating-point register values back to memory before branching to another fragment, MAMBO-X64 tries to keep values in registers across fragment boundaries by using *dynamic register bindings*. This optimization is based on the work previously done in Pin [LCM+05], but has been improved in several ways to make it more suitable for AArch32 floating-point register translation.

This optimization works by creating specialized versions of a fragment, based on the same source AArch32 code but with different register bindings on entry. The register bindings describe which value each AArch64 floating-point register contains and whether it is 'dirty' (different from the in-memory register state). For example, the bindings `[V0=D1 V1=S15!]` mean that the AArch64 register `V0` contains the value of the AArch32 register `D1`, and AArch64 register `V1` contains the value of AArch32 register `S15` which has not been written back to the in-memory register state yet.

Since this requires both the source and target fragments of a branch to agree on a set of register bindings, it is only possible to apply this optimization within fragments or around direct branches. As the target of an indirect branch

is not known in advance, all floating-point register values are written back to memory before taking such branches.

Generating an excessive number of fragments can bloat code cache memory usage and increase instruction cache pressure, which can outweigh the benefits of register bindings. To avoid this, MAMBO-X64 has three mechanisms to reduce the number of fragments that are generated:

Biased register allocation When floating-point registers are allocated, the register allocator will look at all exit branches of the current fragment and gather a list of all existing fragments for the branch targets. It will then try to prefer registers which match the bindings for the branch targets, which can avoid having to create a new fragment variant with different bindings.

Liveness-aware binding matching When linking a fragment into the code cache, MAMBO-X64 will take register liveness in the target block into account when trying to match the bindings of an exit branch with a target block. For example, consider a branch with bindings `[V0=D1 V1=S15!]` and a target fragment with bindings `[V0=D1]`. Normally these bindings would be incompatible since the target fragment expects the value of `S15` to be in the in-memory register state. However if `S15` is known to be dead in the target fragment, then these bindings are compatible since the value of `S15` is never going to be read.

Register binding reconciliation If the number of fragment variants for a single entry point address exceeds a threshold, then new fragments with branches to that address will be forced to reconcile their register bindings with those of one of the existing variant instead of creating a new one.

Type	AArch32	AArch64
No offset	`LDR R0, [R1]`	`LDR W0, [X1]`
Immediate offset	`LDR R0, [R1, #8]`	`LDR W0, [X1, #8]`
Register offset	`LDR R0, [R1, R2]`	`LDR W0, [X1, X2]`
Shifted register offset	`LDR R0, [R1, R2, LSL #2]`	`LDR W0, [X1, X2, LSL #2]`

Table 4.3: Examples of memory addressing modes in AArch32 and AArch64.

4.1.4 Speculative address generation

Load and store instructions in both AArch32 and AArch64 support a similar set of addressing modes[4], of which a few examples are shown in Table 4.3. Despite their similarity however, simply translating the AArch32 addressing modes into their AArch64 equivalent will not always produce correct results. This is due to the address width used by the processor when performing an address calculation, which can affect the result if the calculation overflows.

For example, consider the case where `R1` has the value `0xffff0000` and `R2` has the value `0x40000`. On AArch32, adding these two registers together in an address calculation will wrap around the 32-bit address space and result in the address `0x30000`. On AArch64 however, this will not overflow the 64-bit address width and will instead result in the address `0x100030000`, which is outside the 32-bit address space.

The simplest solution is to use a separate 32-bit `ADD` instruction in the translated code to perform the addition, which will properly truncate the result to 32 bits. The resulting value can then be used as the address for the load/store instruction. While this approach provides correct behavior for all address calculations, it requires an additional instruction for each translated load/store,

[4]AArch32 also supports a variety of more obscure address modes that AArch64 does not, such as indexing with right shifted or rotated registers, but these are rarely used and thus are not a performance concern.

which has a significant impact on performance due to the added data dependency and additional instruction cache pressure.

The fact that AArch32 limits immediate offsets for load/store instructions to ±4095 bytes offers a partial solution to this problem since this limits potential overflows to just 4 kB outside the 32-bit address space. By reserving a guard page at the end of the address space, MAMBO-X64 ensures that out-of-bounds accesses from immediate offset addressing will raise a `SIGSEGV` signal because they will either hit the guard page after the 4 GB address space or wrap around the 64-bit address space and hit inaccessible kernel addresses.

To maintain transparency, MAMBO-X64 also prevents the translated process from mapping the first and last pages of the virtual address space, which ensures that accessing an address both directly and through a wrap-around will produce the same signal (e.g. accessing `0x400` and `0x100000400` both generate `SIGSEGV`). Once MAMBO-X64 catches the signal, it can adjust the faulting address before passing it on to the signal handler of the translated application.

A more general solution, which also applies to register offset addressing, is to use *mirror mappings*, which involve mapping the same physical pages at multiple virtual addresses. Such mappings have been used in recent work to support a variety of use cases [DA06, LWH11, NT14, HDBZ15]. However, creating such parallel mappings on mainstream operating systems such as Linux is not always possible due to the existence of copy-on-write memory mappings, such as those used for anonymous memory and private file mappings, for which the operating system may not propagate modifications to mirror mappings.

A new, more general approach was developed which speculatively assumes that the address calculation will not overflow, which is the case for the vast majority of loads and stores. When translating a register offset load/store instruction, MAMBO-X64 takes advantage of a feature of the AArch64 instruction

set which allows the second operand of a register offset load/store instruction to be sign-extended from 32 bits to 64 bits. This matches the common convention on ARM, which is that the first operand is a base address and the second operand is an offset from that base address. The sign-extension handles the cases where the offset is negative, at the expense of the much-rarer cases where the offset is over 2 GB.

MAMBO-X64 must detect situations where this assumption is invalid and correct them. This is achieved by extending the range of memory reserved by MAMBO-X64 to the first 12 GB of virtual memory. AArch64 allows the second operand in a 32-bit register to be shifted left by one or two places after being sign-extended, which gives it a potential range of ±8 GB. Combined with the first operand, this results in such a load/store being able to access any address from −8 GB to 12 GB. Any overflowing address computations will either fault on the pages reserved by MAMBO-X64 or wrap around and fault in the kernel address space.

Once MAMBO-X64 detects a fault due to mis-speculation, it performs several steps:

1. First, the source address of the mis-translated instruction is recorded in a 'blacklist' hash table which is used during translation to disable speculative optimizations on instructions that are known to mis-speculate.

2. The translated fragment in which the fault occurred is then marked for deletion and any incoming branches to it are redirected elsewhere. This will force the DBT to re-translate the fragment if it is in a loop, this time taking the blacklisted instruction into account.

3. The faulting instruction is emulated in the signal handler, with the resulting address being properly truncated to 32 bits.

4. Execution of the translated fragment is resumed after skipping the mis-speculated instruction.

This system allows MAMBO-X64 to emulate 32-bit load/store addressing modes efficiently while still handling edge cases that overflow the address calculation correctly. Mis-speculation is rare: in experiments, mis-speculation was only observed in the hand-written assembly code in glibc which converts numbers to strings.

4.2 Return-aware trace generation

A significant optimization performed by MAMBO-X64 is *trace generation*, which involves collecting a linear sequence of basic blocks and combining them into a single large code fragment. This results in improved code layout and performance improvements due to the elimination of inter-block branches as well as additional opportunities for optimizations such as dead code elimination.

New code is initially translated into basic blocks, and frequently-executed basic blocks are detected and translated into traces. MAMBO-X64 uses a variant of the *Next Executing Tail* (NET) scheme [DB00] from Dynamo [BDB00] to generate traces. NET works by adding an execution counter to basic blocks which are the target of a backwards branch or an exit from an existing trace, which is incremented every time the basic block is executed.

Once a counter reaches a pre-defined threshold value, the DBT will begin *recording* a trace. This involves following the execution flow of the translated code one basic block at a time until control loops back to the start of the current trace or reaches an existing trace[5]. The basic blocks are then collected and compiled into a single-entry, multiple-exit trace.

[5]There are other conditions for terminating a trace, such as exceeding a size limit, but these are rarely triggered.

Since MAMBO-X64 uses a thread-shared code cache, all running threads will share the same set of counters. Since these counters are only used to detect hot code, they do not need to be exact. By exploiting this property, MAMBO-X64 can avoid using expensive atomic add instructions to increment these counters and use a non-atomic load-increment-store instruction sequence instead. While data races could, in theory, delay trace creation for a basic block indefinitely, this does not occur in practice and a trace is always eventually created.

MAMBO-X64 allocates traces in a separate part of the code cache to keep hot code close together and improve instruction cache locality. This also allows fragments to be linked to each other using conditional branch instructions that have a limited addressing range (±1 MB) rather than having to use an intermediate 'trampoline' branch with a longer range.

4.2.1 Interactions with hardware-assisted function returns

Hardware-assisted function returns (Section 3.1) exploit hardware return prediction by ensuring that the return target of a call is located immediately after the corresponding translated call instruction. This is done by not ending a basic block when a call instruction is encountered. Translated code can then use native call and return instructions, which take advantage of hardware return address prediction automatically. This code layout is necessary because the hardware return address predictor makes the assumption that a return instruction will jump to the address immediately after a call instruction.

One of the characteristics of traces generated by NET is that they can span function calls and returns, which allows NET to inline a function call. However this does not preserve the original call and return instructions, which precludes the use of hardware return address prediction. This property significantly de-

grades the effectiveness of hardware-assisted function returns when used with NET.

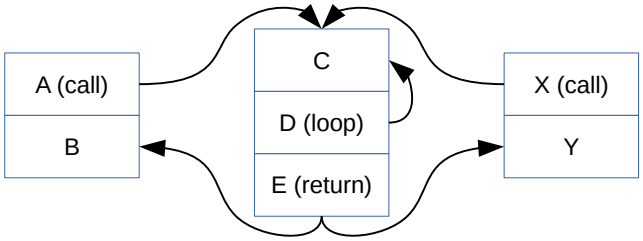
Consider the example in Figure 4.2 which consists of a function containing a loop that is called from two different places. The first trace that NET will create is the inner loop (CD) after it has been executed a sufficient number of times. After the call from A to C is executed a few times, two new traces are created. The first (ACD) crosses over the call instruction to inline the first half of the callee before looping into the CD trace, while the second one (EB) crosses over the return instruction and continues in the caller.

The latter is able to trace across a return instruction by using a *guard*: if the return address is different from the one when the trace was generated then the code will fall back to a hash table lookup to handle the indirect branch. However, this guard can be a big source of branch mispredictions if the function is called from multiple places. This is shown by the traces generated when the function is called from X: while the first block is similar, upon exiting CD control will go to the EB trace, which will then fail its guard and jump to the Y trace after a hash table lookup, thus incurring additional overhead.

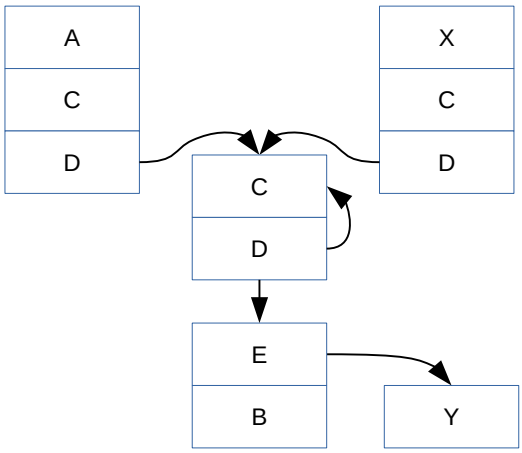
4.2.2 Taking advantage of hardware return prediction

MAMBO-X64 introduces an improved version of NET which is compatible with hardware-assisted function returns, called *return-aware trace generation*, or simply ReTrace. The principle is that traces should not cross function calls or returns. This is implemented by adding the following rules to NET:

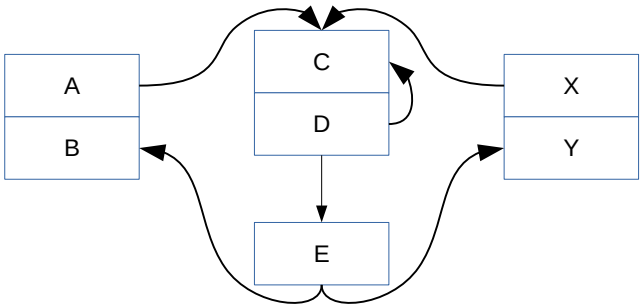
- If a return instruction is reached while recording a trace, the trace is stopped at that instruction.



(a) Original control flow



(b) Translation result with NET



(c) Translation result with ReTrace

Figure 4.2: Example traces showing the differences between NET and the ReTrace algorithm used by MAMBO-X64. It consists of two pieces of code (AB and XY) both calling a single function (CDE) which contains a loop (CD).

- If a call instruction is reached while recording a trace, the DBT will save the current state of the trace and stop recording. An entry is pushed onto the RAS as would normally be done for any call instruction, however the code cache address in the RAS entry will point to a *resume stub* instead. Control is then transferred to the call target normally (without recording).

- Once control reaches a resume stub — which are only reachable through a function return — the saved trace context is restored and trace recording is resumed.

Going back to the example in Figure 4.2, this new algorithm will also create the CD trace first, as it is the inner loop. However the trace starting at A will be different: trace recording is paused after the call instruction and the trace is pushed onto the stack of active traces. Control is then passed to the inner loop trace. When the inner loop exits, it will create a trace E which stops at the return instruction. The return will pop the address of the resume stub from the RAS and resume recording of the trace started at A to form the AB trace. The XY trace is constructed in the same way as the AB trace. The resulting trace layout is much closer to the original code layout than that generated by NET.

The main advantage of this trace layout is that it allows translated code to make use of hardware return address prediction. This is possible because the use of a resume stub allows the return target in the final trace to be located immediately after the call instruction, which matches the expectations of the hardware predictor.

This approach does have some downsides: the generated traces are shorter, which gives MAMBO-X64 fewer opportunities for optimization, and return instructions force all bound registers to be flushed to memory. However, these

disadvantages are outweighed by the performance improvement from hardware-assisted functions returns, as shown in Section 4.4.4.

4.2.3 Avoiding memory leaks

A paused trace is pushed onto a stack when recording reaches a call instruction and popped from that stack when a resume stub is executed. The address of a resume stub is pushed onto the RAS when a trace is paused, which makes it only reachable through a translated return instruction. However if an entry is removed from the RAS through other means then the paused trace will never be resumed. This can occur in two situations under MAMBO-X64:

RAS overflow When the RAS becomes full, attempts to push another entry will trigger a fault on a guard page. The fault handler will free up space by moving the top half of the RAS into the bottom half and adjusting the RAS pointer register accordingly. Since this effectively drops the bottom half of the RAS entries, a thread-local flag is set to indicate that the RAS has overflowed. If the flag is set when control is returned to MAMBO-X64 then any paused traces whose return stubs were dropped from the RAS will be aborted.

RAS unwinding When a RAS misprediction occurs, MAMBO-X64 will attempt to unwind the RAS by matching the given return address with one of the entries in the RAS. If a match is found then all entries above it are discarded, which avoids multiple return mispredictions when the translated program performs stack unwinding (due to an exception, for example). If any return stubs are discarded by this procedure then their matching paused traces are aborted.

4.3 Precise OS signal handling

A *signal* is a mechanism by which an OS can interrupt the execution of an application process to notify it of some event. Such events include external events, such as a timer (`SIGALRM`), or application-generated events such as an unhandled page fault (`SIGSEGV`).

Precise handling of operating system signals is challenging in DBTs because they can interrupt program execution at arbitrary points. When a signal is delivered, the operating system invokes an application-defined *signal handler* function and passes it the execution context at the interruption point. This execution context contains the full register state of the processor at the point where the code was interrupted and is used to resume execution of the interrupted code if the application signal handler returns. This poses several challenges for DBTs:

- Signals are delivered between two instructions, however instruction boundaries in translated code may not match those of the original application code.
- The registers used by translated code will be different from those of the original code, so a DBT must reconstruct the original application register state from the register values of the translated code.
- A DBT may perform optimizations which eliminate writes to registers that appear dead, however these registers must contain a correct value in the context passed to the application signal handler.
- A signal should be delivered to the application in bounded time, otherwise the application may remain stuck in an infinite loop while waiting for a signal.

4.3.1 State reconstruction

Signals can be separated into two types: synchronous and asynchronous. Synchronous signals are delivered in response to a processor-generated exception from a specific instruction, usually a load or store instruction. MAMBO-X64 tracks all potentially exception-generating instructions and ensures that, if an exception occurs, the contents of all AArch32 registers are either directly available or can be derived from the values currently in AArch64 registers.

MAMBO-X64 also creates a table of all instructions that can generate an exception (e.g. load/store instructions) within each fragment, containing the original instruction address, the current register mappings and any other metadata necessary to recover the original execution context if a fault occurred at that instruction. Since this metadata is rarely used, it uses a compact encoding scheme to minimize memory overhead.

Asynchronous signals are delivered in response to an external event, usually outside the control of an application, which means that they can occur at any instruction in the translated code. Extending the previously described mechanism to record metadata for all instructions is impractical because it limits optimization opportunities, increases memory usage and complicates the translation of certain AArch32 instructions (e.g. `LDM`, `STM`) which require multiple AArch64 instructions to emulate. Since these signals are inexact, signal delivery to the application is instead postponed until control leaves translated code and returns to MAMBO-X64, at which point the full AArch32 register state is available in a well-defined state.

4.3.2 Fragment unlinking

While control will naturally return to MAMBO-X64 when the application code tries to execute a system call or when a new block needs to be translated, waiting for such an event to deliver a signal can be a problem if the application is stuck in an infinite loop. To avoid postponing a signal for an unbounded time, MAMBO-X64's signal handler detects whether it has been interrupted in the middle of a fragment and, if so, will *unlink* the exits of the interrupted fragment. This will force any exit from that fragment to return control to MAMBO-X64. There are four ways in which control can exit a fragment:

Direct branches These branches are unlinked by dynamically patching the branch instruction to redirect it to a code stub which records which exit was taken before returning control to MAMBO-X64.

Indirect branches These branches have been translated into an inline hash table lookup. They are unlinked by replacing the hash table pointer with that of an empty table, which will cause a miss and return control to MAMBO-X64. If the signal was delivered in the middle of a lookup then the program counter is rewound back to the start of the lookup so that the new hash table is used.

Function returns MAMBO-X64 tracks function calls and returns using a return address stack to predict the target of return instructions. Unlinking returns is done by replacing the top entry of the return address stack with the address of a code stub, which guarantees that control is returned to MAMBO-X64 whether the return address is correctly predicted or not.

Exception-generating instructions These instructions include system call (`SVC`) and undefined instructions as well as normal loads and stores that may

trigger a page fault. Since they are already translated into a branch that returns to MAMBO-X64, nothing special needs to be done to handle them.

Once control has exited the code cache and returned to MAMBO-X64, all fragment exits are then re-linked to their previous state. Because multiple threads may receive a signal while in the same fragment, a reference count is used to track whether the direct branches in the fragment should be kept unlinked. The fragment is only re-linked once no more threads have a pending signal while executing inside that fragment. Another thread executing an unlinked fragment will only suffer a minor slowdown due to the forced exit to MAMBO-X64, after which it will resume execution without any adverse effects. Moreover, the window for this race condition is very small and has not been observed in any of the tested benchmarks.

4.3.3 Race-free signal delivery

Delaying signal delivery until execution has reached a safe point can lead to race conditions if certain events happen between the DBT receiving the signal from the kernel and delivering that signal to the translated application. These events are:

System call A system call must not be executed while a signal is being held by the DBT, since this could lead to an application missing a signal entirely if the system call involves waiting for a signal. Consider the case of `sigwait`: invoking this system call during the delay would result in the application blocking indefinitely since, from the point of view of the kernel, the signal has already been delivered to the application.

Asynchronous signal Receiving a second asynchronous signal during the delay can be problematic since the application signal handler for the first signal will execute with a different signal mask. If this signal mask would have blocked the second signal then that signal must be kept in a list in the DBT until the application signal mask is changed again to allow it to be delivered to the application. However holding a signal for an extended period can lead to incorrect results from system calls that inspect the set of pending signals in the kernel.

Synchronous signal A synchronous signal from an exception-generating instruction can also lead to similar issues, however this is complicated by the fact that execution cannot continue after such a signal, since attempting to re-execute the exception-generating instruction would simply lead to the same signal being raised again.

To preserve the Linux signal delivery semantics, MAMBO-X64 only delivers a single signal at a time. All signals are blocked while the DBT signal handler is executing, and all signals except those used for fault handling (e.g. `SIGSEGV`, `SIGBUS`) are kept blocked until the signal is delivered to the application. The blocked signals are restored immediately before the DBT starts executing the signal handler set up by the translated program.

If a synchronous fault occurs while a signal is pending then the fault is not delivered to the application. Instead, fragment metadata is used to recover the register state at the fault point and this state is then used when delivering the original signal to the application. The faulting instruction will execute again once the application has finished handling the signal.

MAMBO-X64 handles system calls using an atomic check that only performs a system call if there are no currently pending signals. The code for this is shown in Figure 4.3: if a signal is waiting to be delivered to the application

```
atomic_begin:
    LDRB W9, signal_pending_flag
    CBNZ W9, restart_syscall
    SVC 0
atomic_end:
    RET

restart_syscall:
    MOV X0, #-ERESTARTSYS
    RET
```

Figure 4.3: Code to atomically execute a system call only if there are no pending signals. If a signal is generated between `atomic_begin` and `atomic_end` then the signal handler will rewind the program counter to `atomic_begin`. The `signal_pending_flag` is set when an asynchronous signal has been received by the DBT but not yet delivered to the application.

then the system call will not be executed and an error code indicating a system call restart is returned. MAMBO-X64 will handle this error by immediately delivering the pending signal to the application as if it had occurred immediately before the system call instruction.

Note that this mechanism handles system call restarting transparently, which allows certain system calls that were interrupted by a signal to be restarted automatically once the signal handler returns. The kernel supports this by rewinding the program counter to the system call instruction in the context structure passed to the signal handler. When MAMBO-X64's signal handler inspects this context, it will see that it was interrupted just before the system call instruction and handle it as if the system call had not been executed yet.

4.4 Evaluation

This section evaluates the performance of MAMBO-X64 and how the different techniques contribute to its low performance overhead using the SPEC CPU2006 [Cor] and PARSEC [Bie11] benchmark suites.

Because the ARMv8 processors used in these experiments are capable of running AArch32 code directly, all benchmarks were 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. All benchmarks are compiled with GCC 4.9.1 and optimization level `-O2`.

4.4.1 Experimental setup

The performance of MAMBO-X64 was evaluated on two 64-bit ARMv8 systems. The first 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.

The second system is an Intrinsyc Dragonboard 810 with a Qualcomm Snapdragon 810 processor. The processor is a big.LITTLE configuration with 4 Cortex-A57 out-of-order cores running at 1.96 GHz and 4 Cortex-A53 in-order cores running at 1.56 GHz. The Cortex-A57 cores have 32 kB of L1 data cache, 48 kB of L1 instruction cache and 2 MB of shared L2 cache. The machine comes with 4 GB of RAM and runs Android 5.0.2 Lollipop with Linux kernel version 3.10.49.

4.4.2 Overall performance

Figure 4.4 shows the performance of SPEC CPU2006 when running under MAMBO-X64 on the two test systems. Since SPEC CPU2006 is a single-threaded

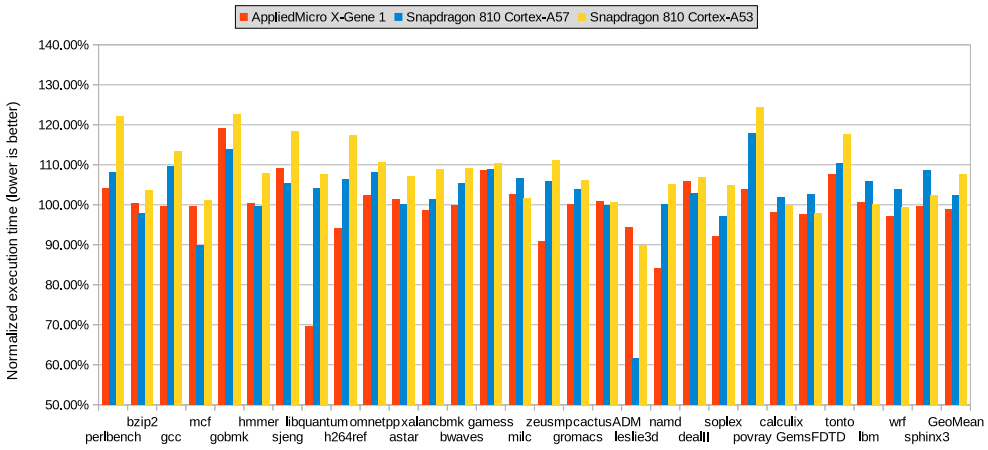


Figure 4.4: Performance of SPEC CPU2006 on MAMBO-X64 on different processors. Performance numbers are relative to the benchmark running natively in 32-bit mode on the same processor.

benchmark suite, it was run twice on the Dragonboard 810: once on an out-of-order Cortex-A57 core and once on an in-order Cortex-A53 core.

These results show that MAMBO-X64 reaches near-native performance on out-of-order cores such as the Cortex-A57 or X-Gene, with a geometric mean overhead of 2.5 % on the former, and a geometric mean performance *improvement* of 1 % on the latter. The geometric mean overhead on the in-order Cortex-A53 core is higher at 7.5 %, but this is likely to improve in the future as MAMBO-X64 has not yet been fully optimized to target in-order cores.

MAMBO-X64 is 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 in to a single AArch64 instruction.

- Previous research in Dynamo [BDB00] has shown that effective trace generation in a DBT can improve runtime performance compared to native execution.

- It has been 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. Eliminating this outlier brings the geometric mean performance down from a 1 % speedup to a 0.25 % slowdown.

Many of the floating-point benchmarks (such as *povray*, *sphinx3*, *tonto* and *gromacs*) have significantly higher overhead on the Cortex-A57 than on the other two micro-architectures. It has been determined that this is due to a peculiarity in the floating-point pipeline of the Cortex-A57 which only affects execution in AArch64 mode [ARM16]; the core will steer floating-point multiply instructions to one of the two floating-point execution units depending on whether the destination register of that instruction is odd or even, rather than picking an idle execution unit, which can lead to load imbalance between the two execution units. ARM has fixed this in newer revisions of the Cortex-A57.

The *gobmk* benchmark performs relatively poorly on all tested systems; it is the only benchmark with an overhead of over 10 % on all systems. This is because the *gobmk* benchmark is instruction cache-bound when run natively. Running the benchmark under a DBT increases the instruction cache pressure which contributes to the performance degradation.

The remaining results in this section are only shown for the X-Gene system for brevity. Additionally, the Snapdragon system produces relatively noisy results, with a typical variation of around ±2 % between runs, whereas the

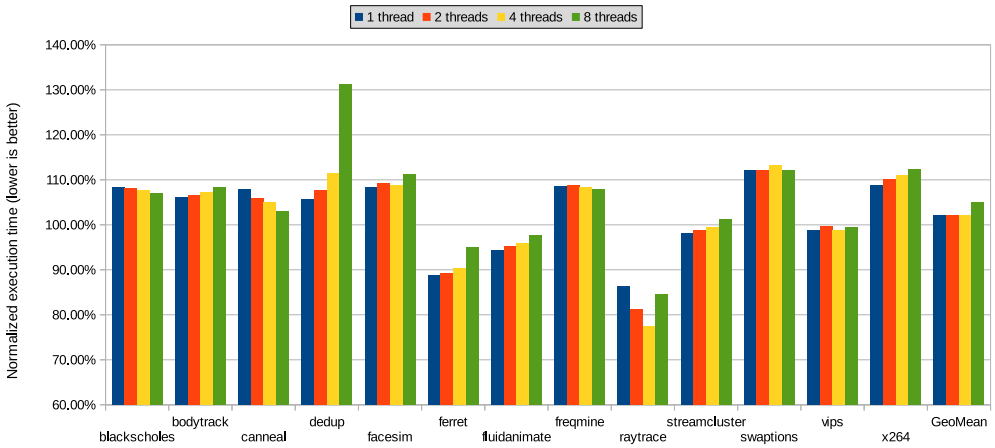


Figure 4.5: Performance of the PARSEC benchmarks running on MAMBO-X64 on the X-Gene system with different numbers of threads. Performance numbers are relative to the benchmark running natively in 32-bit mode with the same number of threads.

X-Gene system has much more stable results with a typical variantion of only about 0.1 %.

4.4.3 Multi-threaded performance

Figure 4.5 shows the performance of the PARSEC multi-threaded benchmark suite when running under MAMBO-X64 on the X-Gene system. The benchmarks were run with 1, 2, 4 and 8 threads since the X-Gene system has 8 processor cores.

These results show that MAMBO-X64 scales well to multiple threads thanks to its thread-shared code cache architecture. Since the code cache is shared among all running threads, code only needs to be translated once instead of having to be re-translated for each thread. This also allows significant savings in memory usage because the code cache and its associated metadata is not duplicated for all threads.

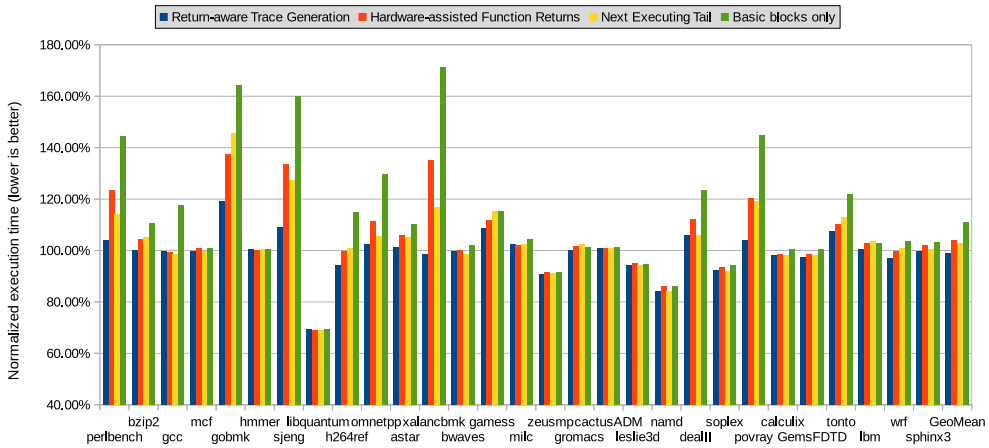


Figure 4.6: Performance of SPEC CPU2006 running on MAMBO-X64 on the X-Gene system with different trace generation techniques. Performance numbers are relative to the benchmark running natively in 32-bit mode.

MAMBO-X64 achieves a low geometric mean overhead of 2.1 % when running PARSEC with 1, 2 and 4 threads, but this overhead climbs to 4.9 % when running with 8 threads. This is mainly due to the *dedup* benchmark having an overhead of over 30 %, which happens because the benchmark only runs for about 16 seconds and does not allow execution of the translated code to amortize the cost of translation. If *dedup* is excluded from the results then the geometric mean overhead drops down to 3.0 %, which is closer to the results with fewer threads.

4.4.4 ReTrace

The effects of trace generation and function call handling on the performance of translated code was also investigated. Figure 4.6 shows the performance of MAMBO-X64 in four configurations:

Return-aware trace generation Combines NET with hardware-assisted function returns using the ReTrace algorithm.

Hardware-assisted function returns Extends basic blocks across call instructions, which allows the use of a return address stack and hardware return address prediction.

Next Executing Tail Combines 'hot' sequences of basic blocks into traces using the NET algorithm. However this does not make use of hardware-assisted function returns because the call structure is not preserved.

Basic blocks only Only translates code into single-entry, single-exit basic blocks and does not make use of the hardware return address prediction mechanism built into the processor.

By themselves, hardware-assisted function returns improve performance by reducing the overhead from 11.1 % to 4.1 %. Similarly, NET alone reduces the performance overhead from 11.1 % to 2.9 %. ReTrace is able to combine the benefits of both of these techniques, allowing it to exceed the performance of native execution.

4.4.5 Register bindings

The effect of inter-fragment register allocation on performance was measured by disabling dynamic register bindings in MAMBO-X64. This is done by forcing all floating-point register values to be written back to memory before any fragment exits and reloading those values from memory as necessary in the target fragment.

The results are shown in Figure 4.7. The effect is minimal on the SPEC integer benchmarks since MAMBO-X64 uses static register bindings for general purpose registers and therefore does not need to write them to memory. However there is significant performance degradation on the SPEC floating-point

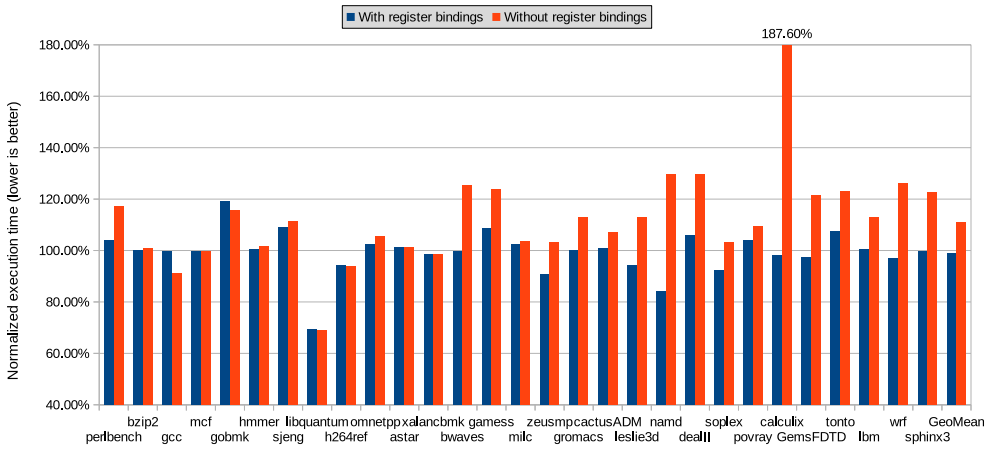


Figure 4.7: Performance of SPEC CPU2006 on MAMBO-X64 on the X-Gene system with and without register bindings. Performance numbers are relative to the benchmark running natively in 32-bit mode. Benchmarks up to *xalancbmk* are part of SPECint, and benchmarks starting from *bwaves* are part of SPECfp.

benchmarks, which is mainly due to the extra memory traffic in the benchmark inner loops. These results show that register bindings offer a significant performance improvement in floating-point applications.

4.4.6 Speculative address generation

The impact of speculative address generation in MAMBO-X64 was tested by measuring the performance effect of disabling this optimization. This involves translating all load/store instructions which use register offset addressing into an `ADD` instruction to perform the address calculation, followed by a load/store instruction using the resulting address.

Figure 4.8 shows that this optimization, unlike the previous one, primarily impacts integer benchmarks as opposed to floating-point benchmarks. This is due to a detail of the AArch32 instruction set: unlike loads and stores to general-purpose registers, memory transfer instructions which target floating-point registers only support a single, immediate offset addressing mode.

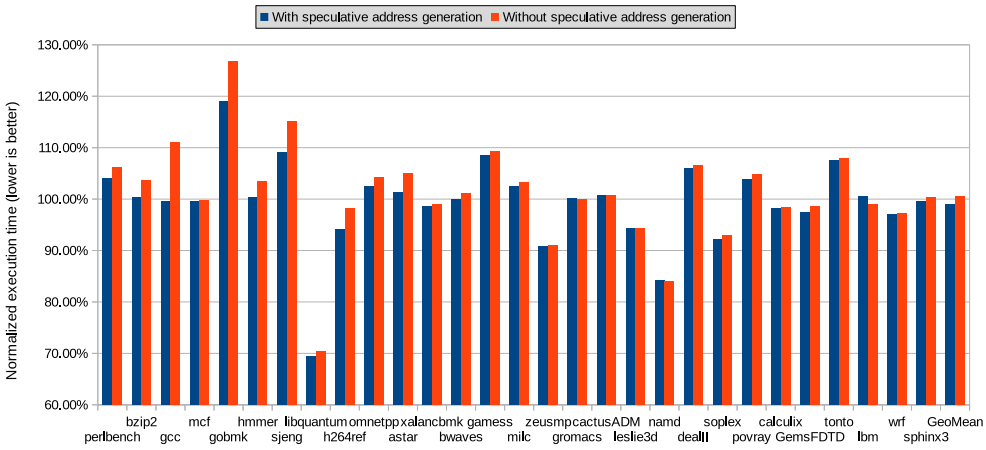


Figure 4.8: Performance of SPEC CPU2006 on MAMBO-X64 on the X-Gene system with and without speculative address generation. Performance numbers are relative to the benchmark running natively in 32-bit mode. Benchmarks up to *xalancbmk* are part of SPECint, and benchmarks starting from *bwaves* are part of SPECfp.

The impact of this optimization is clearly shown in benchmarks such as *bzip*, *gcc* and *hmmer* where the overhead is reduced to almost zero. The *gcc* benchmark has the largest gain due to speculative address generation, going from a performance overhead of 11.1 % to a performance improvement of 0.4 %.

4.5 Related work

Dynamic binary translation is a mature technology, and many DBTs have been created to target different use cases. This section explores some of these systems and compares them to MAMBO-X64.

4.5.1 Dynamic binary translation

QEMU [Bel05] is a sophisticated and detailed environment that allows booting up full operating systems using dynamic binary translation as a key en-

abler. The basic mechanism translates from a source instruction set into the native instruction set using an architecture-independent Intermediate Representation (IR), which is then just-in-time compiled. Another research project, HQEMU [HHY+12], has modified QEMU to use LLVM [LA04] as the IR to try to improve performance by taking advantage of the LLVM JIT backends. However, even with these optimizations, QEMU remains many times slower than native execution (Section 3.4.2). The IR used by MAMBO-X64 is closer to the source architecture which enables performing ARM-specific optimizations such as the handling of AArch32 floating-point registers (Section 4.1.2).

In a different setting, memTrace [PKG13] allows memory accesses in 32-bit x86 programs to be instrumented, which enables the creation of tools for detecting commonly occurring memory errors or watching certain addresses for reads or writes. This tool is based on the fastBT [PG10] binary translation framework which is designed to have low translation overhead. Similarly to MAMBO-X64, it uses the 64-bit memory address space of x86-64 as an integral part of the implementation, as well as translating from 32-bit x86 to x86-64. However, even with instrumentation disabled, it has an average performance overhead of about 17 % (as measured on SPEC CPU2000), unlike MAMBO-X64 which has close to native performance.

StarDBT [WHK+07] is a cross-platform dynamic binary translator which translates 32-bit x86 programs into x86-64 code. It also has the ability to run as a system-level translator which can translate an entire operating system. It has slightly lower performance than memTrace, with an average performance overhead of 27 % compared to native execution.

Binary translation has also been used previously to assist architecture transitions: IA-32 EL [BDE+03] is a tool which dynamically translates 32-bit x86 programs into Itanium instructions and has allowed Itanium processors to elim-

inate hardware support for the x86 instruction set. IA-32 EL uses specialized schemes to translate the x86 floating point register stack and emulate misaligned memory accesses, but these concerns do not apply to the ARM architecture. IA-32 EL uses a mechanism similar to MAMBO-X64 for state reconstruction when a synchronous signal occurs, which works by associating a group of instructions with a commit point, which is typically inserted before store and branch instructions. Each commit point has exception recovery metadata associated with it and instruction reordering is restricted across commit points. MAMBO-X64 is different in that it effectively treats load instructions as commit points, which is necessary to emulate accesses to memory-mapped I/O correctly.

HP Aries [ZT00] is another tool, with a similar purpose to IA-32 EL, which translates PA-RISC programs into Itanium code. Like MAMBO-X64, Aries is also structured as multiple components, separating instruction set emulation from system calls and signal translation, and improving portability to other operating systems. One significant difference from the work presented in this thesis is that Aries uses interpretation instead of translation when encountering code for the first time; it switches to translation when it has identified hot code through profiling in the interpreter. This allows the application to start up faster since rarely used code does not need to be translated.

FX!32 [HH97, CHH+98] is an emulator combined with an offline binary translator which was used to help migrate x86 Windows applications to the Alpha architecture. Like Aries, this system works by first emulating an x86 application using an interpreter while generating profiling data. A background process then generates translated Alpha code from the profile data and stores it in a database. Unlike most similar tools, FX!32 does not translate the operating system ABI at the system call layer. Instead, it provides wrapper libraries for

the entire Windows API which forward to a native Alpha implementation of these libraries, which requires a significant amount of work to maintain as the Windows API evolves.

Rosetta [App06] is the product used by Apple to migrate from PowerPC to x86, based on the QuickTransit [Tra08] technology. QuickTransit is an IR-based dynamic binary translator with different ISA frontends as well as backends, allowing increasing optimization levels to be applied to the most executed parts of the code. Another key feature of QuickTransit is its support for mapping system calls from the application into the native OS, such as when running a Solaris application on a Linux system. MAMBO-X64 uses similar methods to map Linux system calls from 32-bit to 64-bit.

The Transmeta Crusoe [DGB+03] is an x86-compatible processor which uses a binary translator called the Code Morphing Software (CMS) to translate x86 instructions into the VLIW instruction set used by Crusoe. The CMS runs in a privileged execution mode and is completely transparent to the x86 operating system running on the processor. A key feature of the CMS and the underlying Crusoe architecture is that each translated block generated by the CMS acts as a transaction: if execution of a translated block is aborted (e.g. due to a page fault) then the entire register state of the processor reverts to its state at the start of the block, after which the block can be executed by an interpreter, one instruction at a time. While this feature opens up many optimization opportunities, it is not available to MAMBO-X64 because it requires specialized hardware.

The Nvidia Project Denver [BBTV15] is another processor which is similar to the Transmeta Crusoe in that it uses a binary translation layer to translate ARM instructions to an internal VLIW instruction set. Unlike Crusoe, Project Denver includes a simple hardware decoder for ARM instructions to reduce the

performance penalty when encountering code for the first time. Additionally, Denver uses hardware mechanisms to profile code as it runs in the ARM decoder and gathers information such as branch histories, thus avoiding the need to run the code under an interpreter.

DynamoRIO [Bru04] is an open-source dynamic binary modification tool which mainly focuses on x86, albeit initial efforts to support ARM have been developed. Unlike MAMBO-X64, it does not perform instruction set translation and is instead a framework for developing dynamic instrumentation tools. DynamoRIO uses the NET algorithm for trace generation, but this chapter has shown that NET traces interact poorly with return address prediction. A new trace generation algorithm improving on NET was developed (Section 4.2), which preserves the ability to work with return address prediction mechanisms.

Pin [LCM+05] is a product developed by Intel for dynamic binary translation on x86. It functions similarly to DynamoRIO in that it is a framework which allows the creation of specialized tools which modify or instrument a program as it runs. This is supported by a register renaming algorithm which dynamically allocates registers for the original code and for the instrumentation code. Like MAMBO-X64, Pin makes use of dynamic register bindings and register binding reconciliation to preserve allocated registers across multiple translated blocks. MAMBO-X64 applies and improves upon techniques to emulate the AArch32 floating-point registers on AArch64.

Valgrind [NS07] is an open-source dynamic binary modification tool (similar to Pin and DynamoRIO). Although it only supports same-ISA translation, Valgrind translates source instructions into an IR to make the implementation of instrumentation tools easier. One characteristic of Valgrind is that it serializes the execution of multi-threaded applications. This makes the implementation of instrumentation tools simpler by avoiding the need to deal with issues

such as race conditions, but hurts the performance of multi-threaded applications. The main disadvantage of Valgrind is that applications running under it incur a significant slowdown (4.3× on SPEC CPU2000).

HDTrans [SSB07] is another dynamic binary instrumentation tool which, like fastBT, focuses on minimizing the overhead of code translation. This allows it to be much faster than similar tools when executing short-running programs, where translation time is a significant proportion of the total runtime. The key technique to achieve this is the use of table-driven translation rather than code-driven translation. The translation process in MAMBO-X64 is more complicated because it needs to translate between different instruction sets, which makes it poorly suited to table-driven translation.

ArcSim [BFT10, BvKK+11] is an architectural simulator that uses dynamic binary translation for acceleration. The binary translator takes instructions from a given ISA and transforms them into LLVM IR. The JIT code generation is parallelized and it applies optimizations specific for architectural simulation. It has similar functionality to QEMU but with improved execution times. However, like QEMU, it is designed as a retargetable system which precludes it from exploiting architecture-specific optimizations like MAMBO-X64.

Walkabout [CLU02] is a retargetable binary translation framework designed for experimenting with new binary translation techniques. Walkabout achieves retargetability through the use of a specification that describes the target instruction set. This specification is used to generate an assembler, disassembler and emulator for the target architecture automatically. Walkabout also provides a dynamic binary rewriting tool called PathFinder which dynamically translates between two architectures. However this retargetability comes at the cost of about an order of magnitude slowdown compared to native execution in most benchmarks.

4.5.2 Signal handling

The majority of existing DBTs can be split into two categories based on the way they handle signals:

Signal queuing DBTs in this category include QEMU, DynamoRIO, Pin, Valgrind, Aries and IA32-EL. These DBTs handle synchronous signals in a similar manner to MAMBO-X64 by keeping track of all potentially faulting instructions in a fragment and recovering the untranslated register state at the fault point using fragment metadata. Asynchronous signals are handled by appending the signal information to a queue. The execution translated code is then redirected to the DBT, either by setting a flag that is periodically checked by translated code or by patching the translated code directly like MAMBO-X64. While this approach provides correct signal context information to application signal handlers, it suffers from race conditions since the DBT may end up in a situation where multiple signals are queued but cannot be delivered to the application due to signal masking.

Untranslated signals DBTs in this category include HDTrans and fastBT. These DBTs make no attempt to hide themselves from the application: when the DBT receives a signal, it will simply pass on the signal context from the OS to the application unmodified. Since the signal context is not translated, the context of a signal which occurred while executing translated code will show the program counter pointing to a code cache address rather than to the original application code. While this approach benefits from simplicity and low overhead, it can cause applications that rely on precise signal information, such as the Java Virtual Machine (JVM) [PVC01], to malfunction. This approach is generally only viable on same-ISA DBTs

since it requires that the host signal context match the one that the application is expecting.

DynamoRIO is slightly different in that it generates fragment metadata for faulting instructions lazily by re-translating the fragment from its source instructions upon receiving a synchronous signal. While this approach allows for some memory savings, since synchronous signals occur rarely, it requires that fragment translation be deterministic, which precludes the use of certain optimizations such as the biased register allocation technique.

Valgrind has another peculiarity: it blocks all asynchronous signals while running translated code and, instead, periodically polls the kernel to check whether any signals are pending for the current thread. While this model avoids race conditions from delivering multiple signals simultaneously, it suffers from poor performance because the check for pending signals usually requires a relatively expensive system call.

4.6 Summary

The viability of MAMBO-X64 as a replacement for hardware-level AArch32 support is strengthened by its very low overhead compared to native execution on existing ARMv8 processors. MAMBO-X64 was evaluated by taking 32-bit benchmarks from the SPEC CPU2006 suite and comparing their execution times with those of the same benchmark binaries running natively in the AArch32 mode on the same processor. Results show an average overhead of 2.5 % on an out-of-order Cortex-A57 processor and 7.5 % on an in-order Cortex-A53 processor, all of which are significantly lower than those of pre-existing DBT systems. Additionally, MAMBO-X64 achieves an average 1 % *speedup* when running SPEC on an X-Gene 1 processor.

The thread-shared code cache architecture of MAMBO-X64 also allows it to scale well with multi-threaded applications, which are becoming increasingly common as the number of cores in the latest smartphones keeps growing. This has been shown by evaluating the performance of MAMBO-X64 using the PARSEC benchmark suite on an X-Gene 1 system. Results show that MAMBO-X64 achieves an average overhead of only 2.1 % when running with 1, 2, and 4 threads, which climbs to 4.9 % with 8 threads.

MAMBO-X64 is able to achieve such low overheads through the use of optimizations such as instruction merging, dead code elimination, code layout optimization, constant inlining and speculative address generation, as well as highly optimized indirect branch translation. A large part of the high performance of MAMBO-X64 is due to the use of the novel return-aware trace generation algorithm, called ReTrace, which combines the benefits of hardware-assisted function returns and next-executing tail trace generation. These two techniques significantly reduce the overhead of MAMBO-X64 independently, but ReTrace is able to combine the benefits of both to reach near-zero performance overhead.

This chapter has also demonstrated an efficient scheme for mapping the AArch32 floating-point/SIMD register bank onto the effectively smaller AArch64 one. This is done by dynamically allocating the values of AArch32 floating-point registers into AArch64 registers and maintaining allocated registers across multiple translation blocks using a technique called dynamic register bindings. This involves creating specialized translated code fragments based on the same source instructions that accept different sets of bound registers on entry. Unbounded growth of translated code size is avoided by limiting the number of specialized fragments using biased register allocation, liveness-aware binding matching and register binding reconciliation.

Finally, a novel signal handling scheme was implemented, which allows precise delivery of operating system signals while avoiding race conditions and minimizing performance overhead. This works by using fragment unlinking and signal masking to deliver asynchronous signals to the application's signal handler. Synchronous signals are handled by recording fragment metadata for each potentially faulting instruction, allowing a signal context to be recovered for that fault.

These results, without having modified hardware, constitute the best DBT results published so far when moving from a 32-bit to a 64-bit architecture. Such low overheads make the creation of pure 64-bit ARMv8 processors a viable prospect.

Chapter 5

Using hardware virtualization to support high-performance transparent binary translation

A 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 source ar-

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

This chapter describes 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. Each translated code fragment is associated 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 hypervi-

sor, 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.

A prototype of HyperMAMBO-X64 was built on top of the Linux Kernel Virtual Machine (KVM) [DN14] hypervisor and its performance evaluated by running SPEC CPU2006 and several microbenchmarks. Results on SPEC CPU2006 show that HyperMAMBO-X64 is able to match the performance of MAMBO-X64, an equivalent application-level translator. As with MAMBO-X64, experiments 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 [CMR07] and PinOS [BL07] 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. In particular, HyperMAMBO-X64 runs the guest kernel natively in AArch64 mode, which eliminates the need to translate the guest OS page tables.

The rest of this chapter is organized as follows. Section 5.1 presents an overview of binary translation technology and the ARM architecture. Section 5.2 describes the design and implementation of the HyperMAMBO-X64 system. Section 5.3 presents performance results on a selection of bench-

marks. Section 5.4 summarizes some related works and Section 5.5 concludes the chapter.

5.1 ARMv8 virtualization extensions

The traditional ARM architecture is not classically virtualizable [PG74] because it contains several sensitive instructions that have observably different behavior depending on the current privilege level [PKR+13]. While there have been several attempts to support virtualization for the ARM architecture through hardware modifications [BRG+06], binary rewriting [SZP+13] or paravirtualization [DLC+12], 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 5.1 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 specialized applications such as digital rights management and is outside the scope of this chapter.

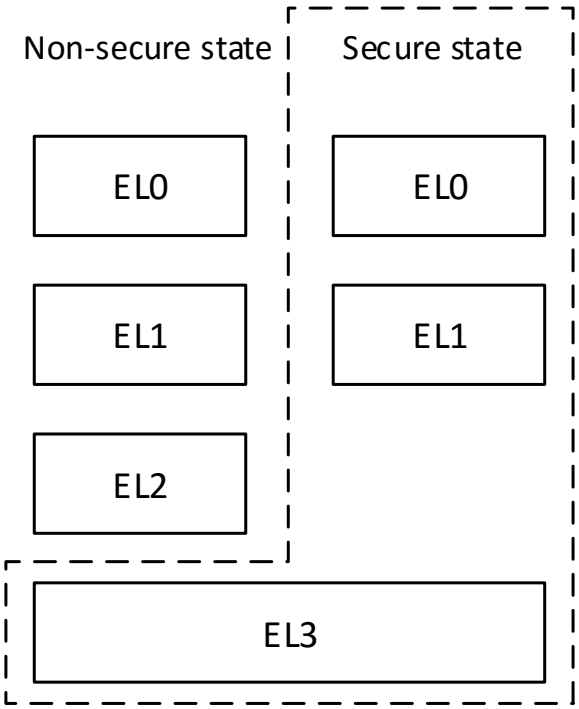


Figure 5.1: ARMv8 exception levels.

- 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.

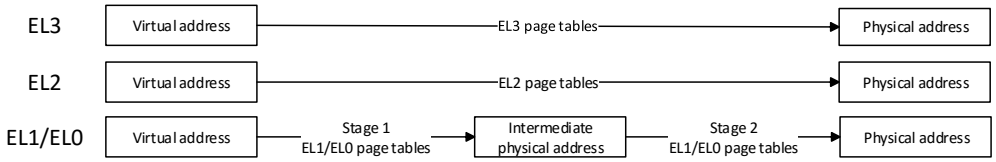


Figure 5.2: ARMv8 virtual memory address translation for different exception levels.

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 5.2. 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.

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

5.2 HyperMAMBO-X64

A binary translation system, called HyperMAMBO-X64, was developed. It 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.

As described in Section 2.3, 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 [CWH+14] or performing guest page table walks in software and caching the results in a software TLB [TKKM14].

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 [BK08, RCCS07] which allow faster startup and can be shared among

[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.

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.

5.2.1 Proposed approach

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, an application-level translator designed to translate AArch32 Linux programs into AArch64 code. The code of MAMBO-X64 was adapted to work in a hypervisor environment without any dependency on either the host or guest OS.

The main disadvantage of the proposed approach compared to a full system-level translator is that it requires 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 the proposed 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 [WYZM16], 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 [ARM13] 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 [CJE+12]. While both clusters typically support the same ISA to allow an operating system to migrate processes from one cluster to another transparently, 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

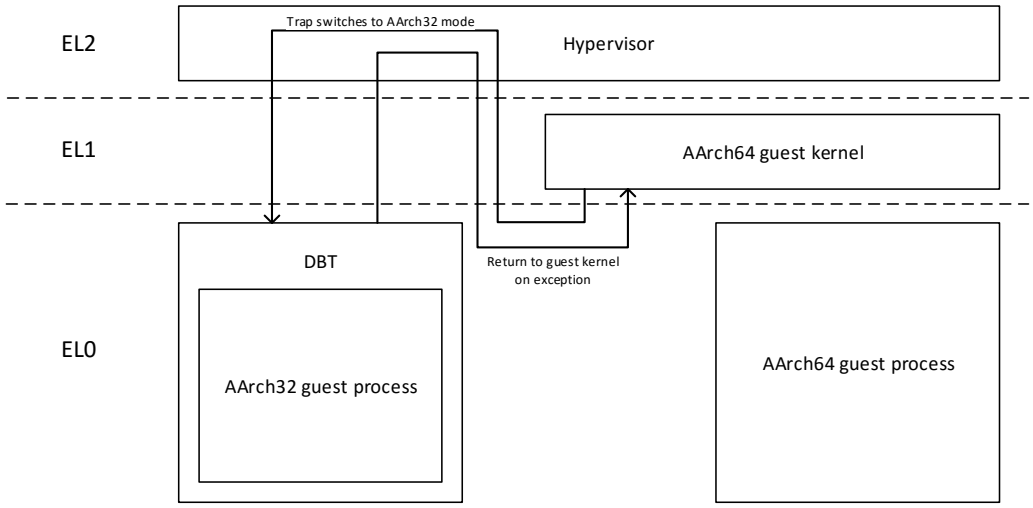


Figure 5.3: Overall architecture of HyperMAMBO-X64.

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.

5.2.2 Architecture

Figure 5.3 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.

5.2.3 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 AArch32 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.

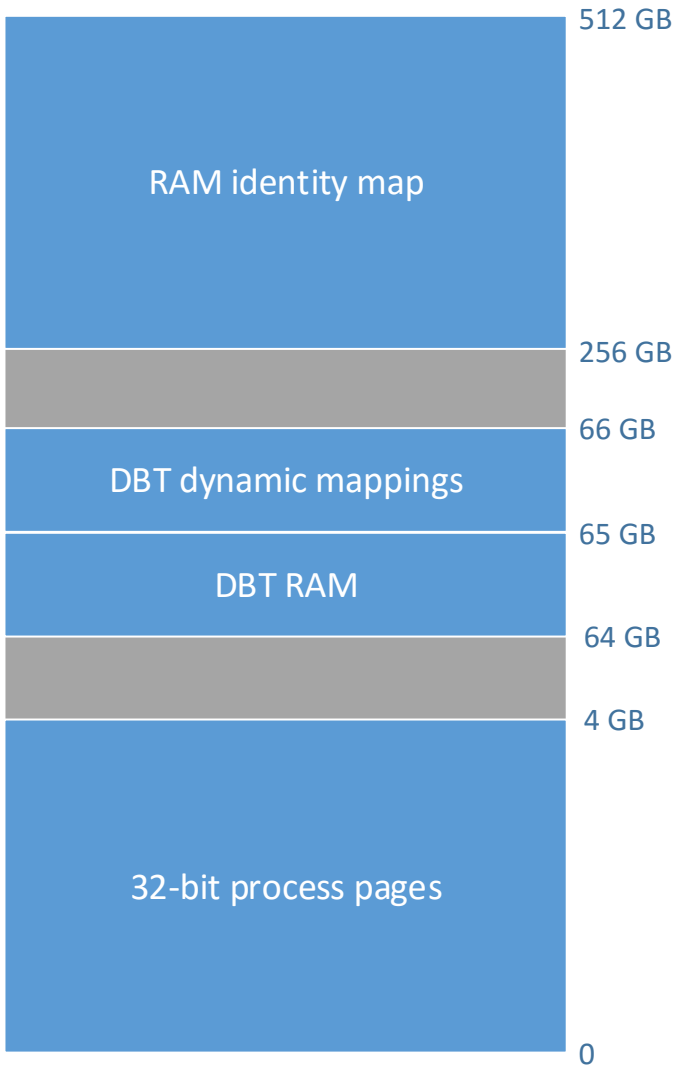


Figure 5.4: Virtual memory map of a process running under the HyperMAMBO-X64 DBT.

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 5.4 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.

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 used for optimizing function returns in the translated code, which requires guard pages to catch stack overflows, as described in Section 3.1.4.

[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.

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 this 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. Trapping guest TLB flush instructions is also exploited for the code cache consistency algorithm described in Section 5.2.4.

5.2.4 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 keeping track of any changes can be done 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, shown in Figure 5.5.

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.

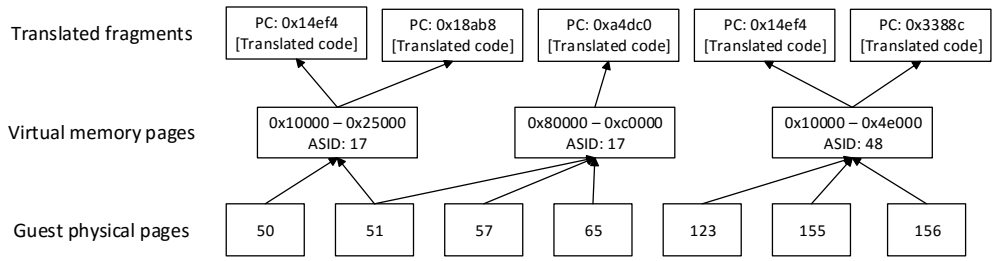


Figure 5.5: HyperMAMBO-X64's data structures for tracking code cache invalidation. Note that a guest physical page may be mapped at multiple virtual addresses or in multiple process address spaces, and fragments from all mappings of this page must be invalidated if that page is modified.

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 several optimizations were implemented 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:

- 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 branch table translation (Section 3.2.2): 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.

5.2.5 Implementation

A prototype implementation of HyperMAMBO-X64 was built on top of the Linux Kernel Virtual Machine (KVM) [DN14] hypervisor. However the general concept is portable to other AArch64 hypervisors such as Xen [BDF+03] or

Xvisor [Pat14]. 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 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 the prototype, this issue was worked around 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. It is anticipated 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.

5.3 Evaluation

This section evaluates the performance of HyperMAMBO-X64 and how it compares to a similar application-level translator using 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.

The same benchmarks are also run 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

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 5.1: Microbenchmark results under HyperMAMBO-X64 in the three tested configurations. All results are in seconds.

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.

The 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.

5.3.1 Microbenchmarks

First, some microbenchmarks which stress particular aspects of the implementation of HyperMAMBO-X64 were run. Four different benchmarks were run, and their results are shown in Table 5.1. These microbenchmarks were chosen because they exercise control transfers between the kernel and the translated application, which require register state reconstruction and pass through the hypervisor.

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. However the results show that there is negligible difference in performance in the three tested configu-

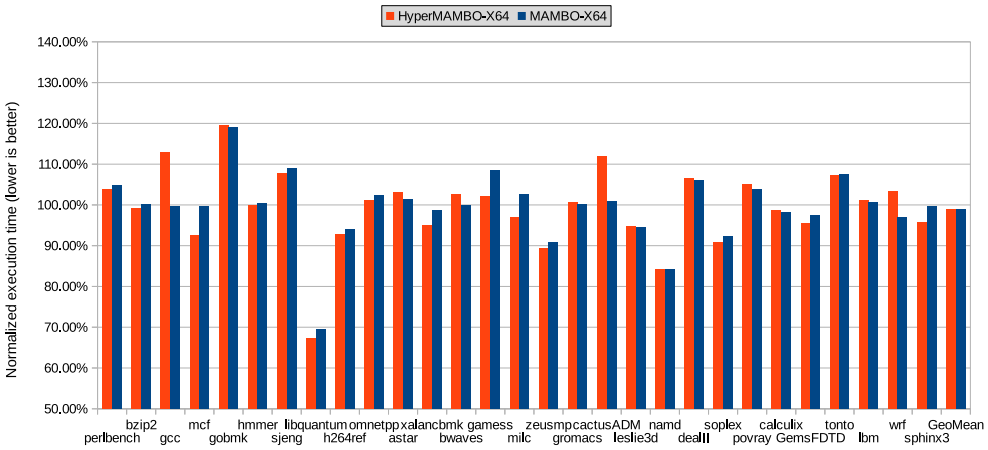


Figure 5.6: Performance of SPEC CPU2006 under HyperMAMBO-X64 and MAMBO-X64. Performance numbers are relative to the benchmark running natively in AArch32 mode.

rations, 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.

5.3.2 SPEC CPU2006

To evaluate the performance of HyperMAMBO-X64 with complex applications, the SPEC CPU2006 [Cor] benchmark suite was run under HyperMAMBO-X64, MAMBO-X64 and natively. Figure 5.6 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.

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 [DB00] 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.

5.4 Related work

MagiXen [CMR07] is probably the closest work to HyperMAMBO-X64, which also integrates a DBT with a hypervisor, in this case to translate a 32-bit x86 op-

erating 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 [BL07] is an extension of the Pin [LCM+05] 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 [Bel05] 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 [BBTV15] and Transmeta's Crusoe [DGB+03] 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 [ZT00] (PA-RISC to IA-64), IA-32 EL [BDE+03] (x86 to IA-64), FX!32 [HH97, CHH+98] (x86 to Alpha) and Rosetta [App06] (PowerPC to x86).

5.5 Summary

This chapter proposed and evaluated HyperMAMBO-X64, a new type of DBT 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.

Chapter 6

Conclusions

Dynamic binary translation has been used in the past to support architecture transition. Whilst providing functionality, there has always been a significant performance overhead. This thesis has demonstrated that this overhead may be reduced or eliminated, not only giving better performance during a transitional period but also providing an *alternative* to the costly backwards compatibility support currently provided by processor manufacturers.

This thesis has primarily focused on ARM's transition from a 32-bit architecture to a 64-bit one, for which MAMBO-X64 — a Dynamic Binary Translator (DBT) which translates AArch32 code into AArch64 code — was developed.

6.1 Summary of contributions

The design of DBTs has tended to be constrained by a trade-off between two factors: **transparency** and **performance**. While DBT design has been an active research area for many years and a multitude of techniques have been proposed to address performance concerns, a large fraction of those have tended to sacrifice transparency in some way (see Sections 3.5 and 4.5). MAMBO-X64

— and the various optimizations described in this thesis — have been designed specifically with transparency in mind and therefore maintain MAMBO-X64's behavioral transparency while still enabling a high performance.

Rather than focusing on a single optimization which brings a large performance improvement, this thesis has presented a range of new optimizations with smaller individual impacts but which, overall, contribute to a large reduction in DBT overhead by addressing the various inefficiencies of DBTs. However, in particular, Chapter 3 has addressed the issue of indirect branch translation, which previous research [KS03, HWH+07] has characterized as the largest source of overhead in DBTs. Chapter 4 has described several other optimizations, some of which are specific to MAMBO-X64, such as AArch32 floating-point register translation and speculative address generation, while others are architecture-agnostic and applicable to any DBT, such as the ReTrace algorithm and precise, transparent OS signal handling.

One of the key achievements of this thesis is MAMBO-X64's ability to *exceed* the performance of native execution when compared to running the same AArch32 benchmark directly on the same processor. While it is not uncommon for a DBT to exceed native performance on one or two benchmarks, MAMBO-X64 was able to achieve this on roughly half of the benchmarks tested on one system, thus reaching an *average* performance on SPEC CPU2006 exceeding that of native execution.

It should be noted that the relative performance of MAMBO-X64 depends significantly on the underlying hardware it executes on, as shown in Table 6.1. The general trend in these results seems to indicate that MAMBO-X64 benefits significantly from processors that support out of order execution and higher levels of instruction-level parallelism. These factors help to hide the higher instruction count in the code generated by MAMBO-X64 compared to the source

Processor	Characteristics	Execution time relative to native
Cortex-A53	2-way superscalar, in-order	107.5 %
Cortex-A57	3-way superscalar, out-of-order	102.5 %
X-Gene 1	4-way superscalar, out-of-order	99.0 %

Table 6.1: Summary of the performance of SPEC CPU2006 on MAMBO-X64 on different processors. The percentage indicates the geometric mean benchmark execution time on MAMBO-X64 relative to native execution.

instructions it is based on. Many of the additional AArch64 instructions are used to maintain transparency by emulating edge cases in certain AArch32 instructions and thus do not contribute to data dependencies which might block application progress.

Despite MAMBO-X64's focus on transparency, there are fundamental limitations to the level of transparency that can be provided by an application-level translator since it is treated as a 64-bit process by the operating system. Conversely, while system-level translators allow for better transparency, this typically comes at a cost in performance due to the overhead of address translation. Chapter 5 presented HyperMAMBO-X64, a novel hybrid of these two types of translators, which exploits the ARM virtualization extensions to provide full OS-level transparency while preserving the performance of application-level translators. This approach also has the benefit of being OS-independent, unlike MAMBO-X64 which would have to be manually ported to non-Linux OSes. While the current prototype of HyperMAMBO-X64 requires the guest kernel to be modified, this restriction can be lifted on a pure AArch64 processor since that would allow a hypervisor to trap attempted switches into AArch32 mode.

A common theme in the optimization techniques explored in this thesis is the exploitation of hardware features to improve performance. The most obvious examples are hardware-assisted function returns and ReTrace making use of the processor's return address prediction mechanism, but other exam-

ples include the use of guard pages to catch return address stack overflows, exploiting atomicity for indirect branch translation tables, taking advantage of the larger 64-bit address space and using the ARMv8 virtualization extensions to maintain code cache consistency in HyperMAMBO-X64.

6.2 Future research

While this research has demonstrated low overheads for a dynamic binary translator, there remain several areas in which further performance improvements may be gained.

6.2.1 Improved startup times through pre-translation

The optimizations explored in this thesis have largely focused on improving the performance of translated code, but with relatively little regard for the performance of the translation process itself. This works well for benchmarks and long-running server applications, since time spent translating code is amortized over the runtime of the application. Even then, some of the benchmarks with shorter runtimes, such as *dedup* which runs only for a dozen seconds, show a measurable overhead attributed to code translation (Section 4.4.3).

However, the effect of code translation overhead is the most visible in interactive applications: in a test of a 32-bit Android application, the application took over two minutes to start up under MAMBO-X64, compared to only ten seconds when running natively. While the application ran smoothly once it had finished loading, such a long loading time is unacceptable for an interactive application and delivers a poor user experience.

The traditional approach for solving this problem is to use a *persistent code cache* [BK08, RCCS07] which is built up incrementally as the application runs

and preserves translated code across multiple executions of the application. While this approach will indeed accelerate subsequent launches of the application, the first launch will still suffer from a significant slowdown, which can severely damage a user's first impressions.

An alternative approach is to use a form of off-line 'pre-translation' where an executable image is pre-processed to find all function entry points, for which translated code can be generated and saved to a persistent code cache. This pre-processing does not have to be precise — this is impossible to guarantee due to the code discovery problem — since any 'missing' code will just be translated dynamically instead. This process can be performed when an application is installed, where the translation delay is less likely to be noticed by the user.

6.2.2 Automatic vectorization

Most modern computer architectures feature some form of Single Instruction Multiple Data (SIMD) [Fly72] processing functionality, which allows applications to exploit data level parallelism in processors. This functionality is typically exposed through dedicated SIMD instructions, which an application can use explicitly through inline assembly and compiler intrinsic functions, or implicitly through compiler-driven automatic vectorization [Wol96].

Automatic vectorization is typically performed by static source language compilers since they have a high-level view of the source code and more freedom for code transformations than binary translators, but this is not the case in AArch32. While scalar floating-point instructions (VFP[1]) in AArch32 follow the IEEE 754 floating-point standard, this is not the case for AArch32 SIMD floating-point instructions (NEON) because they do not handle 'denor-

[1] Despite VFP standing for "Vector Floating Point", the vector functionality of this ISA extension has been deprecated and modern implementations of VFP only support scalar operations.

mal' values correctly, instead simply rounding them to zero. Because of this, a standards-compliant compiler cannot make use of these instructions to vectorize floating-point operations[2].

AArch64 is not subject to this limitation because its floating point instructions — both scalar and SIMD — fully support denormal values. Since AArch32 compilers are not able to perform automatic vectorization, this presents an opportunity for MAMBO-X64 to do so and, potentially, achieve significant performance improvements compared to native execution.

While automatic vectorization in binary translators has been explored previously [NMO11, YF08], this work has not taken transparency into account, particularly with regards to synchronous signals from faulting load/store instructions. Most existing auto-vectorization algorithms will reorder and merge load/store instructions to exploit SIMD instructions which load multiple values into a single vector register, but such transformations may violate transparency when a fault occurs if the precise source register state at the fault point cannot be reconstructed.

6.2.3 Improving the handling of dynamically generated code in HyperMAMBO-X64

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 modification of code pages, is becoming increasingly common to accelerate the execution of scripting languages such as Javascript.

[2]GCC can be forced to use NEON instructions for auto-vectorization by enabling the `-funsafe-math-optimizations` flag.

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 [HDBZ15]. 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.

6.2.4 Persistent code caching for system-level translators

Persistent code caching allows translated code to be re-used across multiple executions of an application, and for multiple processes to share the same code cache, thus reducing overall system memory usage. A key factor in existing persistent code caching techniques is that they require associating a translated fragment with a persistent version of the source instructions. In application-level translators, this is typically the executable image or shared library file that is mapped into the translated application's address space. This approach does not work for system-level translators since they work above the guest OS and have no knowledge of its disk cache and file system.

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 [vdV15] 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 [Cor14], 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.

6.3 Closing remarks

It appears that dynamic binary translation can be competitive with hardware-level backward compatibility in supporting legacy ISAs. In view of this, and the cost both in silicon area and verification effort in providing the hardware, dynamic binary translation may be the way to provide ISA migration in the future.

Bibliography

[AA06] 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 2006*, pages 2–13. ACM, 2006.

[App06] Apple. Apple — Rosetta, 2006. [Archived at `http://web.archive.org/web/20060113055505/http://www.apple.com/rosetta/`].

[ARM13] ARM. big.LITTLE technology: The future of mobile, 2013. (Visited on 13/07/2016).

[ARM16] ARM. *Cortex-A57 Software Optimization Guide*, 2016.

[BBTV15] Darrell Boggs, Gary Brown, Nathan Tuck, and K. S. Venkatraman. Denver: Nvidia's first 64-bit ARM processor. *IEEE Micro*, 35(2):46–55, 2015.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In Monica S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver,*

Britith Columbia, Canada, June 18-21, 2000, pages 1–12. ACM, 2000.

[BDE+03] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel 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, San Diego, CA, USA, December 3-5, 2003*, pages 191–204. ACM/IEEE Computer Society, 2003.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Timothy L. Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In Michael L. Scott and Larry L. Peterson, editors, *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, pages 164–177. ACM, 2003.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 41–46. USENIX, 2005.

[BFT10] Igor Böhm, Björn Franke, and Nigel P. Topham. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In Fadi J. Kurdahi and Jarmo Takala, editors, *Proceedings of the 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2010), Samos, Greece, July 19-22, 2010*, pages 1–10. IEEE, 2010.

[BGA03] Derek Bruening, Timothy Garnett, and Saman P. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003)*, pages 265–275. IEEE Computer Society, 2003.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[BK08] Derek Bruening and Vladimir Kiriansky. Process-shared and persistent code caches. In David Gregg, Vikram S. Adve, and Brian N. Bershad, editors, *Proceedings of the 4th International Conference on Virtual Execution Environments, VEE 2008, Seattle, WA, USA, March 5-7, 2008*, pages 61–70. ACM, 2008.

[BKGB06] Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji. Thread-shared software code caches. In *Fourth IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2006)*, pages 28–38. IEEE Computer Society, 2006.

[BL07] Prashanth P. Bungale and Chi-Keung Luk. Pinos: a programmable framework for whole-system dynamic instrumentation. In Chandra Krintz, Steven Hand, and David Tarditi, editors, *Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007, San Diego, California, USA, June 13-15, 2007*, pages 137–147. ACM, 2007.

[BRG+06] Rishi Bhardwaj, Phillip Reames, Russell Greenspan, Vijay Srinivas Nori, and Ercan Ucan. A choices hypervisor on the ARM architecture. *Department of Computer Science, University of Illinois at Urbana-Champaign*, 11, 2006.

[Bru04] Derek Lane Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[BvKK+11] Igor Böhm, Tobias J. K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel P. Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 74–85. ACM, 2011.

[BZA12] Derek Bruening, Qin Zhao, and Saman P. Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th International Conference on Virtual Execution Environments, VEE 2012*, pages 133–144. ACM, 2012.

[CHH+98] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S Bharadwaj Yadavalli, and John Yates. FX! 32: A profile-directed binary translator. *IEEE Micro*, (2):56–64, 1998.

[CJE+12] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narváez, and Joel S. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, pages 213–224. IEEE Computer Society, 2012.

[CLU02] Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout: A retargetable dynamic binary translation framework. 2002.

[CMR07] Matthew Chapman, Daniel J Magenheimer, and Parthasarathy Ranganathan. Magixen: Combining binary translation and virtualization. Technical report, Technical Report HPL-2007-77, Hewlett-Packard Laboratories, 2007.

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

[Cor14] Jonathan Corbet. Supporting filesystems in persistent memory, 2014.

[CWH+14] Chao-Rui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. Efficient memory virtualization for cross-isa system mode emulation. In Martin Hirzel, Erez Petrank, and Dan Tsafrir, editors, *10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14, Salt Lake City, UT, USA, March 01 - 02, 2014*, pages 117–128. ACM, 2014.

[DA06] Dinakar Dhurjati and Vikram S. Adve. Efficiently detecting all dangling pointer uses in production servers. In *2006 International Conference on Dependable Systems and Networks (DSN 2006), 25-28 June 2006, Philadelphia, Pennsylvania, USA, Proceedings*, pages 269–280. IEEE Computer Society, 2006.

[DB00] Evelyn Duesterwald and Vasanth 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.

[DGB+03] James C. Dehnert, Brian Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim 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.

[dGGL16] Amanieu d'Antras, Cosmin Gorgovan, Jim D. Garside, and Mikel Luján. Optimizing indirect branches in dynamic binary translators. *ACM TACO*, 13(1):7, 2016.

[DLC+12] 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 Ottawa Linux Symposium (OLS)*, pages 93–107. Citeseer, 2012.

[DN14] Christoffer Dall and Jason Nieh. KVM/ARM: the design and implementation of the linux ARM hypervisor. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 333–348. ACM, 2014.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, 21(9):948–960, 1972.

[Gri11] Richard Grisenthwaite. ARMv8 Technology Preview, 2011.

[HDBZ15] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. Optimizing binary translation of dynamically generated code. In

Kunle Olukotun, Aaron Smith, Robert Hundt, and Jason Mars, editors, *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, pages 68–78. IEEE Computer Society, 2015.

[HH97] Raymond J. Hookway and Mark A. Herdeg. DIGITAL fx!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1), 1997.

[HHY+12] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In Carol Eidt, Anne M. Holler, Uma Srinivasan, and Saman P. Amarasinghe, editors, *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*, pages 104–113. ACM, 2012.

[HK06] Kim M. 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 2006*, pages 261–270. ACM, 2006.

[HLC09] Kim M. Hazelwood, Greg Lueck, and Robert Cohn. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In Hillel Kolodner and Guy L. Steele Jr., editors, *Proceedings of the 8th International Symposium on Memory Management, ISMM 2009*, pages 20–29. ACM, 2009.

[HM80] R. Nigel Horspool and Nenad Marovac. An approach to the problem of detranslation of computer programs. *Computer Journal*, 23(3):223–229, 1980.

[HWH+07] Jason Hiser, Daniel W. Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Fifth International Symposium on Code Generation and Optimization (CGO 2007)*, pages 61–73. IEEE Computer Society, 2007.

[JYHC14a] Ning Jia, Chun Yang, Yu He, and Xu Cheng. DTT: program structure-aware indirect branch optimization via direct-tpc-table in DBT system. In *Computing Frontiers Conference, CF'14*, pages 12:1–12:10. ACM, 2014.

[JYHC14b] Ning Jia, Chun Yang, Yu He, and Xu Cheng. SPTU: improving dynamic binary translation through software prediction with target updating. In *International Conference on Systems and Storage, SYSTOR 2014*, pages 2:1–2:12. ACM, 2014.

[JYW+13] Ning Jia, Chun Yang, Jing Wang, Dong Tong, and Keyi Wang. SPIRE: improving dynamic binary translation through spc-indexed indirect branch redirecting. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '13*, pages 1–12. ACM, 2013.

[KS03] Ho-Seop Kim and James E. Smith. Hardware support for control transfers in code caches. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 253–264. ACM/IEEE Computer Society, 2003.

[LA04] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[LCM+05] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim 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.

[LWH11] Jianjun Li, Chenggang Wu, and Wei-Chung Hsu. Dynamic register promotion of stack variables. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*, pages 21–31. IEEE Computer Society, 2011.

[MBC+09] Ryan W. Moore, José Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason 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 2009*, pages 147–156. ACM, 2009.

[MCGP07] Tipp Moseley, Daniel A. Connors, Dirk Grunwald, and Ramesh Peri. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 4th Conference on Computing Frontiers*, pages 143–152. ACM, 2007.

[Moo65] Gordon Moore. Cramming more components onto integrated circuits, electronics,(38) 8, 1965.

[NMO11] Takashi Nakamura, Satoshi Miki, and Shuichi Oikawa. Automatic vectorization by runtime binary translation. In *Second International Conference on Networking and Computing, ICNC 2011, November 30 - December 2, 2011, Osaka, Japan*, pages 87–94. IEEE Computer Society, 2011.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 89–100. ACM, 2007.

[NT14] Ben Niu and Gang Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1317–1328. ACM, 2014.

[NVI13] NVIDIA. Nvidia tegra 4 family cpu architecture. Technical report, NVIDIA, 2013.

[Pat14] Anup Patel. Xvisor: eXtensible Versatile hypervISOR, 2014.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[PG10] Mathias Payer and Thomas R. Gross. Generating low-overhead dynamic binary translators. In *Proceedings of of SYSTOR 2010: The 3rd Annual Haifa Experimental Systems Conference*. ACM, 2010.

[PKG13] Mathias Payer, Enrico Kravina, and Thomas R. Gross. Lightweight memory tracing. In Andrew Birrell and Emin Gün Sirer, editors, *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 115–126. USENIX Association, 2013.

[PKR+13] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. Formal virtualization requirements for the ARM architecture. *Journal of Systems Architecture - Embedded Systems Design*, 59(3):144–154, 2013.

[PVC01] Michael Paleczny, Christopher A. Vick, and Cliff Click. The java hotspot server compiler. In Saul Wold, editor, *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*. USENIX, 2001.

[RCCS07] Vijay Janapa Reddi, Dan Connors, Robert Cohn, and Michael D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *Fifth International Symposium on Code Generation and Optimization (CGO 2007), 11-14 March 2007, San Jose, California, USA*, pages 74–88. IEEE Computer Society, 2007.

[Sea01] David Seal. *ARM Architecture Reference Manual*. Pearson Education, 2001.

[SIN11] Yukinori Sato, Yasushi Inoguchi, and Tadao 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.

[SKC+04] Kevin Scott, Naveen Kumar, Bruce R. Childers, Jack W. Davidson, and Mary Lou Soffa. Overhead reduction techniques for software dynamic translation. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*. IEEE Computer Society, 2004.

[SN05] Julian Seward and Nicholas 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.

[SSB05] Swaroop Sridhar, Jonathan S. Shapiro, and Prashanth P. Bungale. HDTrans: A low-overhead dynamic translator. In *Proceedings of the 2005 Workshop on Binary Instrumentation and Applications (September 2005), IEEE Computer Society*, 2005.

[SSB07] Swaroop Sridhar, Jonathan S. Shapiro, and Prashanth P. Bungale. Hdtrans: a low-overhead dynamic translator. *SIGARCH Computer Architecture News*, 35(1):135–140, 2007.

[SZP+13] Alexey Smirnov, Mikhail Zhidko, Ying-Shiuan Pan, Po-Jui Tsao, Kuang-Chih Liu, and Tzi-cker Chiueh. Evaluation of a server-grade software-only ARM hypervisor. In *2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*, pages 855–862. IEEE, 2013.

[TKKM14] Xin Tong, Toshihiko Koju, Motohiro Kawahito, and Andreas Moshovos. Optimizing memory translation emulation in full system emulators. *TACO*, 11(4):60:1–60:24, 2014.

[Tra08] Transitive. Transitive, 2008. [Archived at `https://web.archive.org/web/20080914184751/http://www.transitive.com`].

[vdV15] Arjan van de Ven. An introduction to clear containers, 2015.

[Wat08] Jon Watson. Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1, 2008.

[WHK+07] Cheng Wang, Shiliang Hu, Ho-Seop Kim, Sreekumar R. Nair, Mauricio Breternitz Jr., Zhiwei Ying, and Youfeng Wu. StarDBT: An efficient multi-platform dynamic binary translation system. In Lynn Choi, Yunheung Paek, and Sangyeun Cho, editors, *Advances in Computer Systems Architecture, 12th Asia-Pacific Conference, ACSAC 2007, Seoul, Korea, August 23-25, 2007, Proceedings*, volume 4697 of *Lecture Notes in Computer Science*, pages 4–15. Springer, 2007.

[Wol96] Michael Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.

[WYZM16] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. A general persistent code caching framework for dynamic binary translation (DBT). In Ajay Gulati and Hakim Weatherspoon, editors, *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, pages 591–603. USENIX Association, 2016.

[YF08] Efe Yardimci and Michael Franz. Dynamic parallelization and vectorization of binary executables on hierarchical platforms. *J. Instruction-Level Parallelism*, 10, 2008.

[ZKR+11] Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman 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.

[ZT00] Cindy Zheng and Carol L. Thompson. PA-RISC to IA-64: transparent execution, no recompilation. *IEEE Computer*, 33(3):47–52, 2000.