MAMBO: A Low-Overhead Dynamic Binary Modification Tool for ARM

COSMIN GORGOVAN, AMANIEU D'ANTRAS, and MIKEL LUJÁN,

School of Computer Science, University of Manchester

As the ARM architecture expands beyond its traditional embedded domain, there is a growing interest in dynamic binary modification (DBM) tools for general-purpose multicore processors that are part of the ARM family. Existing DBM tools for ARM suffer from introducing large overheads in the execution of applications. The specific questions that this article addresses are (i) how to develop such DBM tools for the ARM architecture and (ii) whether new optimisations are plausible and needed.

We describe the general design of MAMBO, a new DBM tool for ARM, which we release together with this publication, and introduce novel optimisations to handle indirect branches. In addition, we explore scenarios in which it may be possible to relax the transparency offered by DBM tools to allow extra optimisations to be applied. These scenarios arise from analysing the most typical usages: for example, application binaries without handcrafted assembly.

The performance evaluation shows that MAMBO introduces small overheads for SPEC CPU2006 and PARSEC 3.0 when comparing with the execution times of the unmodified programs: a geometric mean overhead of 28% on a Cortex-A9 and of 34% on a Cortex-A15 for CPU2006, and between 27% and 32%, depending on the number of threads, for PARSEC on a Cortex-A15.

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

Additional Key Words and Phrases: Dynamic binary modification, software code cache, indirect branch

ACM Reference Format:

Cosmin Gorgovan, Amanieu d'Antras, and Mikel Luján. 2016. MAMBO: A low-overhead dynamic binary modification tool for ARM. *ACM Trans. Archit. Code Optim.* 13, 1, Article 14 (April 2016), 26 pages.

DOI: <http://dx.doi.org/10.1145/2896451>

1. INTRODUCTION

Dynamic Binary Modification (DBM) is a technique for modifying applications transparently while they are executed, working at the level of native machine code. DBM has numerous applications, some of the more common being dynamic instrumentation [Moseley et al. 2007; Seward and Nethercote 2005], program analysis [Sato et al. 2011; Zhao et al. 2011], virtualisation [Adams and Agesen 2006; Watson 2008] and dynamic translation [Bellard 2005; Dehnert et al. 2003; Boggs et al. 2015].

The ARM architecture, once almost exclusively found in embedded systems, is growing in adoption for general-purpose computing; however, most DBM tools and research have focused on the x86 architecture. This has resulted in the performance of DBM tools for ARM lagging behind; see Pin [Luk et al. 2005] or DynamoRIO [Bruening 2004]

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

Authors' addresses: C. Gorgovan, A. d'Antras, and M. Luján, School of Computer Science, The University of Manchester, Oxford Road, Manchester, UK, M13 9PL; emails: {cgorgovan, bdantras, mikel}@cs.man.ac.uk. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted.

2016 Copyright is held by the owner/author(s).

ACM 1544-3566/2016/04-ART14 \$15.00

DOI: <http://dx.doi.org/10.1145/2896451>

This work is licensed under a Creative Commons Attribution International 4.0 License.

on x86/x86-64. For example, Valgrind serialises multithreaded execution on ARM and x86/x86-64. The specific questions that this article addresses are (i) how to develop such DBM tools for the ARM architecture and (ii) whether new optimisations are plausible and needed. In the context of this article, we use the generic terms *ARM* and *ARM architecture* to refer to the 32-bit version of the ARM architecture.

We describe the general design of MAMBO, a new DBM tool for ARM, and introduce novel optimisations to handle indirect branches. MAMBO, like most other DBM tools, runs in the same process with the application it modifies and controls its execution by *scanning and translating* all code before execution. MAMBO includes an ELF loader, which is used to load the application being modified; it supports loading of both dynamically and statically linked executables. MAMBO uses thread-private code caches, which allow multithreaded code scanning and execution with no synchronisation.

In addition, we explore scenarios in which it may be possible to relax the transparency offered by DBM tools to allow extra optimisations to be applied. These scenarios arise from analysing the most typical usages: for example, application binaries generated in their entirety by a compiler. In this context, transparency refers to any changes compared to ‘native’ execution, which can be observed by the guest application. There are various *types* (e.g., library, heap, stack) and *degrees* of transparency.

MAMBO is designed for *behavioural transparency*, meaning that it only implements the types and degrees of transparency required to correctly execute typical workloads, but it does not aim to achieve *full transparency*. We define typical workloads as applications that follow the platform ABI, do not depend on undefined behaviour (as described by the ARM architectural manual [ARM 2015]), and that use standard system libraries.

All systems and techniques described in this article have been fully implemented in MAMBO, which is currently capable of running many unmodified GNU/Linux applications. MAMBO is publicly available for download from <https://github.com/bee-hive-lab/mambo>. Indirect branches are control flow instructions with targets not known at translation time. Looking up the target of indirect-branch instructions at runtime is the major source of overhead for DBM systems [Kim and Smith 2003]. Novel optimisations are presented for two types of indirect-branch instructions: function returns, used at the end of functions to return control to the caller, and table branches, used to implement switch statements.

The performance evaluation shows that MAMBO introduces small overheads for SPEC CPU2006 [Henning 2006] when comparing with the execution times of the unmodified programs; an average (geometric mean) overhead of 28% on a Cortex-A9 (a dual-issue, out-of-order superscalar processor with 8 to 11 pipeline stages) and of 34% on a Cortex-A15 (a triple-issue, out-of-order superscalar processor with 15 to 24 pipeline stages). For PARSEC 3.0 [Bienia 2011], the overhead is even lower, between 27% and 32% on a Cortex-A15, depending on the number of threads.

The article is organised as follows. Section 2 contains a detailed description of MAMBO and a discussion of our observations regarding DBM transparency. Sections 3 and 4 present the linking schemes used by MAMBO for indirect and direct branches, respectively. Section 5 is the performance evaluation. Section 6 discusses related work and the differences between MAMBO and other DBM systems. Section 7 presents our conclusions.

1.1. Contributions

The following are the contributions of this article:

- We describe the overall architecture of MAMBO, a low-overhead DBM system for ARM.

Fig. 1. ARM registers.

- We propose *behavioural transparency*, which relaxes the transparency requirements for DBM systems in order to minimise implementation complexity and improve performance.
- We describe a novel return address prediction scheme that trades off guaranteed transparency for improved performance.
- We describe an improvement of the fastBT [Payer and Gross 2010] table branch linking scheme, which is implemented more efficiently on ARM and reduces the code cache space requirements on all architectures.
- We describe how to implement and configure several established DBM optimisations for ARM.
- We evaluate the performance of MAMBO on ARM Cortex-A9 and Cortex-A15 systems and the impact of individual optimisations on overall DBM performance.
- We compare MAMBO against two other DBM systems, Valgrind and QEMU, showing that MAMBO is, on average, 2.8 times faster than Valgrind and 14.9 times faster than QEMU when running SPEC CPU2006 on a Cortex-A15.

2. SYSTEM OVERVIEW

We have implemented a new high-performance and multicore-scalable DBM platform for researching optimisations on modern ARM hardware, named MAMBO. To the best of our knowledge, it achieves lower overhead than any other reported results on ARM. It makes use both of optimisation techniques previously published in the literature and of novel optimisations that we have developed. MAMBO was designed to be able to run all applications following the ARM ABI. It is currently capable of running a wide range of applications, including the SPEC CPU2000 and CPU2006 benchmark suites, the PARSEC multithreaded benchmark suite, and many unmodified GNU/Linux applications, including large applications such as LibreOffice 4.2 and GIMP 2.8. One of the priorities in developing MAMBO was to keep its code base small to allow researchers to easily understand and modify it. The current version is implemented in fewer than 10,000 lines of code.

ARM is a load/store architecture. Figure 1 shows the organisation of ARM registers. There are 15 32-bit general-purpose registers and a Program Counter (PC) register, which can be read and written by many of the general-purpose instructions. By convention, register R14 is used to store the function return address, called the Link Register (LR); register R13 is used as the Stack Pointer (SP). In addition, an optional floating-point extension (called VFP) uses dedicated double-precision 64-bit registers, which can also be accessed as 32-bit single-precision registers. An optional SIMD extension

(commonly called NEON) shares the 64-bit registers of the VFP, while also being able to access pairs of 64-bit registers as 128-bit registers.

A peculiarity of the ARM architecture is that it supports two different instruction sets:

- The ARM instruction set, which uses a fixed 32-bit instruction word length, supports conditional execution of most instructions, and has few restrictions on the use of the PC register as an operand.
- The Thumb instruction set, which was designed to improve code density, uses 16-bit instruction words to implement a subset of the functionality provided by the ARM instruction set. Thumb-2 is an extension of Thumb, which adds 32-bit instruction words to implement a larger subset of the ARM instruction set. The extended instruction set will be generically referred to as *Thumb* in this document.

MAMBO currently supports most of the ARMv7-A architecture and the 32-bit execution state in ARMv8 (AArch32), including the ARM and Thumb instruction sets and the optional VFP and NEON extensions. Support for various instructions is improved as they were encountered in the applications run under MAMBO. At the time of writing this article, MAMBO supports 345 Thumb instructions and 130 ARM instructions. The ARMv8 architecture manual [ARM 2015] defines 293 AArch32 instructions. However, these numbers cannot be compared directly because instructions are defined differently in the two contexts. For example, MAMBO handles a 16-bit and a 32-bit version of an instruction separately, while the manual counts them as a single instruction. Similarly, MAMBO handles small groups of VFP or NEON instructions that do not access the general-purpose registers and that have similar encodings as single generalised instructions. The Jazelle extension and ThumbEE instruction set are not supported. However, both have been deprecated. We have not encountered any GNU/Linux application using these modes.

MAMBO, like most other DBM tools, runs in the same address space as the application it is modifying and controls its execution by *scanning* and *translating* all of the application machine code before it is executed. MAMBO-translated code is generated using the same instruction set as its source code, which minimises the complexity of the translation logic. While the assembly language syntax for the ARM and Thumb instruction sets is unified, the machine code encoding is completely different, so two different sets of instruction decoders and encoders are used.

Because ARM and Thumb code is commonly intermixed in applications, addressing code poses a challenge: simply using a pointer to an address is not sufficient; the ISA must also be specified. This issue is solved in the instruction set by enforcing halfword or word alignment for all instructions and using the least significant bit of the address, passed to *interworking* branch instructions, to select the instruction set. The same approach is used by MAMBO. Therefore, all code pointers handled by MAMBO use the LSB to encode the instructions set: the bit is set for Thumb and cleared for ARM. The advantage of this approach is that the internal pointers used by MAMBO can be directly passed to interworking branch instructions. On the other hand, addresses used by noninterworking branch instructions in the application must be patched before being passed to MAMBO subsystems, because the value of their LSB is not guaranteed to be correct.

2.1. Scratch Space

The translated code produced by DBM systems for execution from the code cache often uses additional variables. For example, when translating an instruction taking the PC as an input, the value of the application PC must be loaded. Because ARM is a load/store architecture, these additional values must be loaded in registers. While in

Fig. 2. Translation using scratch registers for an instruction that accesses the stack.

some cases it might be possible to use dead registers, that is not always the case. When dead registers are not available, values from some of the live registers must be spilled to memory. This poses a particular challenge on ARM, because the range for immediate offsets for store instructions is only $\pm 4\text{KiB}$ from the base register. When no dead registers are available, only the PC could be used as a base register, meaning that scratch space for spilling registers would have to be reserved at least every 8KiB inside the code cache. Additionally, PC-relative stores are only allowed in ARM mode. This solution has the disadvantages of high overhead from mode switches between Thumb and ARM mode, it creates additional challenges for code-cache allocations, and it requires always mapping the code cache with read/write/execute permissions, with implications for the security of the DBM system.

An alternative approach is to steal a register from the application and use it exclusively as a pointer to scratch space. However, this approach requires being able to rewrite all application instructions to use different registers. To keep complexity low, we prefer to modify as few instructions as possible; therefore, this option was discarded.

It is also possible to use coprocessor registers as scratch space, if they are known to be unused. On ARM, the read/write thread ID register (TPIDRURW) meets this condition on GNU/Linux. Our evaluation showed that the latency to access this register is high and that, on average, the first approach of using scratch space inside the code cache is faster.

Unlike other platforms, the ARM ABI prohibits applications from storing valid data on the stack above the stack pointer. This allows safe use of the application stack for scratch space, for applications following the ABI. However, no persistent MAMBO data can be left on the stack, because it would add an offset to stack accesses from the application. Furthermore, care must be taken to fix up stack offsets when generating the translation for instructions that use the stack themselves. Figure 2 shows an example of such a translation, which must first reserve stack space for the value pushed by the application before spilling a register.

2.2. Executable Loader

MAMBO includes an ELF loader that is used to load the application being run; it supports loading of both dynamically and statically linked executables. As opposed to other methods of attaching a DBM tool to a process, such as using the LD_PRELOAD mechanism, this ensures that no application code executes before MAMBO takes over.

2.3. Code Cache

The code scanner works on short single-entry, single-exit units called *basic blocks*. To amortise the cost of scanning, the generated code is stored in a *code cache*. MAMBO uses thread-private code caches, which allow scanning and execution from multiple threads with no synchronisation.

Figure 3 shows the data structures created by MAMBO for an example basic block. Each thread-private code cache consists of a number of data structures:

Fig. 3. MAMBO data structures for an example basic block.

- a set of fixed-size blocks that hold translated basic blocks;
- metadata unique to each basic block;
- a hash table that maps application addresses to code-cache addresses; and
- meta-data used by the basic block allocator.

The first version of MAMBO focuses on what is feasible without using *traces* (also known as *superblocks*), which are single-entry, multiple-exit units built by merging multiple basic blocks on the hot execution path [Bala et al. 2000; Kim and Smith 2003].

2.4. Code Scanner

The code scanner reads instructions from the source application, applies any requested modifications and outputs position-independent code that can be executed from the code cache. MAMBO has two code scanners: one for the ARM instruction set and one for the Thumb instruction set. Each of these scanners outputs code using the same instruction set as its input, which allows many types of instructions to be copied into the code cache unchanged. Both code scanners work in a single pass and manipulate native code directly, without using an intermediate representation, which enables fast code scanning and translation, minimising application startup overhead.

A *code scanner* consists of a loop that reads, decodes, and translates one instruction at a time. The type of the instruction is compared to a list of instruction types that need to be translated. The translated instructions include instructions that make use of the PC register (e.g., PC-relative memory operations, data-processing operations that use the PC as an input or output) and explicit control flow instructions (e.g., branch, branch-and-link). The instructions whose type is in this list are passed to translation routines, while other instructions are copied unmodified to the code cache. The scanner stops after translating the first control flow instruction in each block, which ensures that basic blocks have a single exit point.

2.5. System Call Interception

All system call instructions are translated into calls to an interception routine, which allows MAMBO to modify the arguments and return values of system calls or even to completely replace it with a different operation. This mechanism is used to handle multithreading, signals, and to detect events such as thread and application exit.

2.6. Plugins

MAMBO does not apply any behavioural changes to the application; it only applies the transformations required to efficiently run applications from the code cache. Additional modifications can be performed through the plugin API, which allows plugins to

inspect and modify the instruction stream, and to observe the same events available to MAMBO, for example, system calls and application exit. The plugin API is used to implement tools for dynamic instrumentation, program analysis, and so on, on top of MAMBO.

An example plugin which implements a dynamic execution counter is provided in Appendix A, however a detailed description of the plugin API is outside the scope of this article.

2.7. Transparency

In the context of DBM systems, transparency refers to any changes compared to native execution that can be observed by the guest application. There are various *types* (e.g., library, heap, and stack) and *degrees* (e.g., behavioural, full) of transparency.

Some types of transparency can be fully supported with minimal implementation complexity and overhead. For example, library transparency can be implemented by statically linking any libraries used by the DBM tool and ensuring that no data is shared between the copy used by the DBM tool and any copies that might be used by the application. Other types of transparency are impractical to fully support or would have a high overhead, for example, timing transparency.

Many DBM tools are designed to support the best degree of transparency for as many types of transparency as possible. However, this goal can directly conflict with the goal of minimising execution overhead. At the same time, implementing some types of transparency can have a major impact on the design of a DBM system and require significant development effort, while only being required to correctly execute very few applications.

MAMBO aims to support *behavioural transparency*, meaning that it only implements the types and degrees of transparency required to correctly execute typical workloads, rather than aiming to achieve *full transparency*. Typical workloads are defined as applications that follow the platform ABI, do not depend on undefined behaviour (as described by the ARM architectural manual [ARM 2015]), and use standard system libraries.

One design decision affecting transparency is to use the application stack for scratch space, to work around the very limited address range (4KiB) reachable using immediate data transfer instructions on ARM. A fully transparent implementation would require the availability of a scratch register, either by stealing it from the application or by temporarily storing its value inside the code cache (due to the limited addressable range), likely incurring additional overhead. Because the ARM ABI prohibits applications from storing data on the stack at addresses lower than the stack pointer, a behaviourally transparent implementation can safely store temporary data (used in the translation of a single application instruction) on the stack. The only case when the scratch data on the stack requires special handling is for signal delivery; signals must be delivered when no temporary data is present on the stack.

3. OPTIMISATIONS FOR INDIRECT BRANCHES

Indirect branches are control flow instruction with a target not known at translation time. Looking up the target of indirect-branch instructions at runtime is the major source of overhead for DBM systems [Kim and Smith 2003]. Two types of indirect-branch instructions can be handled specially:

- Function returns*: Used at the end of functions to return control to the caller. The target address is either copied from a register or loaded from the stack.
- Table branches*: Used to implement switch statements from higher-level languages. The address or offset for such an instruction is loaded from a fixed table in memory.

Fig. 4. Example of a typical function call and the translation generated by MAMBO.

3.1. Function Returns: Low Overhead Return Address Prediction

Figure 4(a) shows a typical function call in ARM code. A *caller* function contains a branch-and-link to the entry address of the *callee*. The callee preserves the return address, executes, and returns to the instruction following the branch-and-link in the caller using a return instruction, which branches to the return address in the link register. Because it branches to an address in a register, this return instruction is an indirect branch. Fast return address prediction in DBM systems has been shown to be critical for achieving low overhead [Kim and Smith 2003].

While the return instruction is an indirect branch because its target address cannot be determined statically, it has the property that its translated address can be predicted with very high accuracy when its matching branch-and-link instruction executes. This property is sometimes exploited by a technique that stores pairs of untranslated and translated addresses on a shadow Return Address Stack (RAS) for every call instruction. Return instructions then load the untranslated address from the shadow RAS, compare it with the return address in the application, and, if they match, directly return to the translated address from the shadow stack. We call this linking scheme a *fat entry* RAS return-address predictor. This optimisation was used by several DBM systems, including Pin for ARM [Hazelwood and Klauser 2006] and fastBT [Payer and Gross 2010]. However, when implemented in MAMBO, this scheme causes a slowdown compared to our low-overhead inline hash lookup, a result similar to that experienced by DynamoRIO [Bruening 2004]. Additionally, the performance overhead of maintaining a shadow return address stack has been shown to be around 10% compared to direct native execution of returns [Dang et al. 2015].

We have developed an alternative scheme that trades off some transparency guarantees for increased performance, while still being able to execute typical applications correctly. Return-type instructions are initially translated to exits to the dispatcher and their basic block is marked as exiting with a return instruction. When the dispatcher handles an exit from this type of basic block, it looks up the translated address and compares it with the entry at the top of the RAS. In the case of a match, the dispatcher rewrites the exit code in the basic block to directly branch to the address at the top of the RAS. Translated returns are handled through the dispatcher only when they execute for the first time. Further executions of a translated return are handled through the fast RAS-based return operation inlined in the basic block. Our implementation of the call and return operations is shown in Figure 4(b).

3.1.1. Comparison with Fat Entry RAS Return-Address Prediction. Compared to the pair of 32-bit addresses used by a fat-entry RAS return predictor, the low-overhead return predictor only pushes and pops a single 32-bit value (the predicted code cache return address) for each branch-and-link and return instruction. On ARM, this eliminates 4 instructions from the critical execution path. In addition, the correctness of predictions

is not checked on the critical path for every return instruction. This eliminates another 4 instructions and avoids placing a conditional branch in the translation of returns, which would increase the pressure on the branch predictor and affect branch prediction rates. On applications using deep-nested function calls, the pressure on the data cache and on the data TLB is reduced due to the smaller size of the shadow return stack.

3.1.2. Restrictions. This scheme is not fully transparent because it relies on functions following the ARM ABI for function calls (which is always the case for compiler-generated code). When a return instruction is executed for the first time, the dispatcher verifies that the predicted return address is correct, which is intended to catch any mispredictions caused by use of nonstandard call or return operations, or by exceptions. When a misprediction is detected, MAMBO can be configured to:

- attempt to balance the RAS if it contains stale entries because of missed return instructions or stack unwinding,
- flush the code cache and disable this linking scheme (the default option), or
- print an error message and exit.

This configuration option allows the selection of different trade-offs between safety and performance. In addition, MAMBO will disable this scheme and flush the code cache if any instructions replace the value of the LR with a dynamic value (i.e., the value depends on other values apart from immediate operands or the PC). Static modifications of the LR are translated to push the predicted return address on the shadow stack.

Given these measures, our return-address prediction scheme can only cause a misprediction and execution of the incorrect code if the following conditions are met at the same time:

- the behaviour causing the misprediction is conditional and it does not execute before the affected return instruction is executed for the first time; and
- the modified return address is generated in a register different from LR, written to the stack and then POP-ed in the PC.

The only situation in which we have encountered this behaviour is in applications that throw exceptions. Because exception handling is done by unwinding the stack to search for exception handlers, it is possible for stale entries to remain on the RAS. However, the *libgcc* exception handling code that we have examined is guaranteed to cause a return-address misprediction because it always overwrites its return address, which is detected by MAMBO the first time an exception is thrown. Implementing a portable stack unwinding detector, which would allow use of this scheme in all applications that use exceptions, is a possible area of future development. Another case in which applications could cause mispredictions is when using the *longjmp/siglongjmp* functions. These functions are implemented in *glibc* similarly to the exception handling code, and also overwrite their return address, which allows MAMBO to detect the misprediction. We have not encountered any applications that cause RAS mispredictions only after one or more executions of a return instruction, which would not be possible to detect when using the low-overhead return-address predictor.

3.2. Table Branches: Space-Efficient Linking

Table branch instructions are a type of indirect branch that determine their target address by loading it from a table in memory, indexed by a dynamic value. Table branch instructions are commonly used by compilers to implement switch-type constructs.

Fig. 5. Space-efficient shadow branch table.

Fig. 6. Comparison of shadow branch table size for SPEC CPU2006.

We propose a scheme that adapts the *shadow jump table* linking scheme introduced by fastBT [Payer and Gross 2010] for use on the ARM architecture and improves the space efficiency of the shadow branch table.

Shadow branch table size. Most table branch instructions have the property that the number of different indexes used during a typical execution (*selected_indexes*) is much lower than the largest selected index (*max_index*). The fastBT scheme allocates a shadow jump table that can fit all code cache targets starting from index 0 and up to a maximum index. When ported to ARM, the size of the shadow table is $fastbt_table_size = (max_index + 1) \cdot 4$ bytes, each entry being encoded as a 32-bit target address.

The space-efficient shadow branch table that we have implemented, shown in Figure 5, uses a two-level table; the first-level table contains *max_index* bytes. The first table encodes offsets into a second-level table that contains direct branches (of size 4 bytes) to basic blocks. The size of this shadow table is $mambo_table_size = max_index + 1 + selected_indexes \cdot 4$.

We have profiled the two parameters, *selected_indexes* and *max_index*, for table branch instructions executed by SPEC CPU2006 benchmarks. The violin plot in Figure 6 shows the distribution of shadow branch table sizes for the two schemes. The width of the curve at various positions on the vertical axis shows the distribution of shadow branch tables of the corresponding size in each benchmark. The red horizontal

markers show the minimum, mean, and maximum (when under 400 bytes) sizes. Use of our scheme reduces both the median and maximum shadow branch table size for all SPEC CPU2006 benchmarks. Across all SPEC CPU2006 benchmarks, the mean shadow table size is reduced from 159 bytes to 72 bytes and the maximum size is reduced from 2,700 bytes to 1,388 bytes.

Use of the space-efficient shadow branch table saves on average 3,550 bytes of code-cache space per benchmark. Most space is saved for the gcc (47KiB), perlbench (21KiB) and gobmk (10KiB) benchmarks.

Detection. The Thumb instruction set, unlike x86 and ARM, includes explicit table branch instructions, called *TBB* and *TBH*. Both Thumb table branch instructions take as input two registers: the first points to the beginning of the table in memory and the second is the index. The branch offset is the value loaded from the table multiplied by two. In ARM mode, table branches are usually implemented with a load instruction that uses the program counter as both the destination and the base registers and, with another register, shifted left 2 bits, as the index.

Implementation. Figure 5 shows our implementation of the space-efficient shadow branch table. The size of the *offset table* determines the maximum index that can be handled; the size of the *shadow branch table* determines how many unique indexes can be cached. Indexes higher than *max_index* are handed off to an inline hash lookup routine. Similarly, if the shadow branch table gets filled, any additional indexes that execute are redirected to the inline hash lookup by setting the appropriate offset in the offset table. The inline hash lookup routine is discussed separately in Section 3.3. The current value of *max_index* is 152 and the size of the shadow branch table is 32 entries. These values were chosen experimentally as a compromise between size requirements and performance for the SPEC CPU2006 benchmarks. These sizes allow over 99.9% of the table branches executed across SPEC CPU2006 benchmarks to be cached in the shadow branch tables.

The default value for all offsets in the offset table points to *call_dispatcher*, so that the dispatcher is always called when a certain index is selected for the first time. The *call_dispatcher* routine passes both the untranslated target address and the index to the main dispatcher, which, based on these parameters, allocates an available entry in the shadow branch table and updates the offset in the offset table. Once a specific index is cached, all future executions with the same index will use the cached code-cache target.

Does the extra level of indirection affect performance? Even though the space-efficient shadow branch table appears to trade off performance for space compared to the single level fastBT shadow branch table, experimental results do not match this. When evaluated using a microbenchmark, which applies no pressure on the branch predictors or caches, the fastBT scheme is around 5% faster than the space-efficient scheme. However, when used by MAMBO running a more complex workload, such as the SPEC CPU benchmarks, the space-efficient scheme is consistently faster than the fastBT scheme. Benchmarking and profiling results are discussed in Section 5.3.

Restrictions. This scheme is applied only when the base of the branch table used by the translated TBB or TBH instruction can be statically determined (which is the case for compiler-generated code implementing switch statements) and if the table is stored in write-protected memory (which is the case for the code segment in ELF executables). The shadow branch table must be invalidated if the application unmaps or remaps with write permissions the area in which the branch table resides, which has been observed to happen only rarely, with virtually no impact on the performance of translated table branches.

```

struct hash_table_entry {
    uint32_t application_address;
    uint32_t code_cache_address;
};

struct hash_table {
    hash_table_entry entries[TABLE_SIZE+OVERP];
};

```

Listing 1: Structure of the hash table.

3.3. Inline Hash Lookup for Indirect Branches

The previously described linking schemes deal with only special cases of indirect branches, leaving generic indirect branches to execute through the dispatcher and incurring a high overhead from the associated context switch. Because the targets of indirect branches cannot be statically determined, direct linking similar to that used by the previous schemes is impossible to implement efficiently. Instead, this linking scheme aims to:

- minimise the context switch overhead,
- implement an efficient hash lookup routine, and
- facilitate hardware branch target prediction.

The hash lookup routine is encoded inline in the basic blocks that contain the translation of an indirect branch. This allows adapting every instance to minimise context switch cost and also allows the processor to handle branch target prediction for every translated indirect branch individually.

The structure of the hash table is defined in Listing 1. Each entry consists of a pair of 32-bit addresses: the untranslated application address and the translated address in the code cache. Our implementation uses linear probing to solve collisions. To minimise the number of required registers, there is no wraparound. Instead, a number of additional slots are used to handle possible collisions at the end of the table, followed by a guard entry, identical to empty slots, which marks the end.

The hash function is simple and can be implemented with a single bitwise AND instruction:

```
hash = key & 0x1FFFF
```

Because the keys are code addresses, all bits are significant, including the LSB that is used to indicate the instruction set (ARM or Thumb). The size of the hash table is around twice the maximum number of basic blocks to minimise the number of collisions. We have found that hash-table collisions become very expensive due to branch misprediction in the hash-lookup routine.

Figure 7 shows the inline hash-lookup routine. The first operation saves the context. This operation is specialised for each instance of the inline hash lookup and frees up a number of scratch registers. It saves only a reduced number of registers or none at all if a stack pop in the application can be delayed until the hash-lookup routine has completed, subject to data dependencies.

Next, the hash key is computed using bitwise instructions and the corresponding entry is loaded from the hash table. If the *application_address* in the hash table entry is equal to the target address, the lookup is successful; therefore, the values of the scratch registers are restored and execution branches directly to the *code_cache_address*. In the case of a mismatch, it is checked to see whether the hash table entry is empty. If it is, there is no translation of the target address in the code cache and the dispatcher is called. If the entry is not empty, it is still possible for the correct entry to be at another index due to a collision, in which case the previously computed hash key is incremented and execution loops back. This loop is exited only when either the correct entry or an empty entry is found.

All inline hash-lookup routines access a single thread-private hash table; therefore, encoding a new routine takes up only code-cache space and not any additional data

Fig. 7. Inline hash-lookup routine.

Fig. 8. Linking of fallthrough branches.

space. The size of the inline hash-lookup routine is between 94 and 118 bytes in Thumb mode and either 116 or 120 bytes in ARM mode, depending on the type of translated instruction and available registers. This includes fallback code for calling the dispatcher if the translation of the target address is not yet present in the code cache. Section 5.5 evaluates the overall code-cache overhead when enabling inline hash lookups for SPEC CPU2006 benchmarks.

3.4. Fallthrough Branch Linking

Conditional branches have an implicit fallthrough branch to the following instruction. The fallthrough branch is taken only if the conditional branch is skipped. Even if the conditional branch is indirect, the fallthrough branch is always a direct branch, which can be directly linked in the translated code. Figure 8 shows how fallthrough branches are translated: MAMBO links the fallthrough branch by placing a conditional direct branch with the opposite condition compared to that of the source branch before the indirect-branch lookup routine.

To avoid the potential overheads and error transparency issues involved in scanning the target of a fallthrough branch before it is taken, we use *stub basic blocks*. Stub basic blocks are basic blocks that are allocated in the code cache and can be linked to, but have not been scanned. Stub basic blocks contain only a call to the dispatcher. When a stub basic block is executed for the first time, the scanner overwrites the initial stub with the translated contents of the basic block.

3.5. Indirect-Branch Target Prediction

Other DBM systems, such as Pin [Luk et al. 2005], use a short series of inlined compare-and-branch sequences as their main method of resolving indirect branches. However, we have found that, on ARM, such an approach is generally outperformed by our fast inline hash-lookup system. A fundamental limitation of a compare-and-branch predictor is that updating the predicted addresses is a relatively expensive operation (as

opposed to a hardware indirect-branch predictor, which can be updated every time the indirect branch executes); therefore, the performance of the system relies on indirect branches being predictable and on using a good heuristic to decide which target to select. However, by instrumenting the indirect branches in SPEC CPU2006 benchmarks, which make heavy use of indirect branches, we have determined that many indirect branches are not easily predictable. An oracle that always predicts the address taken most often for each indirect branch can get misprediction rates as high as 80% for *perlbench* with *splitmail.pl*, 52% for *sjeng*, or 47% for *gcc* with *scilab.in*. A practical branch predictor heuristic is likely to have even higher miss rates.

A second issue with compare-and-branch predictors is that only a few execution cycles can typically be saved compared to an inline hash lookup, only when both the software predictor and the hardware branch predictor hit. However, the compare-and-branch predictor uses additional conditional branches that can be mispredicted by the hardware branch predictor. Hardware branch mispredictions have a latency proportional to the number of pipeline stages (e.g., around 20 cycles for Cortex-A15) and can easily start to dominate the lookup time. An inline hash lookup is generally expected to cause at most one hardware branch misprediction, only if the target is different compared to the last execution of the branch. On the other hand, a 2-entry compare-and-branch predictor with a fallback inline hash lookup can cause up to three branch mispredictions.

4. OPTIMISATIONS FOR DIRECT BRANCHES

4.1. Direct-Branch Linking

When basic blocks are first created, their exit code stub saves the application context, sets or computes the target address, then branches to the code dispatcher. However, the context switch and the call to the dispatcher introduce significant overhead. This linking scheme avoids that overhead by replacing the context switch and call to the dispatcher with direct branches to the translated target basic block, if it is present in the code cache.

There are two types of direct branches:

- unconditional direct branches, which includes various types of branch and branch-and-link instructions; and
- conditional direct branches—direct branches can be executed conditionally either because the instruction encoding explicitly supports conditional execution or because they are preceded by an If-Then (IT) instruction that makes them conditional.

The various encodings support different branch offsets, from a range of $-256/254\text{B}$ bytes up to $\pm 16\text{MiB}$ in Thumb mode and $\pm 32\text{MiB}$ in ARM mode. When direct-branch linking is used in the code cache, most types of branches are linked using branch instructions with the maximum range. This range defines the maximum size of a code cache; a larger code-cache size would require linking using slower indirect-branch instructions.

Conditional branches that use the status register are linked as a conditional branch (implemented as an IT instruction followed by a branch) and one unconditional branch, to link both possible execution paths. Compare and branch on zero/nonzero (CBZ/CBNZ) instructions conditionally branch depending on the result of the comparison between a register and the value 0. These have a very limited range, thus are linked using a CB(N)Z instruction and two unconditional branches, with the CB(N)Z instruction conditionally skipping over the first unconditional branch.

Fig. 9. Comparison between translations with and without unconditional direct-branch eliding.

4.2. Eliding Unconditional Direct Branches

Modern ARM cores use either 32-byte or 64-byte cache lines, while instruction words are only 2 or 4 bytes in length. Because the software code cache allocates basic blocks of fixed size, execution of short basic blocks can potentially fill a large portion of the hardware instruction cache with invalid code that never executes.

This optimisation aims to improve the density of valid translated code in the hardware instruction cache by increasing the average size and reducing the total number of basic blocks. Instead of stopping the code scanner when encountering unconditional direct branches or branch-and-link instructions, these instructions are instead elided by continuing to translate, in the same basic block, the target of the branch. When this linking scheme is enabled, it takes precedence over unconditional direct-branch linking; all unconditional direct branches are elided instead of being linked.

Figure 9 shows a comparison between the translations of the application code from Figure 4(a) with and without eliding the unconditional direct branch. In this example, by eliding the unconditional branch, the number of basic blocks is lowered to one, the total number of instructions is reduced, and a branch instruction is eliminated.

A disadvantage of this linking scheme is caused by *tail duplication*, with potential overhead in terms of total code-cache size: code that would otherwise be translated only once, in a single basic block, can end up being duplicated in multiple basic blocks.

Eliding unconditional branches can cause the scanner to attempt generating infinite-size basic blocks, for example, when scanning a loop that contains no conditional control flow instructions. To prevent this from occurring, we limit the maximum number of elided unconditional branches in a single basic block. We use different limits for forward and backward branches. The limit for forward branches controls the size of code that can potentially be duplicated in multiple basic blocks, while the lower limit for backward branches is primarily intended to control duplication within unique basic blocks, that is, loop unrolling.

The violin plot in Figure 10 compares the distribution of basic block sizes depending on whether this linking scheme is enabled or not. The width of the curve at various positions on the vertical axis shows the distribution of basic blocks of the corresponding size in each benchmark. The red horizontal markers show the minimum and median sizes. It can be observed that eliding unconditional branches increases both the median and minimum size of basic blocks compared to linking unconditional branches. Four-byte basic blocks (i.e., the minimum size of basic blocks for most benchmarks with linked unconditional direct branches), generated when the first instruction in a basic block is an unconditional direct branch, are completely eliminated.

Fig. 10. Comparison of basic block sizes.

5. EVALUATION

5.1. Experimental Setup

The results presented in this section have been obtained on two single-board computers:

- ODROID-X2, which is built around a Samsung Exynos 4412 Prime System-on-Chip with 4 Cortex-A9 cores running at 1.7GHz, with 32KiB L1 data and instruction caches (32-byte cache lines) and a shared 1MiB L2 cache. The system has 2GiB of LP-DDR2 memory; and
- Jetson TK1, build around an NVIDIA Tegra K1 System-on-Chip with 4 Cortex-A15 cores running at 2.3GHz, with 32KiB L1 data and instruction caches (64-byte cache lines) and a 2MiB L2 cache. This system has 4GiB of DDR3L memory.

Power-management features such as DVFS and core power-gating were disabled and a fan was added to the passive heatsink of the ODROID-X2 to minimise the risk of thermal throttling. SPEC CPU2006 was compiled with GCC 4.6.3, and PARSEC 3.0 was compiled with GCC 4.8.2, both configured to generate Thumb code (which is the default configuration) with NEON and VFP support. Nonessential services were disabled and the systems were otherwise idle.

The *libquantum* benchmark from the SPEC CPU2006 suite has been disabled because it fails to complete, both when executed natively and under MAMBO. All other CPU2006 benchmarks are enabled when running natively or under MAMBO and produce the expected output. Valgrind fails to load the *zeusmp* benchmark because of its large BSS section (1.1GiB) and throws an exception when running *povray* because it fails to decode a valid *ADD.W* instruction. The benchmarks *Canneal* and *Raytrace* from PARSEC 3.0 do not build on ARM. *fluidanimate* requires the number of threads to be a power of two; therefore, it cannot run with three threads. All SPEC CPU2006 results were obtained using the *ref* dataset and all PARSEC 3.0 results were obtained using the *native* dataset.

5.2. Contribution of Different Optimisations

Figures 11 and 12 show a comparison of overhead with different types of optimisations being enabled. The results using the A9 suffix were obtained on the ODROID-X2 system; the results using the A15 suffix were obtained on the Jetson TK1 system. The five configurations that were evaluated are:

- MAMBO—all linking schemes are enabled, it is the fastest version;
- MAMBO-RAS—return-address prediction using the shadow RAS is disabled and return instructions are translated to inline hash lookups;

Fig. 11. Relative execution time for SPEC CPU2006 with the *ref* dataset on ODROID-X2. Fig. 12. Relative execution time for SPEC CPU2006 with the *ref* dataset on Jetson TK1.

- MAMBO-RAS-TB—both return-address prediction and table-branch linking are disabled, table branches are handled using the dispatcher;
- MAMBO-RAS-TB-EUIDB—the previous two linking schemes are disabled and unconditional direct branches are being directly linked to the basic block containing the translation of their target instead of being elided; and
- MAMBO-RAS-TB-EUIDB-IH—the previous three linking schemes and the inline hash lookup are disabled. All types of indirect branches are handled using calls to the dispatcher.

The benchmarks marked with a caret (^) cause return-address mispredictions that require disabling of the RAS predictor at some point during execution.

The inline hash lookup routine is essential for achieving low overhead. The geometric mean of the relative time for -RAS-TB-EUIDB-IH is 1.63 on ODROID-X2 and 1.81 on Jetson TK1, with maximums of 4.44 on ODROID-X2 and 5.37 on Jetson TK1. When inline hash lookups are enabled, the geometric mean of relative times is reduced to 1.36 on ODROID-X2 and 1.42 on Jetson TK1, and the maximums to 3.16 on ODROID-X2 and 3.58 on Jetson TK1. This corresponds to 42% and 48% lower overhead on ODROID-X2 and Jetson TK1, respectively.

Fig. 13. Relative slowdown for selected SPEC CPU2006 benchmarks with the fastBT table-branch linking scheme.

The table-branch linking optimisation affects benchmarks for which table branches are a significant part of the total number of indirect branches: *perlbench*, *gcc*, *sjeng*, *gamess*, and *soplex*. For these benchmarks, the overhead is reduced, on average, by 28% on ODROID-X2 and 25% on Jetson TK1.

Enabling the return-address predictor reduces the overhead of benchmarks with many calls to functions that return relatively quickly. At the same time, benchmarks with high data-cache pressure can be negatively affected by the additional operations on the RAS, and benchmarks with high instruction-cache pressure can be negatively affected by the increased code-cache size.

Eliding unconditional direct branches reduces the average overhead by 5% on ODROID-X2 and by 2% on Jetson TK1. On the ODROID-X2, all benchmarks run at least as quickly with elided unconditional direct branches than without. However, on the Jetson TK1, that is no longer the case, and some benchmarks suffer a slowdown (e.g., 1.8% higher overhead on *perlbench*).

By comparing the relative speedup when the return-address predictor and table-branch linking are enabled, it can be observed that both of these linking schemes appear to be more effective on Cortex-A15. However, based on the higher overhead on Cortex-A15 when using the -RAS-TB and -RAS-TB-IH versions, which rely on hash lookups to resolve indirect branches, we conclude that, in fact, hash lookups perform worse on Cortex-A15. We attribute this behaviour primarily to the higher branch misprediction penalty on Cortex-A15, taken when using linear probing to look up hash table entries with collisions. The return-address prediction and table branch linking schemes generally avoid unpredictable branches, which allows for greater speedup compared to hash lookups.

Another likely contributor to the higher overhead on Cortex-A15 is the use of larger cache lines compared to Cortex-A9 (leading to poor density of valid code) without increasing the size of the L1 instruction cache. This effectively reduces the maximum size of translated code that is cached.

5.3. Comparison of the Space-Efficient and fastBT Table-Branch Linking Schemes

To compare the two table-branch linking schemes, we have limited the set of benchmarks to the six that make significant use of table branches: *perlbench*, *gcc*, *sjeng*, *gamess* and *soplex*, using the *ref* dataset. Figure 13 shows the slowdown of two fastBT configurations relative to the space-efficient scheme:

- fastBT-70 uses the fastBT scheme with up to 70 cached targets, and
- fastBT-152 uses the fastBT scheme with up to 152 cached targets.

Table I. Number of Branch Mispredictions on Jetson TK1 with Different Implementations of Shadow Branch Tables

Benchmark	fastBT-70	fastBT-152	space-efficient BT
perlbench	50,701,305,241	52,214,438,915	31,119,438,795
gcc	22,506,245,623	22,609,245,529	18,001,564,317
sjeng	88,829,466,475	87,830,777,975	77,088,995,881
gamess	21,040,036,701	20,159,607,431	17,513,213,671
soplex	9,076,672,304	9,062,995,152	7,573,834,491

The results were obtained on the Jetson TK1 system. Lower values are better (faster execution).

The maximum index and number of cached targets for the space-efficient scheme were determined experimentally to produce good results with acceptable memory usage across a variety of workloads. For the fastBT scheme, 70 was chosen as a maximum number of cached targets because it uses the same amount of code-cache space as the space-efficient scheme; 152 was chosen to allow the fastBT scheme to cache targets up to index 151, the same as the space-efficient scheme. fastBT-152 reserves more than double the amount of space compared to the space-efficient scheme (608 bytes instead of 280).

In all cases, using the fastBT scheme instead of the space-efficient scheme increases the execution time, up to 3.6% for fastBT-70 and up to 5% for fastBT-152. The difference between different applications is roughly proportional to the number of table-branch instructions, it does not show the efficacy of table-branch linking varying between applications.

fastBT-152 is faster than fastBT-70 for some benchmarks, and fastBT-70 is faster for other benchmarks. This depends on the distribution of table-branch indexes used by each benchmark (fastBT-152 can directly link the translation of higher indexes) and by the pressure on the cache subsystem (fastBT-152 reserves more space for the shadow branch table).

Profiling using the performance counters on the Cortex-A15 shows that the space-efficient scheme runs with fewer branch mispredictions compared to the fastBT shadow branch tables (see data in Table I). For example, perlbench runs with 40% fewer mispredictions on the Cortex-A15 when using the space-efficient shadow table compared to the fastBT-152 shadow table. Unfortunately, the indirect branch predictor in these ARM cores is not documented; therefore, the root cause of this behaviour remains unexplained.

5.4. Overall Performance

5.4.1. Single-threaded Performance: SPEC CPU2006. The results in Figure 14 were obtained by running SPEC CPU2006 with the *ref* dataset, using MAMBO with all linking schemes enabled, Valgrind 3.10, and QEMU 2.0.0 in user mode. QEMU results are reported only for the Cortex-A15 system, due to time constraints caused by QEMU's high overhead. We estimate that SPEC CPU2006 would take more than 20 days to finish running under QEMU on the ODROID-X2. The geometric mean of overheads is summarised in Table II.

MAMBO has higher overhead when running on Jetson TK1 compared to ODROID-X2 for most benchmarks. The likely causes of this pattern are described in Section 5.2. In some cases, the difference can be very large (e.g., from 3% to 28% overhead for soplex) and it warrants further investigation. However, several benchmarks have lower overhead on Cortex-A15 compared to Cortex-A9. These benchmarks likely benefit from having a larger L2 cache, which can better accommodate the larger working set of translated applications.

Fig. 14. Relative execution time for SPEC CPU2006 under MAMBO, Valgrind and QEMU (*ref* dataset).

Table II. Summary of Geometric Mean Overheads for MAMBO, Valgrind and QEMU Running SPEC CPU2006

DBM system	MAMBO		Valgrind		QEMU
System	A9	A15	A9	A15	A15
SPECint	1.57	1.67	3.92	4.64	8.03
SPECfp	1.11	1.16	2.84	3.36	36.31
SPEC CPU	1.28	1.34	3.26	3.85	20.07

Some benchmarks execute with significant overhead under MAMBO. By profiling using the hardware performance counters, the main causes have been determined to be poor L1 instruction-cache utilisation leading to a significant increase in cache misses compared to native execution and high instruction TLB miss rates. Both issues are caused by the fixed-size basic block layout used by MAMBO. This layout causes basic blocks to exclusively use at least one cache line (32 bytes on a Cortex-A9 or 64 bytes on a Cortex-A15) even when their length is significantly shorter. It also spreads out the translated code across more pages than the untranslated code. This issue could be addressed either by modifying the code cache to use variable size basic blocks or by implementing traces.

MAMBO has lower overhead than Valgrind and QEMU on every benchmark. On average, MAMBO has 8.1 times lower overhead than Valgrind on the Cortex-A9 system (28% vs. 226%) and 8.4 times lower overhead than Valgrind on the Cortex-A15 system. QEMU has 56 times higher overhead than MAMBO and 8.3 times higher overhead than Valgrind. The highest overhead on a single benchmark is 154% for MAMBO (perlbench on Cortex-A15), 656% for Valgrind (perlbench on Cortex-A15), and 8707% for QEMU (calculix on Cortex-A15). QEMU's higher overhead on SPECfp is caused by its inefficient translation of VFP instructions, which are translated to function calls to instruction-emulation routines. MAMBO and Valgrind can both generate native VFP code, which will run with no or low overhead.

leslie3d obtains a significant speedup under MAMBO compared to native execution, around 25%. Performance counter analysis shows this is caused by reduced L1 data-cache miss rates (from 14% to 8%) and a subsequent 29% reduction in the number of L2 cache misses. The number of executed instructions is essentially unchanged. MAMBO affects the memory layout of applications by reserving space for itself, however it does not perform any memory-related optimisations on the translated applications; speeding up *leslie3d* appears to be a coincidental side-effect.

5.4.2. Multithreaded Performance: PARSEC 3.0. The results in Figure 15 were obtained by running PARSEC 3.0 with the *native* dataset and MAMBO with all linking

Fig. 15. Relative execution time for PARSEC 3.0 with the *native* dataset.

optimisations enabled, on the Jetson TK1 board. The geometric mean is 1.30 when running with one thread, 1.27 for two threads, and 1.32 for three and four threads.

Multithreaded scaling. Generally, MAMBO shows good performance scaling with multiple threads. The poor scaling shown for the *x264* benchmark is explained by its threading model: most threads it creates execute for only 1s to 4s before exiting, causing MAMBO to translate and link the same code for each newly created thread. Several benchmarks (*dedup*, *freqmine*, and *streamcluster*) have higher overhead for single-threaded execution compared to 2 or more threads. *blackscholes* and *dedup* show poor scaling, with overhead increasing as more threads are used.

5.5. Code-Cache Size

The size of the code cache has been measured in the same five configurations described in Section 5.2. The results for all SPEC CPU2006 benchmarks are shown in Table III. For benchmarks that are launched multiple times with different inputs, the arithmetic mean is shown. For benchmarks that cause return mispredictions and a subsequent flushing of the code cache in the *MAMBO* configuration, marked with a caret ($\wedge$), only the higher value between the size at the time of the misprediction and the size at exit is shown. Note that MAMBO uses lazy linking for conditional branches; if a translation of either of the two possible targets does not exist in the code cache yet, an exit stub that calls the dispatcher is generated. The size of these exit stubs is included in the reported code cache size, even though they will execute, at most, two times and are therefore expected to have a minimal impact on the hardware-cache hit rate.

Enabling the inline hash lookup (column *MAMBO-RAS-TB-EUDB* vs. *MAMBO-RAS-TB-EUDB-IH*) increases the size of the code cache across SPEC CPU2006 benchmarks by around 1.3% and makes no impact on the number of basic blocks. When unconditional direct branches are elided (column *MAMBO-RAS-TB* vs. *MAMBO-RAS-TB-EUDB*), the size of the code cache is increased by around 1% due to code duplication; however, the number of basic blocks is reduced by approximately 3.8%. When table-branch linking is enabled (column *MAMBO-RAS* vs. *MAMBO-RAS-TB*), the code-cache size increases by another 1.9% and the numbers of basic blocks increases by around 0.3%, due to the space required for shadow branch tables. Enabling return-address prediction (column *MAMBO* vs. *MAMBO-RAS*) increases the code-cache size by 21.9% and reduces the number of basic blocks by 9.8%. Due to the fixed size basic block layout used by MAMBO, an increase in code-cache size is not necessarily going to increase the pressure on the hardware instruction cache, as long as the number of basic blocks is not also increasing.

Table III. Code-Cache Size for SPEC CPU2006, in KiB and Number of Basic Blocks

Configuration Benchmark	MAMBO		MAMBO-RAS		MAMBO-RAS -TB		MAMBO-RAS -TB-EUDB		MAMBO-RAS -TB-EUDB-IH	
	KiB	BBs	KiB	BBs	KiB	BBs	KiB	BBs	KiB	BBs
perlbench [^]	887	15153	1002	17178	970	17069	949	17822	933	17824
bzip2	151	2060	143	2131	141	2128	142	2265	142	2265
gcc	3356	44447	2835	52458	2726	52194	2659	53875	2637	53864
mcf	120	1645	112	1659	110	1655	110	1746	104	1746
gobmk	1174	15835	1122	18990	1117	18978	1071	19549	1013	19543
hmmer	246	3150	218	3453	215	3447	214	3610	205	3609
sjeng	240	3261	220	3384	215	3373	203	3472	198	3470
h264ref	540	6733	487	7257	481	7243	477	7615	473	7623
omnetpp [^]	487	9115	577	11192	572	11180	573	11521	524	11516
astar	199	2474	158	2480	156	2476	153	2579	147	2579
xalancbmk	2409	25281	1787	27961	1768	27918	1761	29037	1439	29026
bwaves	249	2897	202	2976	194	2956	194	3118	191	3117
gamess	1176	13125	1058	14160	1039	14111	1031	14807	1094	14800
milc	242	2887	202	3166	199	3159	200	3333	196	3327
zeusmp	612	6088	513	6393	504	6370	496	6588	545	6586
gromacs	440	5226	400	5463	392	5444	392	5723	378	5720
cactusADM	601	6458	452	7503	445	7488	449	7803	439	7803
leslie3d	373	4279	319	4527	309	4507	311	4747	320	4744
namd	299	3724	281	3926	279	3922	275	4067	286	4063
dealII	1060	11941	803	13376	798	13366	803	14051	782	14050
soplex	517	5832	397	6123	391	6110	390	6397	365	6391
povray [^]	564	9128	669	11042	654	11006	647	11504	632	11503
calculix	1008	12001	821	13740	806	13706	802	14276	814	14268
GemsFDTD	887	9785	746	11327	733	11297	734	11708	772	11705
tonto	1730	17445	1349	19107	1315	19026	1303	19757	1352	19750
lbm	96	1279	89	1291	88	1289	88	1365	85	1365
wrf	3187	27655	1760	25689	1744	25650	1746	26309	1871	26302
sphinx3	414	5135	356	5753	353	5747	353	6019	349	6011

6. RELATED WORK

Most low-overhead DBM systems have focused on the x86 architecture, while MAMBO targets ARM. This fundamental difference makes it difficult to directly compare the design or performance of these systems. However, MAMBO employs and builds upon a large number of techniques from the DBM/DBT literature, which we have specialised for ARM.

DynamoRIO. DynamoRIO is an open-source dynamic instrumentation platform for x86 and x86_64. The design of MAMBO has been influenced by DynamoRIO. For example, our implementation of an inline hash lookup routine for resolving indirect branches takes inspiration from that of DynamoRIO [Bruening 2004]; however, we reserve additional space at the end of a hash table to avoid false misses when handling collisions, without having to wrap around. Compared to DynamoRIO, which is a mature, full-featured DBM tool, MAMBO is easier to understand and modify because of its small code base (around 25 times fewer lines of code than core DynamoRIO). DynamoRIO is currently being ported to ARM.

fastBT. fastBT is a generator for low-overhead dynamic binary translators, currently only supporting x86_32. It introduced the *shadow jump table* linking scheme [Payer and Gross 2010], which we have ported to ARM and optimised for size. Our optimisation

saves up to 47KiB of code-cache space in the gcc SPEC CPU2006 benchmark, which can reduce cache pressure.

Pin for ARM. The Pin dynamic instrumentation system was ported to ARM in the past [Hazelwood and Klauser 2006]; however, the ARM port has been discontinued and performance results for current ARM cores are not available.

The main differences between Pin for ARM and MAMBO is that Pin did not make use of an inline hash lookup routine; instead, it used a local prediction table to speed up generic indirect branches and a *fat entry* RAS-based return-address predictor for function returns. In our testing, both of these linking schemes appear to be slower compared to an inline hash lookup when running on current Cortex-A ARM cores. Pin for ARM used traces in addition to basic blocks, a feature that is not yet implemented in MAMBO.

HDTrans. HDTrans uses an eager return-address prediction scheme based on a *return cache* [Sridhar et al. 2007]. The result of the prediction is validated in the translation of the return code. Compared to our RAS predictor, it incurs additional overhead because the result of the prediction is checked dynamically. Additionally, hash collisions in its return cache are expensive.

QEMU and HQEMU. QEMU [Bellard 2005] is a dynamic binary translator that implements both user- and system-level virtualisation and supports translation to and from many architectures, including ARM to ARM, which is similar to running MAMBO without an instrumentation plugin. QEMU has been designed for portability and handles translation using an intermediate representation. Our evaluation (Section 5.4.1) shows that QEMU has significant overhead, with a 20x slowdown when used for ARM-to-ARM userspace translation.

HQEMU improves QEMU's performance by using LLVM as a backend for code generation and by improving the trace-selection algorithm [Hong et al. 2012]. An evaluation of ARM-to-ARM translation is not provided; however, the ARM-to-x86 performance is improved by a factor of 2.4X, from the 8.2X geometric mean slowdown for SPEC CINT2006 of QEMU to 3.4X for HQEMU.

Valgrind. Valgrind is a dynamic instrumentation framework designed for heavy-weight instrumentation [Nethercote and Seward 2007]. While Valgrind has up-to-date support for the ARM ISA, it is not designed for performance: it serialises multithreaded applications and it has high overhead, in our evaluation (Section 5.4.1) achieving 3.26x and 3.85x geometric mean slowdown (on ODROID-X2 and Jetson TK1, respectively) for SPEC CPU2006 with the Nulgrind *null* instrumentation. In contrast, MAMBO has low execution overhead and uses a shared-nothing architecture to achieve good multithreaded and multicore scaling.

7. CONCLUSIONS

MAMBO is a new low-overhead DBM tool for the ARM architecture, with support for the ARM and Thumb instruction sets and the optional VFP and NEON extensions. We have presented an overview of MAMBO and have covered how to optimise direct and indirect branches. By trading off the transparency of the DBM, we have been able to propose, implement, and evaluate novel optimisations for function returns as well as table branches. The first optimisation allows MAMBO to use a low-overhead return-address prediction. The new heuristic return-address prediction scheme eliminates the prediction correctness check from the hot execution path. The second optimisation provides a more space-efficient table-branch linking scheme. Further optimisations have been described for direct-branch linking: unconditional direct branches and conditional direct branches, plus addressing conditional indirect branches.

The performance evaluation shows that MAMBO introduces small overheads when compared with native execution: for SPEC CPU2006 a geometric mean overhead of 28% on a Cortex-A9 and of 34% on a Cortex-A15; and for PARSEC 3.0 a geometric mean overhead of 27% to 32%, depending on the number of threads, on a Cortex-A15. A major source of remaining overhead has been determined to be L1 instruction cache and instruction TLB misses. Addressing this issue is a topic of future research.

APPENDIX

A. EXAMPLE PLUGIN: DYNAMIC EXECUTION COUNTER FOR TBB AND TBH INSTRUCTIONS

```
// Called for each instruction scanned by MAMBO, before the translation is generated
void inst_cnt_pre_inst_handler(mambo_context *ctx) {
    if (mambo_get_type(ctx) == MAMBO_THUMB
        && (mambo_get_inst(ctx) == THUMB_TBB32) || (mambo_get_inst(ctx) == THUMB_TBH32)) {
        emit_thumb_push16(ctx, (1 << r0) | (1 << r1) | (1 << r2)); // PUSH {R0-R2}
        emit_thumb_push_cpsr(ctx, r0); // MRS R0, CPSR; PUSH {R0}
        // MOVW R0, #(ptr_to_ctr & 0xFFFF)
        // MOVT R0, #(ptr_to_ctr >> 16)
        emit_thumb_copy_to_reg32(ctx, r0, mambo_get_thread_plugin_data(ctx));
        emit_thumb_ldrdi32(ctx, r1, r2, r0, 0); // LDRD R1, R2, [R0, #0]
        emit_thumb_addi16(ctx, r1, r1, 1); // ADDS R1, R1, #1
        emit_thumb_adci32(ctx, r2, r2, 0); // ADCS R2, R2, #0
        emit_thumb_strdi32(ctx, r1, r2, r0, 0); // STRD R1, R2, [R0, #0]
        emit_thumb_pop_cpsr(ctx, r0); // POP {R0}; MSR CPSR, R0
        emit_thumb_pop16(ctx, (1 << r0) | (1 << r1) | (1 << r2)); // POP {R0-R2}
    }
}

// Called when a new thread is created, including the initial thread
void inst_cnt_pre_thread_handler(mambo_context *ctx) {
    uint64_t *inst_counter = mambo_alloc(ctx, sizeof(uint64_t));
    *inst_counter = 0;
    assert(inst_counter != NULL);
    mambo_set_thread_plugin_data(ctx, (uint32_t)inst_counter);
}

// Called when a thread exits, or in all threads when the process exits
void inst_cnt_post_thread_handler(mambo_context *ctx) {
    uint64_t *inst_counter = mambo_get_thread_plugin_data(ctx);
    printf("%llu instructions executed in thread %d\n", *inst_counter, mambo_get_thread_id());
    mambo_free(ctx, inst_counter);
}

void init_plugin() {
    mambo_register_pre_inst_cb(&inst_cnt_pre_inst_handler);
    mambo_register_pre_thread_cb(&inst_cnt_pre_thread_handler);
    mambo_register_post_thread_cb(&inst_cnt_post_thread_handler);
}
```

Listing 2: Example plugin: Dynamic execution counter for TBB and TBH instructions.

This plugin uses MAMBO's plugin API to modify the instruction stream: for each TBB or TBH instruction in the scanned code, it inserts an inlined code snippet that temporarily stores the required context on the stack; it loads, increments and stores back a 64-bit counter; and, finally, it restores the context of the application. This is achieved using a callback registered for the *pre_inst* event. It gets called for each instruction that MAMBO scans, but before the translation has been inserted in the code cache, which allows the plugin to insert its own code before the translated code.

Another callback is registered for the thread creation event *pre_thread*, which gets called when a new thread is created (including the initial thread) and allocates a new thread-private counter, saving the pointer in the thread-private metadata. A third callback is registered for the thread exit event *post_thread*, which is called when a thread or the whole process exit, and prints the value in the counter.

REFERENCES

- Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGPLAN Notices* 41, 11, 2–13.
- ARM. 2015. *ARM[®] Architecture Reference Manual, ARMv8, for ARMv8-A Architecture Profile*. <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0487a.i/index.html>.
- Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A transparent dynamic optimization system. In *ACM SIGPLAN Notices*, Vol. 35. ACM, New York, NY, 1–12.
- Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*. 41–46.
- Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University, Princeton, NJ.
- Darrell Boggs, Gary Brown, Nathan Tuck, and K. S. Venkatraman. 2015. Denver: Nvidia’s first 64-bit ARM processor. *IEEE Micro* 2 (2015), 46–55.
- Derek Lane Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.
- Thurston H. Y. Dang, Petros Maniatis, and David Wagner. 2015. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS’15)*, Vol. 15.
- James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The transmeta code morphingTM software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE Computer Society, 15–24.
- Kim Hazelwood and Artur Klauser. 2006. A dynamic binary instrumentation engine for the ARM architecture. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, New York, NY, 261–270.
- John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4, 1–17.
- Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. 2012. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the 10th International Symposium on Code Generation and Optimization*. ACM, New York, NY, 104–113.
- Ho-Seop Kim and James E. Smith. 2003. Hardware support for control transfers in code caches. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 253.
- Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, Vol. 40. ACM, New York, NY, 190–200.
- Tipp Moseley, Daniel A. Connors, Dirk Grunwald, and Ramesh Peri. 2007. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 4th International Conference on Computing Frontiers*. ACM, New York, NY, 143–152.
- Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN Notices*, Vol. 42. ACM, New York, NY, 89–100.
- Mathias Payer and Thomas R. Gross. 2010. Generating low-overhead dynamic binary translators. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*. ACM, New York, NY, 22.
- Yukinori Sato, Yasushi Inoguchi, and Tadao Nakamura. 2011. On-the-fly detection of precise loop nests across procedures on a dynamic binary translation system. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*. ACM, New York, NY, 25.
- Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*. 17–30.

- Swaroop Sridhar, Jonathan S. Shapiro, and Prashanth P. Bungale. 2007. HDTrans: A low-overhead dynamic translator. *ACM SIGARCH Computer Architecture News* 35, 1, 135–140.
- Jon Watson. 2008. Virtualbox: Bits and bytes masquerading as machines. *Linux Journal* 2008, 166, 1.
- Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman Amarasinghe. 2011. Dynamic cache contention detection in multi-threaded applications. In *ACM SIGPLAN Notices*, Vol. 46. ACM, New York, NY, 27–38.

Received August 2015; revised December 2015; accepted January 2016