Low Overhead Dynamic Binary Translation on ARM

DOI:
10.1145/3062341.3062371

Document Version
Accepted author manuscript

Published in:
Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017

Low Overhead Dynamic Binary Translation on ARM

Amanieu d'Antras Cosmin Gorgovan Jim Garside Mikel Luján
School of Computer Science, University of Manchester, UK
{bdantras,cgorgovan,jgarside,mikel}@cs.manchester.ac.uk

Abstract

The ARMv8 architecture introduced AArch64, a 64-bit execution mode with a new instruction set, while retaining binary compatibility with previous versions of the ARM architecture through AArch32, a 32-bit execution mode. Most hardware implementations of ARMv8 processors support both AArch32 and AArch64, which comes at a cost in hardware complexity.

We present MAMBO-X64, a dynamic binary translator for Linux which executes 32-bit ARM binaries using only the AArch64 instruction set. We have evaluated the performance of MAMBO-X64 on three existing ARMv8 processors which support both AArch32 and AArch64 instruction sets. The performance was measured by comparing the running time of 32-bit benchmarks running under MAMBO-X64 with the same benchmark running natively. On SPEC CPU2006, we achieve a geometric mean overhead of less than $7.5\,\%$ on in-order Cortex-A53 processors and a performance improvement of $1\,\%$ on out-of-order X-Gene 1 processors.

MAMBO-X64 achieves such low overhead by novel optimizations to map AArch32 floating-point registers to AArch64 registers dynamically, handle overflowing address calculations efficiently, generate traces that harness hardware return address prediction, and handle operating system signals accurately.

CCS Concepts • **Software and its engineering → Just-in-time compilers**

Keywords Binary Translation, ARM

1. Introduction

While ARM has traditionally been a 32-bit architecture, the ARMv8 version of the architecture [14] introduced a new 64-bit execution mode and instruction set, called *AArch64*. Most of the current generation of ARMv8 processors is capable of running existing 32-bit ARM applications directly in *AArch32* mode, but maintaining this support comes at a significant cost in hardware complexity, power usage and development time. For example, Cavium does not include hardware support for AArch32 in their ThunderX processors for this reason.

This paper investigates using *binary translation* to translate an AArch32 program into AArch64 code dynamically, which enables it to run on a processor that only supports the AArch64 instruction set. Binary translation allows a program to be translated transparently, instrumented or modified at the machine code level.

We have developed MAMBO-X64, a Dynamic Binary Translator (DBT) 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.

There are several complex problems that need to be addressed to achieve this objective. For example, while AArch64 has a strictly larger general purpose register bank than AArch32, this is not necessarily the case for the floating-point register bank: AArch32 can, in some situations, exploit register bank aliasing to hold up to 48 floating point values in registers, whereas AArch64 is limited to only 32 floating point registers. Naïvely translating the latter case can generate significant performance degradation.

Load and store instructions in both AArch32 and AArch64 support a similar set of addressing modes. Despite their similarity, however, simply translating the AArch32 addressing modes into their AArch64 equivalent will not always produce correct results; the address width used can affect whether the calculation overflows.

It is widely acknowledged that indirect branch handling is the single biggest source of overhead for DBTs [16, 18] and that trace compilation is an integral part of advanced DBTs for improving performance. However, we show that existing trace generation algorithms interfere with optimizations for handling indirect branches.

Precise handling of operating system signals is challenging in DBTs because they can interrupt program execution at

arbitrary points. In particular, a signal is delivered between two instructions, however instruction boundaries in the translated code may not match those of the original application code.

This paper presents the architecture of MAMBO-X64 (Section 2) and describes the following contributions:

1. An efficient scheme for mapping AArch32 floating-point registers to AArch64 registers dynamically (Section 3);
2. A method for efficiently translating AArch32 load/store addressing modes into AArch64 by using speculation (Section 3.3);
3. A novel trace compilation algorithm that leverages hardware return address prediction to improve performance (Section 4); and
4. A system for delivering operating system signals to translated programs without suffering from race conditions, with minimal performance impact (Section 5).

While the first two contributions are described as implemented for AArch32 to AArch64 translation, they can be generalized to other architectures with similar characteristics. The last two contributions are general and are applicable to all DBTs.

When evaluated on the X-Gene, an ARMv8 system which support both AArch32 and AArch64 in hardware (Section 6), a 32-bit build of SPEC CPU2006 runs on average $1\,\%$ *faster* under MAMBO-X64 compared to running the same 32-bit binary natively on the processor. Particular benchmarks are 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 $19\,\%$. No other DBT has achieved a similar level of performance when translating from a 32-bit to a 64-bit architecture (Section 7).

2. MAMBO-X64 Architecture

MAMBO-X64 is structured as three components:

- **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 OSes. From the binary translator's point of view, system interactions primarily consist of either synchronous traps or asynchronous interrupts. This concept has even been extended to HyperMAMBO-X64 [11], which integrates the binary translator component of MAMBO-X64 into a hypervisor to allow fully transparent translation without the need for any OS modifications.

2.1 Binary Translator

The binary translator works by scanning sequences of AArch32 instructions on demand and converting them into AArch64 *code fragments*, stored in a *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. As fragments are added to the code cache, they are *linked* to each other using branch instructions. In the case of direct branches, this is simply a direct branch instruction to the target fragment, since the branch target is known at translation time.

Indirect branches need to be handled differently because the branch target is only known at execution time and can vary from one execution to the next. This requires its own dynamic translation and can impose a serious runtime penalty. Previous research [16, 18] has shown that indirect branch handling is the biggest performance bottleneck in a DBT. Function returns have been shown to be the most common type of indirect branch [10], which is why MAMBO-X64 uses a novel algorithm called ReTrace to translate these branches in a way that leverages hardware return prediction. The details of this algorithm are described in Section 4.

MAMBO-X64 uses a *thread-shared code cache* model where the same translated code is shared among multiple threads. This model has been shown to scale significantly better than thread-private code caches on multi-threaded applications [7, 15]. Executing code in the code cache does not require any synchronization and MAMBO-X64 is able to perform code translation concurrently in multiple threads. Synchronization is only required when adding or removing a fragment from the code cache, which is a rare operation compared to the execution of translated code.

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.

Section 3 describes the translation process for individual fragments.

2.2 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 $4\,\mathrm{GB}$ '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

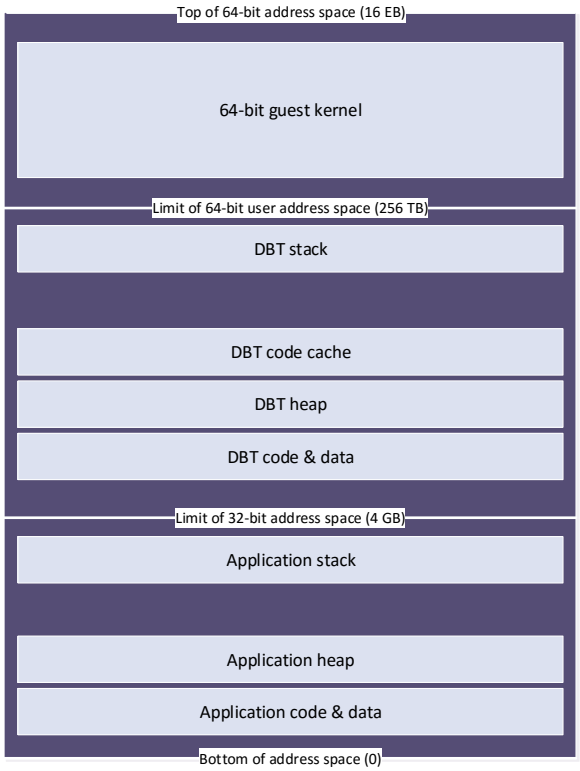


Figure 1: Address space layout of an application running under MAMBO-X64.

restricted to this address space since the original code uses 32-bit memory addresses. This layout, shown in Figure 1, 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[1], 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 and uses a scheme, described in Section 5, which involves fragment unlinking and signal masking to achieve race-free and efficient signal delivery to the application.

3. 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[2]. As instructions are gathered, the DBT will also determine the set of input and output registers and condition flags for each 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. Table 1 shows some examples of instructions translated by MAMBO-X64.

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:

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. Not all such instructions can be translated to AArch64 directly, and may require additional instructions to calculate the 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

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

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

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`

`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 1: Examples of AArch32 instruction sequences translated by MAMBO-X64.

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 $4\,\mathrm{kB}$ 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. The naïve approach would be to load the constant from the code pages of the original program.

3.1 Register Allocation

Table 2 shows the user-visible registers available on AArch32 and AArch64. 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 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 2. 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`.

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
`D0` – `D31`	64-bit floating-point/SIMD registers
`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

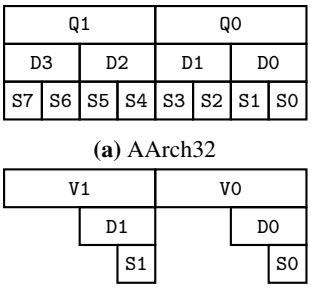
Register	Usage in MAMBO-X64
`X0` – `X14`	Mapped to `R0` – `R14`
`X15` – `X18`	Scratch registers
`X19`	`APSR.Q` flag
`X20`	`APSR.C` and `APSR.V` flags
`X21`	`FPSCR.NZCV` flags
`X22`	Shadow copy of the `FPCR` register
`X23`	`APSR.GE` flags
`X24` – `X27`	Indirect branch hash table parameters
`X28`	Return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

(c) AArch64 register usage in MAMBO-X64

Table 2: Comparison of AArch32 and AArch64 registers and how MAMBO-X64 uses them.

3.2 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 [19],



(a) AArch32

(b) AArch64

Figure 2: Floating-point register aliasing in AArch32 and AArch64.

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 translated 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.

AArch32	AArch64
`LDR R0, [R1]`	`LDR W0, [X1]`
`LDR R0, [R1, #8]`	`LDR W0, [X1, #8]`
`LDR R0, [R1, R2]`	`LDR W0, [X1, X2]`
`LDR R0, [R1, R2, LSL #2]`	`LDR W0, [X1, X2, LSL #2]`

Table 3: Examples of memory addressing modes in AArch32 and AArch64.

3.3 Speculative Address Generation

Load and store instructions in both AArch32 and AArch64 support a similar set of addressing modes[3], of which a few examples are shown in Table 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 instruction `LDR R0, [R1, R2]`: if `R1` has the value `0xffff0000` and `R2` has the value `0x40000` then, 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`. Naïvely translating this instruction to the closest AArch64 equivalent (`LDR W0, [X1, X2]`) will not overflow the 64-bit address width and will instead result in the address `0x100030000`, which is outside the 32-bit address space.

Calculating the target address correctly requires translating this load instruction into two separate instructions:

- `ADD W16, W1, W2`: Performs the address calculation by adding the registers and truncates the result by writing it to a 32-bit register.
- `LDR W0, [X16]`: Performs the actual load using the previously calculated address.

While correct, this approach has two significant downsides: not only does it double the translated code size for what is a very common instruction, which impacts memory overhead and instruction cache locality, it also requires

[3] 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.

a data-dependent addition cycle before each load or store instruction[4].

Note that the same problem also applies to immediate-offset loads and stores: adding an immediate value (up to ± 4095 bytes) from a 32-bit base address can still overflow or underflow the 32-bit address space. We developed two different solutions to this problem, one for immediate-offset addressing and one for register-offset addressing.

Immediate-Offset Addressing The fact that AArch32 limits its immediate offsets for load/store instructions to ± 4095 bytes offers a partial solution to this problem since potential overflows are limited to just $4\,\mathrm{kB}$ 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 $4\,\mathrm{GB}$ 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.

Register-Offset Addressing The previously described approach is not viable for register-offset addressing since each operand can have any 32-bit value. Instead, we developed a new, more general approach 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 (`LDR W0, [X1, W2, SXTW]`). 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 this 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 $2\,\mathrm{GB}$.

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 $12\,\mathrm{GB}$ 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 $\pm 8\,\mathrm{GB}$. Combined with the first operand, this results in such a load/store being able to access any address from $-8\,\mathrm{GB}$ to $12\,\mathrm{GB}$. 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, the load/store instruction address is blacklisted so that it will be translated with a separate `ADD` instruction in the future. The faulting fragment is then invalidated so that it will be re-translated later with the blacklisted instruction taken into account.

This approach allows MAMBO-X64 to handle edge cases that overflow the address calculation correctly and efficiently. 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. 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 [13] from Dynamo [3] 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.

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 ($\pm 1\,\mathrm{MB}$) rather than

[4] On all of the ARM cores that we tested, most address calculations that are part of a load/store instruction do not cost any additional cycles.

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

having to use an intermediate 'trampoline' branch with a longer range.

4.1 Interactions with Hardware-Assisted Function Returns

MAMBO-X64 uses hardware-assisted function returns [10], which 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 degrades the effectiveness of hardware-assisted function returns when used with NET.

Consider the example in Figure 3 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.

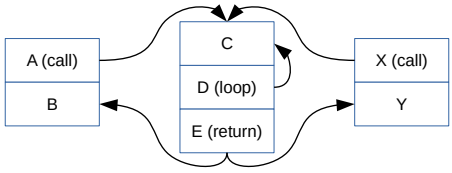
The latter is able to trace across a return instruction by using a *guard*: if the return address is different from the one when the trace was generated then the code will fall back to a hash table lookup to handle the indirect branch. However, this guard can be a big source of branch mispredictions 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 Integration with Hardware Return Prediction

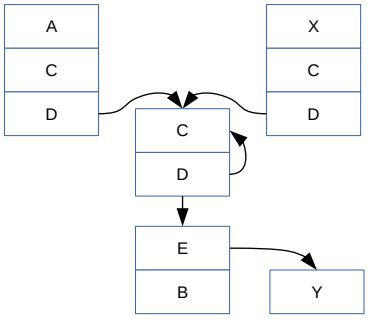
MAMBO-X64 introduces an improved version of NET which is compatible with hardware-assisted function returns, called *return-aware trace generation*, or simply ReTrace. The principle is that traces should not cross function calls or returns. This is implemented by adding the following rules to NET:

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

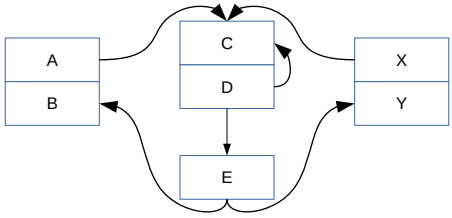
Going back to the example in Figure 3, this new algorithm will also create the CD trace first, as it is the inner loop. However the trace starting at A will be different: trace recording is paused after the call instruction and the trace is pushed onto the stack of active traces. Control is then passed to the inner loop trace. When the inner loop exits, it will



(a) Original control flow



(b) Translation result with NET



(c) Translation result with ReTrace

Figure 3: 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).

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 6.3.

5. 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.

5.1 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.

5.2 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 we have only managed to observe it in specially-crafted test programs.

5.3 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.

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

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

Figure 4: Code to atomically execute a system call if there are no pending signals. Atomicity is ensured by having a signal handler rewind the program counter to `atomic_begin` if a signal occurs between `atomic_begin` and `atomic_end`.

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: if a signal is waiting to be 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.

6. Evaluation

We evaluate the overall performance of MAMBO-X64 (Section 6.1) and how the different novel techniques contribute to

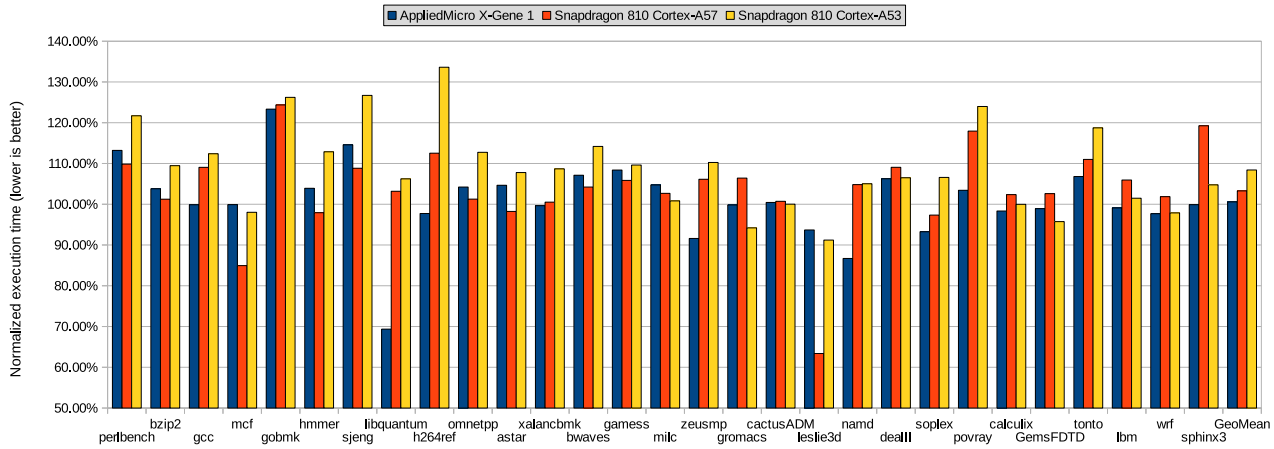


Figure 5: Performance of SPEC CPU2006 on MAMBO-X64 on different processors. Performance numbers are relative to the benchmark running natively in 32-bit mode on the same processor.

its low performance overhead (Section 6.3) using the SPEC CPU2006 [24] and PARSEC [5] 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`.

Two ARMv8 systems were used for the evaluation. The first 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.

The second system is an Intrinsyc Dragonboard 810 with a Qualcomm Snapdragon 810 processor. The processor is a heterogeneous multicore configuration (big.LITTLE) 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.

We use `taskset` on the Snapdragon system to force benchmarks to run on either the A53 cluster or the A57 cluster.

6.1 Overall Performance

Figure 5 shows the performance of SPEC CPU2006 when running under MAMBO-X64 on the two test systems.

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 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 [3] 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 microarchitectures. We have determined that this is due to a peculiarity in the floating-point pipeline of the Cortex-A57 which only affects execution in AArch64 mode [2]; 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

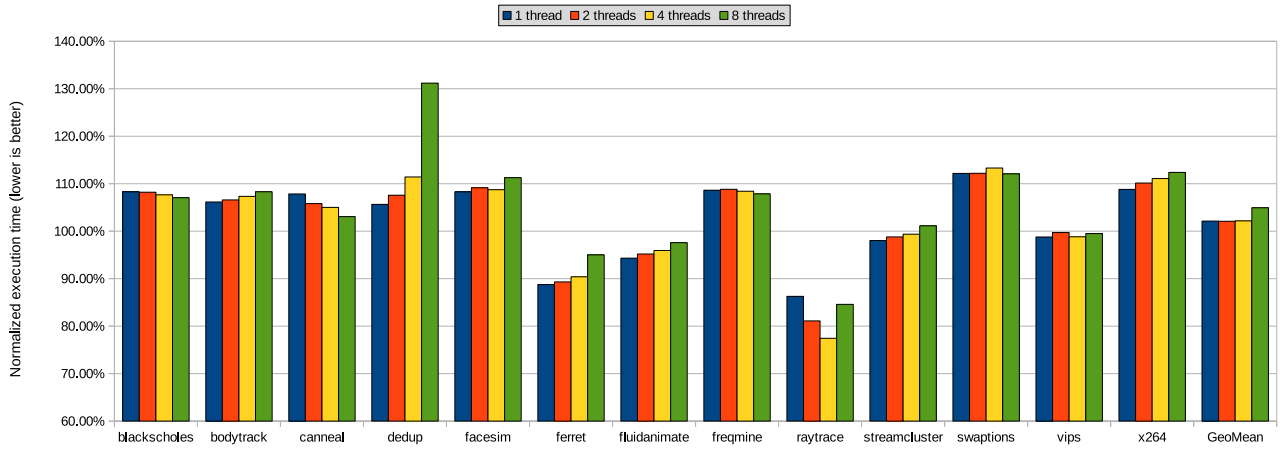


Figure 6: Performance of the PARSEC benchmarks running on MAMBO-X64 on the X-Gene system with different numbers of threads. Performance numbers are relative to the benchmark running natively in 32-bit mode with the same number of threads.

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.

We found that SPEC CPU under MAMBO-X64 has a geometric mean memory overhead of $4.24\,\%$. The benchmark on which MAMBO-X64 has the highest memory overhead is *gcc*, with an overhead of $31.9\,\mathrm{MB}$, and the one with the lowest memory overhead was *libquantum* with an overhead of $1.58\,\mathrm{MB}$.

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 $\pm 2\,\%$ between runs, whereas the X-Gene system has much more stable results with a typical variation of only about $0.1\,\%$.

6.2 Multi-Threaded Performance

Figure 6 shows the performance of the PARSEC multi-threaded benchmark suite when running under MAMBO-X64 on the X-Gene system.

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.

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 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.

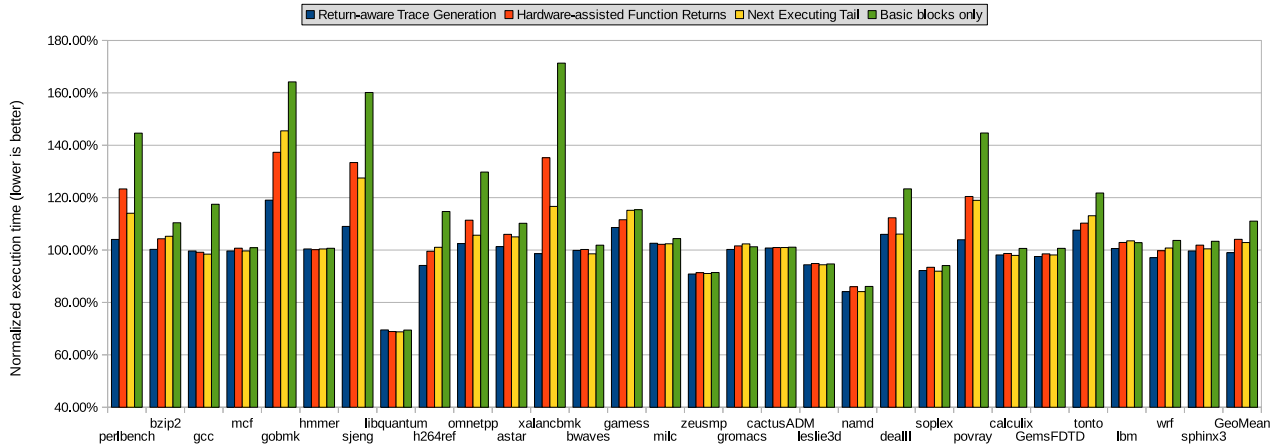
6.3 Detailed Analysis

We analyzed the performance impact of the various optimizations presented in this paper with SPEC CPU2006 on the X-Gene system.

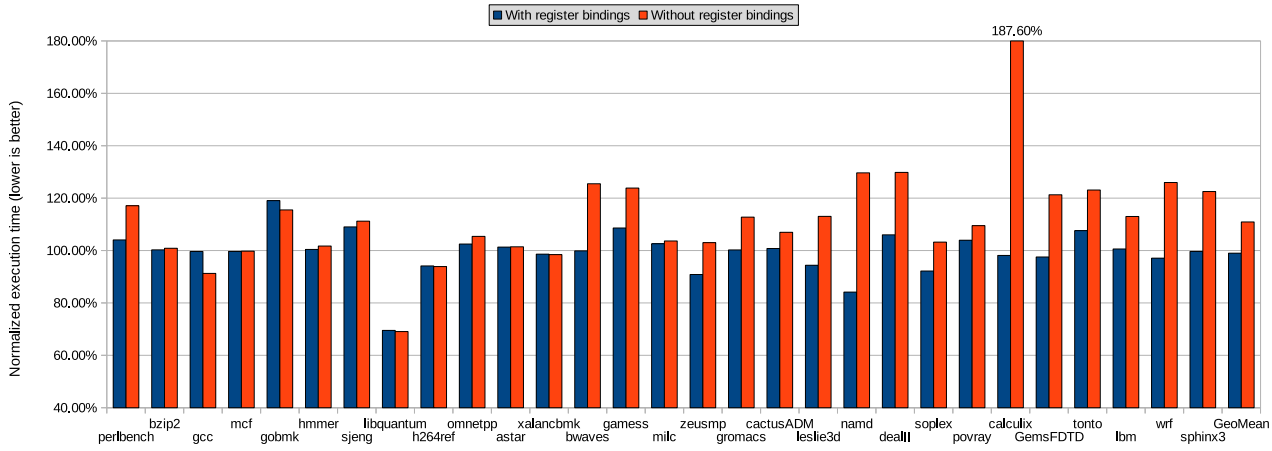
ReTrace We also investigated the effects of trace generation and function call handling on the performance of translated code. Figure 7a shows the performance of MAMBO-X64 in four configurations. *Return-aware trace generation* combines NET with hardware-assisted function returns using the ReTrace algorithm. *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 NET does not preserve the call structure. The *basic blocks only* configuration translates only 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 by $1.0\,\%$.

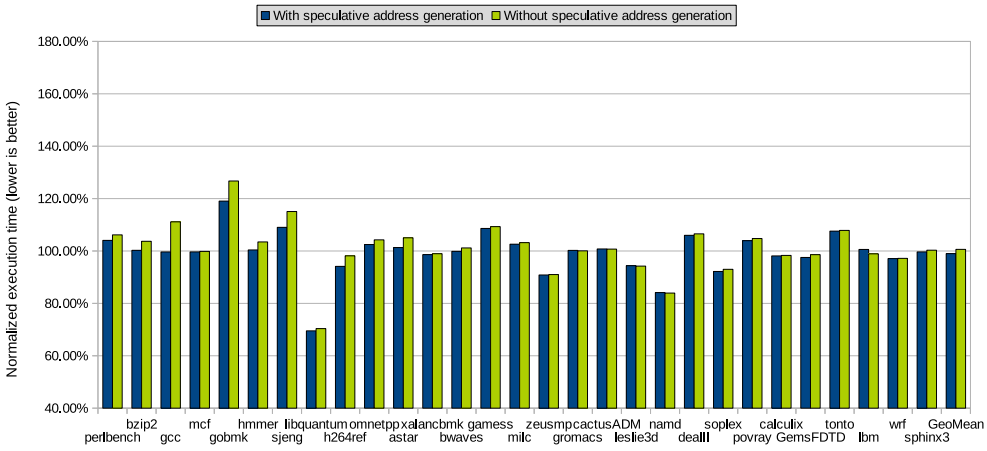
Register Bindings We measured the effect of inter-fragment register allocation on performance by disabling dynamic register bindings in MAMBO-X64. This is done by forcing all floating-point register values to be written back to memory before any fragment exits and reloading those values from memory as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



(a) ReTrace



(b) Register bindings



(c) Speculative address generation

Figure 7: Performance of SPEC CPU2006 running on MAMBO-X64 on the X-Gene system with different optimizations. Performance numbers are relative to the benchmark running natively in 32-bit mode.

The results are shown in Figure 7b. 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 benchmarks, which is mainly due to the extra memory traffic in the benchmark inner loops. These results show that register bindings offer a geometric mean performance improvement of $21.0\,\%$ in floating-point benchmarks.

Speculative Address Generation We tested the impact of speculative address generation in MAMBO-X64 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 7c 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.

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 $0.4\,\%$.

7. Related Work

Binary translation has previously been used successfully to assist architecture transitions: The most well known is Rosetta [1], which was used by Apple to transit their platform from PowerPC to x86. IA-32 EL [4] and HP Aries [26] both supported the transition to the IA-64 architecture from x86 and PA-RISC respectively. FX!32 [9, 17] was similarly used to help migrate x86 applications to the Alpha architecture. Binary translation has also been used to allow executing code from existing instruction sets on a VLIW core, such as Nvidia Denver (ARM on VLIW) [6] and Transmeta Crusoe (x86 on VLIW) [12].

StarDBT [25] and memTrace [22] are two DBTs which translate from x86 to x86-64 for the purpose of performing dynamic program instrumentation. These take advantage of the larger address space to lower the overhead of instrumentation code. However they still suffer from relatively high performance overheads of around $12\,\%$ and $17\,\%$ respectively on SPEC CPU2006 compared to native execution. Additionally, both of these systems use thread private code caches which do not scale well for multi-threaded applications.

DynamoRIO [8], Pin [19], Valgrind [20], HDTrans [23] and fastBT [21] are DBTs designed for program instrumentation which have the same host and source architectures. The last two do not perform register state recovery when translating signals, and thus may cause some applications to malfunction. The remaining DBTs described in this section make use of signal queues in the DBT, which can lead to certain race conditions with the kernel signal queues. MAMBO-X64 does not suffer from these issues.

8. Conclusions

The results presented, without having modified hardware, constitute the best DBT results published so far when moving from a 32-bit to a 64-bit architecture. For example, MAMBO-X64 has achieved a geometric mean $1\,\%$ *speedup* when running SPEC CPU2006 on an X-Gene 1 processor.

A significant 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.

We have 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. Our effective handling of floating-point registers is the optimization with the highest impact on floating-point benchmarks.

Finally, we introduced a novel signal handling scheme 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.

Dynamic binary translation has been used in the past to support architecture transition. Whilst providing functionality, there has always been a significant performance overhead. Our results show 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.

Acknowledgments

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

References

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

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

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

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

[5] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

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

[7] D. Bruening, V. Kiriansky, T. Garnett, and S. 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. doi: 10.1109/CGO.2006.36.

[8] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

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

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

[11] A. d'Antras, C. Gorgovan, J. Garside, J. Goodacre, and M. Luján. HyperMAMBO-X64: Using virtualization to support high-performance transparent binary translation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2017*, pages 228–241. ACM, 2017. doi: 10.1145/3050748.3050756.

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

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

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

[15] K. M. Hazelwood, G. Lueck, and R. Cohn. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *Proceedings of the 8th International Symposium on Memory Management, ISMM 2009*, pages 20–29. ACM, 2009. doi: 10.1145/1542431.1542435.

[16] J. Hiser, D. W. Williams, W. Hu, J. W. Davidson, J. Mars, and B. 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. doi: 10.1109/CGO.2007.10.

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

[18] H. Kim and J. 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. doi: 10.1109/MICRO.2003.1253200.

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

[20] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100. ACM, 2007. doi: 10.1145/1250734.1250746.

[21] M. Payer and T. R. Gross. Generating low-overhead dynamic binary translators. In *Proceedings of of SYSTOR 2010: The 3rd Annual Haifa Experimental Systems Conference*. ACM, 2010. doi: 10.1145/1815695.1815724.

[22] M. Payer, E. Kravina, and T. R. Gross. Lightweight memory tracing. In *2013 USENIX Annual Technical Conference*, pages 115–126. USENIX Association, 2013. URL `https://www.usenix.org/conference/atc13/technical-sessions/presentation/payer`.

[23] S. Sridhar, J. S. Shapiro, and P. P. Bungale. Hdtrans: a low-overhead dynamic translator. *SIGARCH Computer Architecture News*, 35(1):135–140, 2007. doi: 10.1145/1241601.1241602.

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

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

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