Optimising Dynamic Binary Modification Across ARM Microarchitectures

Cosmin Gorgovan
School of Computer Science
The University of Manchester
cosmin.gorgovan@manchester.ac.uk

Amanieu d'Antras
School of Computer Science
The University of Manchester
amanieu@amanieusystems.com

Mikel Luján
School of Computer Science
The University of Manchester
mikel.lujan@manchester.ac.uk

ABSTRACT

Dynamic Binary Modification (DBM) is a technique for modifying applications transparently while they are executed, working at the level of native code. However, DBM introduces a performance overhead, which in some cases can dominate execution time, making many uses impractical.

The ARM hardware ecosystem poses unique challenges for high performance DBM systems because of the large number and wide range of capabilities of the commercially available implementations: from single issue, in order cores up to 6-issue out-of-order cores and including less traditional implementations. These variations raise the question of whether it is possible to develop DBM optimisations which either improve or, at the very least, do not affect performance on all available systems and microarchitectures. To answer this question, the performance of three new optimisations for the MAMBO DBM system has been evaluated on five systems using different microarchitectures. For comparison, the overhead of DynamoRIO, a high performance DBM system which was recently ported to the ARM architecture, is also evaluated.

KEYWORDS

Dynamic Binary Modification; Dynamic Binary Instrumentation

ACM Reference Format:
Cosmin Gorgovan, Amanieu d'Antras, and Mikel Luján. 2018. Optimising Dynamic Binary Modification Across ARM Microarchitectures. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3184407.3184425

1 INTRODUCTION

Dynamic Binary Modification (DBM) is a technique for modifying applications transparently while they are executed, working at the level of native code. DBM has numerous applications, some of the more common being dynamic instrumentation [23, 27], program analysis [26, 30], virtualisation [1, 28] and Dynamic Binary Translation (DBT) [8, 9, 13].

The ARM hardware ecosystem poses unique challenges for high performance DBM systems because of the large number and wide range of capabilities of the commercially available implementations: from single issue, in-order cores (Cortex-A5), up to out-of-order cores (Cortex-A17 or Applied Micro X-Gene).

These challenges are exacerbated by the wide adoption of single-ISA heterogeneous multicores (such as *big.LITTLE* [3]), which use different microarchitectures (e.g. a cluster of energy efficient in-order cores and a cluster of high performance out-of-order cores) in the same System on Chip (SoC) and allow the migration of active applications from one type of core to another. This raises the question of whether it is possible to develop DBM optimisations which either improve or, at the very least, do not affect performance on all ARM systems and microarchitectures.

MAMBO [17] is an open source [16], DBM framework for the ARM architecture. To further reduce its overhead, three optimisations are proposed and evaluated in this paper. The performance of these new optimisations and of the baseline MAMBO system has been measured on five ARM systems which use different microarchitectures.

The overhead of the baseline MAMBO system is partly caused by microarchitectural inefficiencies, for example by a high number of instruction cache misses [17]. Therefore, the optimisations presented in this paper aim to address this limitation by improving performance at the microarchitectural level (e.g. by reducing the number of cache misses) rather than at the architectural level (e.g. by reducing the number of executed instructions).

The contributions of this paper include:

- a trace system for MAMBO which reduces its overhead by improving code cache locality and eliminating some of the branches on the hot code path, while avoiding software branch target prediction for poorly predictable branches;
- a novel scheme to enable hardware return address prediction in a code cache without use of a software return address stack (Hardware-assisted return address prediction);
- a software indirect branch prediction scheme which allows effective prediction for polymorphic indirect branches (Adaptive Indirect Branch Inlining);
- evaluating the effectiveness of these optimisations when running on a wide range of microarchitectures, including a comparison against the state of the art; and
- reducing the geometric mean overhead of the MAMBO DBM system running SPEC CPU2006 by 27% - 54% on the five evaluation systems.

This work was supported by UK EPSRC grant PAMELA EP/K008730/1. Mikel Luján is supported by a Royal Society University Research Fellowship.

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184425

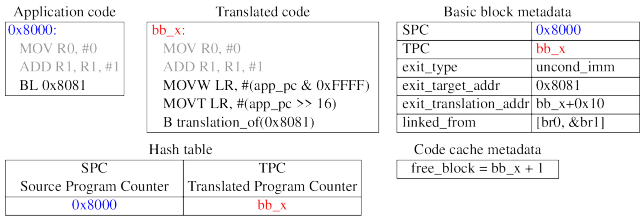


Figure 1: Example basic block and associated data structures.

The rest of the paper is organised as follows. Section 2 is a short description of the baseline MAMBO system. Section 3 describes the newly introduced trace system. Section 4 describes the new optimisations for indirect branches. Section 5 is the performance evaluation and Section 6 draws the final conclusions.

2 BASELINE SYSTEM OVERVIEW

The baseline MAMBO system was described by Gorgovan *et al.* [17]. MAMBO, like most other DBM tools, runs in the same process with the application it modifies and controls its execution by scanning, translating and optionally modifying all code before execution. The process of code discovery, translation and modification is done at the level of *Basic Blocks* (BBs), which are single-entry, single-exit units of code. To amortise the cost of this process, the result is stored in a *code cache* and reused for future executions. MAMBO uses thread-private code caches (and associated data structures), which allow multithreaded code scanning and execution with minimal synchronisation. A hash table is used to map application addresses (Source Program Counters - *SPCs*) to their translation in the code cache (Translated Program Counters - *TPCs*). To minimise the cost of handing over execution from one basic block to another, basic blocks which end with a direct branch are linked directly (using direct branches inside the code cache). Additionally, indirect branches are translated to inline hash table lookups, which perform a lookup of the TPC corresponding to each SPC used by the application, with minimal spilling and restoring of application registers, as opposed to a full context switch to the DBM system.

Figure 1 shows an example basic block and a simplified view of the associated data structures. The application code, at address `0x8000`, contains two data processing instructions and a branch with link (procedure call) to another location. The translation in the code cache, in basic block `bb_x`, contains the unmodified data processing instructions (grey) and the translation of the branch with link instruction (black). The translated branch with link hides that the code is running from a code cache, at a different location than expected, by explicitly setting the value of the Link Register (LR) to the correct value and then executing a regular branch (without link) to the translation of the target address. Some of the metadata created for the example BB is shown in the right hand column: its SPC, TPC, the location, type of branch and its target (for direct branches) in a structure specific to each BB; the space used by the translation in the code cache is marked as used (in the *code cache metadata*); and an entry is added to the thread-private hash table to map the SPC to the TPC. Branches between basic blocks are tracked by maintaining a linked list of *incoming* branches for each basic block (the `linked_from` metadata field).

3 TRACES

The baseline code cache, organised in basic blocks, creates and stores the basic blocks in the order they are first executed. However, the basic blocks in the software code cache have high fragmentation, making inefficient use of the hardware code cache. Furthermore, the two paths of conditional branches are translated in two separate basic blocks in the software code cache, increasing the number of executed branches (by executing a branch in the translated code even when the source conditional branch is not taken). To avoid these limitations, this paper introduces traces (also known as *superblocks*) to MAMBO, which are single-entry, multiple-exit units built by merging together the basic blocks on the hot code path. The single-entry, single-exit units which make up a trace are called *trace fragments*.

Because creating a trace has a non-trivial cost (both in terms of code cache space, and execution time spent creating the trace instead of running the application), it is important to only create traces for hot code, which is expected to execute many times in the future and amortise its creation cost. On the other hand, to get the best performance, it is preferred to create traces for all of the hot code in an application and as early as possible. The challenge is in 1) quickly identifying the hot code in an application and 2) in profiling the hot execution paths through this code with low overhead. MAMBO builds traces using an improvement of the Next Executing Tail (NET) online profiling scheme [15]. The NET algorithm is summarised in Table 1. It is designed to minimise the profiling overhead. Towards that end, NET initially maintains an execution counter only for the basic blocks which are the potential start of a hot path. These instrumented basic blocks are called *trace heads*. The insight is that the hot execution path must consist of cycles, therefore NET uses the targets of backwards branches (both direct and indirect) as trace heads. Once the execution counter for a particular trace head reaches a certain threshold, then the trace is considered hot and NET records the full execution path following the trace head, until a backwards branch is encountered (which *terminates* the trace). This recorded path is then used as the predicted path, based on the rationale that the *trace tail* following a hot trace head is also likely to be part of the hot execution path. For example, let us consider the Control Flow Graph (CFG) depicted in Figure 2, where each box represents a basic block and block *A* ends with a conditional direct branch, blocks *B*, *D*, *E*, *F*, *G*, *H* and *I* end with unconditional direct branches, while block *C* ends with an unconditional indirect branch. Using the NET trace head selection algorithm, the trace heads in this example would be the two blocks which are the target of backwards branches: *A* and *C*. If, for example, the execution count threshold would then be reached for the trace head A and then the blocks *CEH* would execute, the

Hot code profiling	execution counter for *trace heads*
Trace head selection	the targets of backward branches
Trace path	the path taken across forward direct and indirect branches, after the execution counter of a trace head reached a certain threshold
Trace termination	a backward branch is encountered

Table 1: Overview of the NET algorithm.

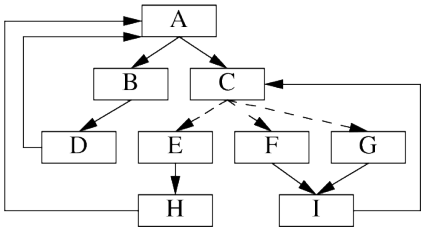


Figure 2: Example control flow graph. Each box represents a basic block. Block A, the entry point, contains a conditional direct branch, block C contains an unconditional indirect branch and all other blocks contain unconditional direct branches.

trace would consist of the blocks *ACEH*, ending with a branch back to the beginning of the trace.

An important property of NET is that it builds traces across indirect branches, statically predicting their target address to be the same as observed in the path recording phase. In the previous example involving the trace *ACEH*, the target of the indirect branch from block *C* is block *E* in the path recording stage, therefore NET builds the trace predicting that the target of block *C* is always *E*. However, analysis of the SPEC CPU benchmarks showed that most indirect branches are polymorphic and poorly predicted by a static target predictor, as used by NET. Furthermore, a static indirect branch predictor adds overhead in the case when the prediction is incorrect. This analysis is available in Section 4.2, which also presents AIBI, a more accurate indirect branch prediction scheme, which has been implemented in MAMBO. To avoid this limitation, the MAMBO trace building scheme terminates on indirect branches, which avoids static target prediction and instead allows their SPC-to-TPC lookup to be implemented using an inline hash table lookup, optionally with Adaptive Indirect Branch Inlining (Section 4.2). However, this change to NET has a number of side-effects which must be managed to maintain good performance, as discussed in the following subsections.

3.1 Trace head selection

The new trace termination condition described in Section 3 avoids adding the targets of an indirect branch to a trace tail, by terminating the trace. However, one or more of these targets are likely part of the hot execution path, therefore all targets of indirect branches should then have execution counters (i.e. become trace heads) to allow the creation of traces. Nevertheless, the NET trace head selection algorithm only instruments the targets of backwards branches and would generally fail to instrument many of these targets. If, for example, block *C* in the CFG shown in Figure 2 is on the hot code path and its indirect branch has a 70% bias toward block *E*, 30% toward block *F* and never branches to block *G*, then both the *E* and *F* blocks are also on the hot code path. If these blocks would be trace heads, then the traces *EH...* and *FI...* would be created. Nevertheless, the unmodified trace head selection of NET does not allow this and instead the blocks *E*, *H*, *F* and *I* could not be trace heads, nor would they be included in trace tails because of the additional termination condition used by MAMBO.

NET also presents an implementation challenge for DBM systems: if a basic block is first reached using a forward branch, then it will be created without an execution counter. However, if it is later reached using a backward branch, then an execution counter has to be added to the existing block or, otherwise a second version of the basic block has to be created. Both options make inefficient use of the code cache space and increase fragmentation. For example, in the control flow graph depicted in Figure 2, the first execution of block *C* would necessarily be a result of the branch from block *A*, therefore not creating a trace. If block *I* would execute at a later time, then the backwards branch to *C* would be discovered and an execution counter would have to be added to the existing block *C*.

Both of these issues are addressed in MAMBO by a single change to the trace head selection algorithm: whether a basic block is a trace head is decided at the time it is scanned, depending on whether it ends with a direct branch (then it is a trace head) or an indirect branch (then it is a regular basic block). Basic blocks containing an indirect branch are not allowed as trace heads because they would be terminated immediately and would therefore create traces containing a single fragment. This algorithm also allows the targets of indirect branches to be trace heads and avoids ulterior transformation of existing basic blocks into trace heads, by removing the reachability of basic blocks as an input to the trace head selection algorithm. Instead, it relies exclusively on the contents of the basic block itself, which are known at the time it is scanned. In the example CFG in Figure 2, all basic blocks apart from block C (which contains an indirect branch) would be trace heads.

Changing the trace head selection algorithm compared to NET results in more basic blocks becoming trace heads and incurring the overhead of updating the execution counter. However, this overhead is limited: the counter is updated by calling a shared procedure, which is implemented using only ten instructions. Additionally, the execution count threshold for trace creation is low, typically in the order of tens or hundreds, which strongly limits the maximum overhead that can be introduced by each trace head. Compared to scanning the trace head and translating it in the code cache, repeatdly incrementing the execution counter up to its threshold is relatively fast.

In MAMBO, a trace head is implemented as a basic block with a header (shown in Listing 1) which: 1) pushes to the stack the contents of 3 scratch registers and of the Link Register, 2) sets the id of the trace head in R0 and 3) calls a shared procedure which then decrements the execution counter of the trace head by one and returns, until it reaches zero. When zero is reached, trace creation is started, using the id passed to the shared procedure to identify the trace head. The rest of the trace building process is described in Section 3.2.

```
PUSH {R0-R2, LR}
MOVW R0, #(trace_head_id & 0xFFFF)
MOVT R0, #(trace_head_id >> 0xFFFF)
BL increment_exec_counter
```

Listing 1: The code added to trace heads.

	NET	MAMBO traces
Hot code profiling	execution counter for *trace heads*	same as NET
Trace head selection	the targets of backward branches	basic blocks exiting with a direct branch
Trace path	the path taken across forward direct and indirect branches, after the execution counter of a trace head reached a certain threshold	the path taken across direct branches, after the execution counter of a trace head reached a certain threshold
Trace termination	a backward branch is encountered	an indirect branch is encountered OR a direct branch to an existing trace is encountered OR the maximum number of fragments has been reached and a backward direct branch is encountered

Table 2: Comparison of MAMBO traces and NET.

3.2 Trace building

Trace building works similarly to NET: when a trace is first created, the SPC of the trace head is used to create the first fragment in the trace. Then this fragment is executed and its selected target is appended to the trace. This process continues iteratively until a termination condition is met. The first such condition is the execution of an indirect branch, as previously discussed. An additional condition is the execution of a direct branch to the entry point of an existing trace (including itself), which is intended to limit *tail duplication* between different traces. If a branch to the entry point of an existing trace is encountered, then a direct branch to that trace is inserted and the partial trace is terminated. For example if a trace was created from block *A* in Figure 2, then the trace would initially contain the fragment *A*. After the fragment *A* would execute, its target would be appended to the trace. If this target was *B*, then the partial trace would contain the fragments *AB*. Since *B* contains a branch to *D*, this fragment would also be added to the trace, which would then contain *ABD*. Finally, the target of the *D* fragment is *A*, for which a trace would already exist (the partial trace itself). The *ABD* trace would be terminated and linked directly to its own entry point.

Additionally, when a trace is created, the SPC-TPC hash table is updated to the TPC of the trace. All direct branches from other basic blocks and traces to the trace head are replaced by branches to the new trace, essentially making the trace head unreachable. In the previous example, the hash table entry for the SPC of *A* would be changed from the address of the *trace head A* to the address of the new *partial trace A*.... Similarly, any branches to *trace head A* would be replaced with branches to the partial trace.

3.3 Trace size limits

Some code duplication is allowed inside each trace, to encourage partial unrolling of short loops. However, excessive code duplication is undesirable, therefore the maximum number of fragments in each trace is limited. If this configurable limit is reached, the trace is terminated on its next backwards branch. For example in the CFG shown in Figure 2, the blocks *CFI* form a loop. If this loop would execute while the trace *ACFICFICFI*... was built, then this would result in an increasingly large trace, which would eventually fill the trace code cache. However, because the maximum number of fragments in a trace is limited, the trace would be terminated on the backward branch from *I* to *C* after a limited number of iterations.

3.4 Summary

Using a software code cache based on basic blocks contributes to the overhead of DBM systems by introducing fragmentation and by executing numerous branch instructions to transfer control between any two basic blocks. These issues are mitigated by traces, which are single-entry and multiple-exit units which group together the basic blocks likely to execute sequentially on the hot code path. The main challenges related to traces are in 1) identifying the hot code with minimal delay and 2) profiling this code to obtain the hot execution paths. The NET online profiling algorithm is commonly used to build traces in DBM systems, however it relies on static target prediction for indirect branches. Nevertheless, indirect branches are shown to generally be polymorphic and poorly predicted by a static target predictor. In this context, several changes to NET are proposed, as summarised in Table 2, which eliminate static indirect branch prediction while managing the undesired side-effects.

4 INDIRECT BRANCHES

Indirect branches are control flow instructions with a target not known at translation time. Looking up TPC for the SPC of indirect branches at runtime is the major source of overhead for DBM systems [21]. We classify indirect branches in three types:

- function returns, for which we introduce *hardware-assisted return address prediction* in Section 4.1; and
- generic indirect branches, handled in MAMBO using inline hash table lookups, for which we introduce the optional *adaptive inlining* - Section 4.2; and
- table branches, handled in MAMBO using the *space-efficient shadow branch table linking* [17].

Figure 3 shows the steps involved in an inline hash table lookup, which is the mechanism used for handling indirect branches in the baseline MAMBO: 1) first, if required and depending on the type of indirect branch, the values of up to three registers are pushed onto the stack to enable their use as scratch registers; then, 2) the SPC

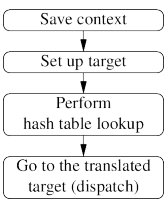
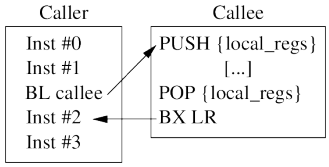


Figure 3: Inline hash table lookup.

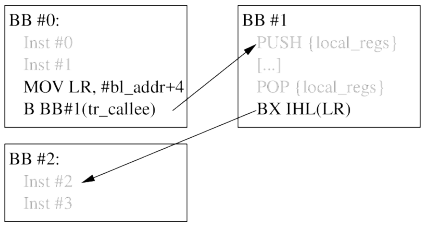
is copied or generated in one of the scratch registers; 3) the hash table lookup is performed, with the TPC being loaded; and 4) finally the values of the scratch registers are restored and a branch to the TPC is performed. The *hardware-assisted return address prediction* and *adaptive indirect branch inlining* optimisations are both an extension to inline hash table lookups.

4.1 Hardware-assisted return address prediction

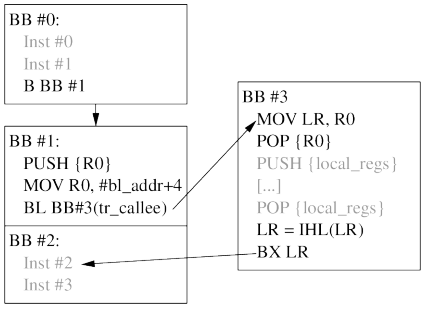
Return instructions are the instructions which execute at the end of a procedure (the callee) to return control back to the caller. More specifically, returns target the instruction immediately following the call instruction. Therefore, at the time a call is executed, the target of the first return to execute can be accurately predicted to be the address of the instruction following the call. If nested calls execute, then all predicted addresses can be recorded in a Last In, First Out (LIFO) structure for later use. These properties are used for return address prediction in virtually all modern microprocessors, including by most ARM implementations, which maintain a Return Address Stack (RAS) which is not exposed architecturally [2, 4–6]. However, the translated code generated by a DBM system does not generally maintain these properties because call instructions are translated to regular branches while returns are translated to regular indirect branches. Consequently, hardware return address prediction is not used. Instead, return instructions are predicted by the hardware using the generic indirect branch prediction mechanisms, which are both less accurate and also limited in the number of indirect branches which can be tracked and predicted simultaneously. Since fast return handling is critical for achieving low overhead in DBM systems [21], this limitation is an important contributor to the total overhead.



(a) The original function call.



(b) The translated function call without hardware return prediction.



(c) The translated function call with hardware return prediction.

Figure 4: Example of a typical function call.

Figure 4(a) shows a typical function call in ARM code. A *caller* function contains a call (implemented using a Branch-and-Link - BL - instruction) to the entry address of the *callee*. The callee preserves the return address from the Link Register (LR), executes, and then returns to it using a return instruction (a Branch-and-eXchange - BX - instruction using the address in the LR, in this example). Because the target address of the return is in a register, this return instruction is an indirect branch.

Hardware return address prediction on ARM works thus: when a call (either a BL or a Branch-with-Link-and-eXchange - BLX - instruction) is executed, an entry, containing the address of the next instruction after the call, is automatically pushed by the core on the hardware RAS. Then, when the matching return instruction is executed, its target address is predicted by automatically popping the first value from the top of the RAS. Since the ARM architecture does not have explicit return instructions, certain types of indirect branches (*return-type instructions*) are treated by the branch predictor as returns, typically: *BX LR*, a *POP* containing the PC in the register list, a SP-relative load into PC, and *MOV PC, LR*.

The naive translation of BL and BLX instructions (from the native code in Figure 4(a) to Figure 4(b)) emulates the call instruction by setting the value of the LR explicitly to the SPC of the instruction following the call and then branches to the translation of the target using a regular (i.e. without link) branch. Similarly, return instructions are translated to an inline hash table lookup (represented by the `IHL()` pseudocode) followed by a regular branch (`BX`) to the TPC of the return address. Therefore, the naive translation of calls and returns is not compatible with the hardware return address predictor, which increases branch mispredictions by 1) translating call-type instructions to regular branch instructions, which do not cause a push on the RAS and by 2) translating return-type instructions to generic indirect branches, which are predicted using the less accurate indirect branch predictor, while also increasing the pressure on the indirect branch predictor. We propose *hardware-assisted return address prediction* to solve these issues, by modifying the translations as shown in Figure 4(c): first, it translates call-type instructions to a sequence which ends with a call-type instruction (`BB #1`), which allows the hardware predictor to push an entry to the RAS. Next, it inserts the translation of the following instructions, i.e. the predicted return (`BB #2`) immediately after the call, as expected by the predictor. Finally, it modifies the translation of return-type instructions to use a return-type instruction, which will allow the hardware predictor to pop the predicted address from the RAS (`BX LR` in `BB #3`).

For return prediction to work correctly, a single translation of each call-type instruction must exist in the code cache, otherwise multiple translations of the predicted return would be generated, which cannot be registered in the hash table mapping the SPC-TPC

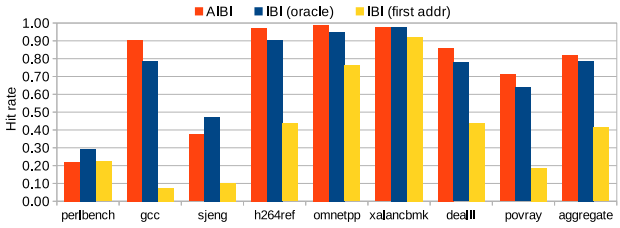


Figure 5: Comparison of hit rates on a selection of SPEC CPU2006 benchmarks for indirect branch predictors.

relationships. This is a potential issue because different entry points into a single linear code area which contain a call-type instruction would normally lead to the creation of multiple basic blocks, each one containing a translation of the call-type instruction. To avoid this issue, if a call-type instruction is scanned without being the first instruction in a basic block, a new basic block is created with the call-type instruction as the entry point, if it does not exist yet. The original basic block is then directly linked to the translation of the call-type instruction. This ensures that when a call-type instruction is scanned, its SPC-TPC mapping will be recorded. Then, if the same call-type instruction is encountered in multiple BBs by the code scanner, all are linked to the unique translation.

As an additional optimisation, when a call-type instruction is not the first one in a basic block and its translation does not exist yet, the separate basic block generated for the call-type instruction will be stored in the code cache area immediately following the first basic block, allowing the eliding of the direct branch. For the example in Figure 4(c), `BB #1` would be stored immediately after `BB #0`, allowing the elimination of the `B BB#1` instruction from `BB #0`.

4.2 Adaptive indirect branch inlining

Indirect branches have dynamic targets, which are not known at translation time. Due to their nature, the translated indirect branches must perform a SPC to TPC lookup every time they execute. This lookup represents a major source of overhead for DBM systems [19, 21]. The baseline version of MAMBO and other DBM systems such as DynamoRIO [10] attempt to reduce this overhead by generating a highly optimised inlined hash table lookup routine for each translated indirect branch. This approach allows the hardware branch predictors to handle separately each translated indirect branch (improving hardware branch prediction rates) and minimises the length of the critical path compared to a shared routine by taking advantage of the available dead registers, on a case-by-case basis. However, the hash table lookup operation inherently requires a number of additional instructions, including memory loads and conditional branches. Other DBM systems, such as Pin for ARM [18], use Indirect Branch Inlining (IBI) [7], which consists of a compare-and-branch chain which compares the current target address against a configurable number of previous targets, using only the code path (i.e. by using immediates). However, previous attempts to use this prediction scheme in MAMBO have failed to improve performance, due to the high overhead associated with updating the predicted target, the poor hit rate due to the polymorphic nature of indirect branches and the high penalty of hardware branch mispredictions triggered in the relatively common case when one or more predictions at the top of the chain miss. We designed the *Adaptive Indirect Branch Inlining* (AIBI) scheme to allow quick updating of the predicted address after every misprediction, while still having a shorter critical path than the inline hash table lookup. This is similar to the way indirect branch target prediction works in most hardware implementations.

Figure 5 compares the hit rates for three indirect branch predictions schemes: *AIBI*, which always predicts the address of the most recent target; *IBI (common)*, which is a static predictor which predicts the most common target for each branch using post-mortem information; and *IBI (first)* which predicts the address of the first target seen for each branch. The selected benchmarks are those which execute a relatively high number of generic indirect branches. The *aggregate* bars show the hit rates when considering together all indirect branch executions from the selected benchmarks. IBI (common) shows the upper bound for a static predictor and since the information to choose the most common target is not available at runtime, practical IBI implementations will almost always have lower hit rates. The IBI (first) hit rate is more relevant for practical IBI implementations, which can either predict the target of the n-th execution of an indirect branch, or, alternatively, can profile the first few executions of the branch and predict the most common target among those samples. It can be observed that the hit rate for AIBI is generally similar to that of the IBI (common) predictor and for most benchmarks and overall, slightly better. On the other hand, the hit rate for IBI (first) is generally much lower, which indicates that practical IBI implementations will tend to have lower hit rates than AIBI.

A major difference of AIBI compared to IBI is that the prediction is updated for every miss, which is achieved by falling back to the inline hash table lookup and unconditionally overwriting the prediction on this execution path. Since this can occur for a large percentage of the executions of a branch, this operation must be implemented very efficiently to minimise the overhead of prediction misses. Using immediates on the code path to generate the predicted address (similar to IBI) was ruled out because ARM uses a modified Harvard architecture, which requires expensive cache flushing and invalidation via system calls to update code. Therefore, the predicted target address and its matching code cache address are accessed as data words, which is the second major difference from IBI. The addition of two unconditional store instructions with no read-after-write dependencies on the fallback execution path appears to have a minimal performance impact on most hardware implementations.

The diagram in Figure 6 shows how AIBI works, where the boxes with a solid border show the additional steps added specifically for AIBI, while the boxes with a dashed border show the unmodified steps which are part of the inline hash table lookup routine (which is shown separately in Figure 3). Listing 2 shows the implementation of AIBI. With AIBI, after the target address has been generated or loaded in a register, the predicted SPC is loaded using a single PC-relative load instruction and then two addresses are compared, as shown in the `check_pred` procedure. The comparison is implemented using a subtract instruction (`SUB`) and a Compare and Branch on NonZero (`CBNZ`) instruction to preserve the flags in the ARM Program Status Register (PSR). In case of a match, the

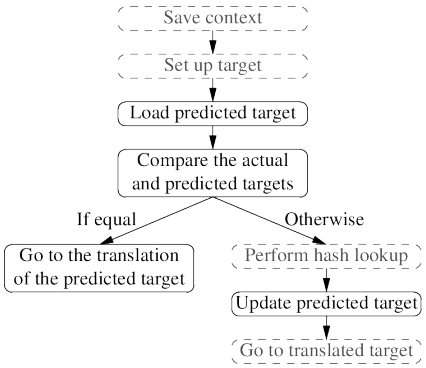


Figure 6: Adaptive indirect branch inlining.

```
check_pred:
    LDR Rs0, [PC, #16]
    SUB Rs0, Rs0, Rtarget
    CBNZ fallback
b_pred:
    POP {Rs0, ..., Rsn}
    LDR PC, [PC, #4]
pred_spc: .word
pred_tpc: .word
fallback:
    ; the fallback inline hash table lookup
    SUB Rs0, PC, offset_to_pred_spc
    STR Rtarget, [Rs0, #0]
    ; the TPC is loaded from the hash table in Rtarget,
    ; overwriting the SPC
    STR Rtarget, [Rs0, #4]
    ...
```

Listing 2: The implementation of AIBI. `Rs0` to `Rsn` are scratch registers, while `Rtarget` is the register which initially contains the target address (SPC).

context is restored and execution branches to the predicted TPC using a second PC-relative load, as shown in the `b_pred` procedure. Otherwise, in case of a miss, the regular inline hash table lookup proceeds, with the difference that after the hash table lookup has been performed, but before branching to the destination, the predicted SPC and TPC are updated, as shown in the `fallback` procedure. PC-relative stores are not allowed in the Thumb mode, therefore the address where the predicted SPC and TPC are stored is first generated using a subtract instruction.

AIBI is similar in predicting the address of the most recent target to the MRU IBI prediction scheme proposed by Dhanasekaran and Hazelwood [14]. However, while the MRU scheme is used in *addition* to IBI, AIBI is an alternative to IBI. When the MRU prediction misses, it falls back to IBI, while AIBI falls back to an inline hash table lookup. MRU updates the predicted address from the IBI target fragments, while AIBI updates the predicted address in the inline hash table lookup. Furthermore, AIBI as implemented in MAMBO is effective in reducing the overhead on all systems used in the evaluation (Section 5), while MRU as prototyped for Pin failed to improve performance on average [14]. Unfortunately, insufficient information is available to determine why. The MRU publication explains that its dynamic instruction count was higher than that of standard IBI despite the increased prediction hit rate. However, on ARM platforms we have observed that the hardware branch prediction rate and other microarchitectural events often have a stronger effect than relatively small changes in the number of executed instructions. For example, when we have implemented IBI in MAMBO, the dynamic instruction count was significantly reduced, however the overhead was increased because of the hardware branch mispredictions introduced by the IBI chain. This could indicate that 1) existing x86 implementations can predict IBI chains better than ARM implementations or, less likely, 2) that branch mispredictions are relatively cheaper on x86 implementations than on ARM implementations. Another possible explanation is that the performance of MRU was affected by the mechanism used to update the predicted address, which it duplicates across every target fragment linked by the IBI chain and whose details are not presented in the publication.

5 EVALUATION

5.1 Experimental setup

Table 3 describes the microarchitectures of the five different systems used for evaluation. All systems use a modified Harvard architecture, with separate 32 KiB L1 data caches and 32 KiB L1 instruction caches, and separate data and instruction L1 TLBs as described in Table 3. Higher level caches and TLBs are unified. The *IB predictor* row describes the hardware indirect branch prediction scheme: *previous* means that the address of the previous target of the instruction is predicted, while *adaptive* means that multiple target addresses can be predicted for each branch instruction.

All systems are running Ubuntu 14.04 LTS with the Linux kernel version supported by the manufacturer: 3.8 for ODROID-X2, 3.10 for ODROID-XU3, Tronsmart R28, Jetson TK1, and 4.2 for APM X-C1. SPEC CPU2006 has been compiled with GCC 4.6.3, configured to generate Thumb-2 code (the default configuration) for the *armhf* architecture using the *-O2* optimisation level and the executables were statically linked. Power management features such as DVFS and core offlining were disabled. The ODROID-XU3 system uses a heterogeneous *big.LITTLE* [3] configuration, with a *LITTLE* Cortex-A7 cluster, which was used for this evaluation and a *big* Cortex-A15 cluster which was not benchmarked because the same ARM core is used on the Jetson TK1 system.

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 and produce the expected output. All SPEC CPU2006 results were obtained using the *ref* data set.

Multiple MAMBO *configurations* have been benchmarked. A configuration is a build of MAMBO with a specific set of enabled optimisations. The configuration with an empty set of optional optimisations enabled is called the *baseline* configuration. This is similar to the MAMBO configuration used by Gorgovan *et al.* [17], with the exception that the *low overhead return address prediction*, which is a return address prediction scheme based on a software RAS, has been disabled because it is incompatible with traces and therefore it is never used in this evaluation. *Hardware-assisted return address prediction*, introduced in this publication, serves a similar role while maintaining full transparency. All other configurations are named *+<name of optimisation 0> ... +<name of optimisation n>*,

System	ODROID-XU3[a]	ODROID-X2	Tronsmart R28	Jetson TK1	APM X-C1
SoC	Exynos 5422	Exynos 4412 Prime	Rockchip RK3288	NVIDIA T124	APM883208
Core	Cortex-A7	Cortex-A9	Cortex-A17	Cortex-A15	X-Gene 1
Frequency	1.4 GHz	1.7 GHz	1.6 GHz	2.3 GHz	2.4 GHz
L2 cache size	512 KiB	1 MiB	1 MiB	2 MiB	256 KiB
L3 cache size	N/A	N/A	N/A	N/A	8 MiB
L1i line length	32	32	64	64	64
L1d line length	64	32	64	64	64
L2 line length	64	32	64	64	64
L1d TLB	10	32	32	32(R) + 32(W)	20
L1i TLB	10	32	32	32	10
L2 TLB	256	132	1024	512	1024
IB predictor	previous[b]	previous	previous	adaptive	adaptive
OOO	N	Y, 2-issue	Y, 2-issue	Y, 3-issue	Y, 4-issue
Pipeline len	8	8-11	10-12	15	15

Table 3: Overview of the systems used for evaluation.

[a]The specifications for ODROID-XU3 apply to the *LITTLE* cluster only. The *big* cluster was not used for this evaluation because it uses the same microarchitecture as the Jetson TK1 system.

[b]Cortex-A7 is documented not to predict the target for branches implemented as loads or data processing operations with PC as the destination, which are used by MAMBO in the translation of most indirect branches.

for example the configuration with *hw_ras* and *traces* enabled is named *+hw_ras +traces*. The following optional optimisations have been evaluated:

- *traces* - code cache traces;
- *hw_ras* - hardware-assisted return address prediction; and
- *aibi* - adaptive indirect branch inlining.

As described in Section 3, traces are created when a trace head reaches a predefined execution count threshold. Our experiments on the SPEC CPU2006 benchmarks have shown that the performance of longer running tasks is not affected by setting a relatively high threshold. However, significant trace cache space savings can be obtained. Therefore, the trace creation threshold for this evaluation was set to 256, the maximum allowed by the implementation.

For comparison, we have also evaluated DynamoRIO [10], the only other maintained and publicly available low overhead DBM system for ARM. We used the git commit *38950ce2* from 19th of January, 2017. Note that DynamoRIO does not implement the hot code tracing optimisation for 32-bit ARM, the architecture used in this evaluation.

		SPEC suite		
Hardware platform	DBM system	int	fp	CPU
ODROID-XU3 (LITTLE)	baseline	1.55	1.11	1.26
in-order Cortex-A7	+traces	1.41	1.10	1.21
	+hw_ras +traces	1.36	1.09	1.19
	DynamoRIO	1.68	1.21	1.38
ODROID-X2	baseline	1.61	1.13	1.30
OOO Cortex-A9	+traces	1.33	1.07	1.17
	+aibi + traces	1.31	1.06	1.15
	DynamoRIO	1.65	1.16	1.34
Tronsmart R28	baseline	1.60	1.12	1.29
OOO Cortex-A17	+traces	1.31	1.09	1.17
	+aibi +traces	1.29	1.08	1.16
	DynamoRIO	1.71	1.26	1.42
Jetson TK1	baseline	1.71	1.16	1.35
OOO Cortex-A15	+traces	1.44	1.11	1.23
	+hw_ras +traces	1.38	1.11	1.21
	DynamoRIO	1.67	1.22	1.38
APM X-C1	baseline	1.59	1.09	1.26
OOO X-Gene1	+traces	1.34	1.07	1.17
	+hw_ras +traces	1.23	1.05	1.12
	DynamoRIO	1.64	1.18	1.34

Table 4: The slowdown of MAMBO baseline, *+traces*, the configuration with the lowest overhead and DynamoRIO for SPEC CPU2006 on each system.

5.2 Overall performance

Table 4 summarises the overall performance of the baseline, *+traces*, the optimal MAMBO configuration and of DynamoRIO for each system (when running SPEC CPU2006), while Figures 7 to 11 show the detailed results for each benchmark. The values reported in the table are the geometric mean of execution time relative to native execution for each set of benchmarks. It can be observed that between the five test systems, two unique MAMBO configurations are needed to achieve the lowest possible overhead. This hints that, as expected, some of the optimisations have varying effectiveness depending on the microarchitecture. Another related observation is that the spread of the average overhead between the microarchitectures is quite high: from only 12% on APM X-C1, up to 21% on Jetson TK1, which further underlines the impact of microarchitecture on the performance of DBM systems. The SPECint benchmarks run with higher overhead than the SPECfp benchmarks because they tend to be control (as opposed to data) bound.

The *traces* optimisation has by far the largest overall effect. This is the expected result, as improved software code cache locality and a reduced number of executed branches reduce the overhead 1) for most benchmarks and 2) on all microarchitectures. While the geometric mean overhead is generally reduced only by a few points for the other optimisations, this is in large part due to these optimisations targeting only specific types of workloads. For example, the *hw_ras* optimisation reduces the overhead of *xalancbmk* on APM X-C1 from 96% to 66%, however, because only a few benchmarks gain a speed-up, the geometric mean overhead is only decreasing from 17% to 12%. By running the optimal configuration on each system, the geometric mean overhead is reduced compared to the

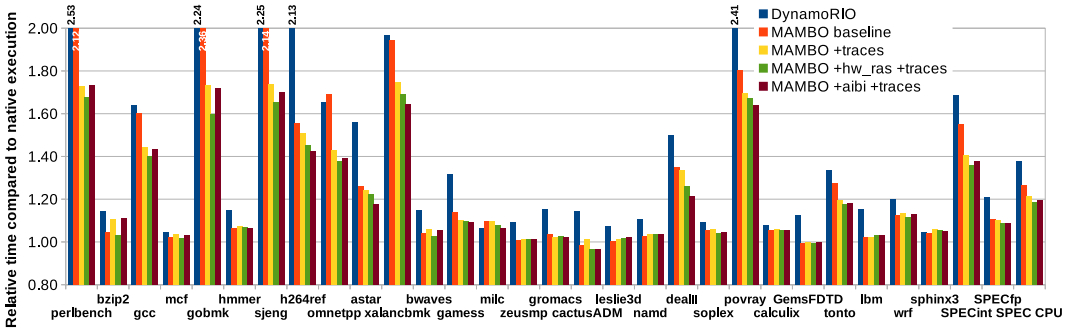


Figure 7: Relative execution time for SPEC CPU2006 on ODROID-XU3 (Cortex A7 in-order).

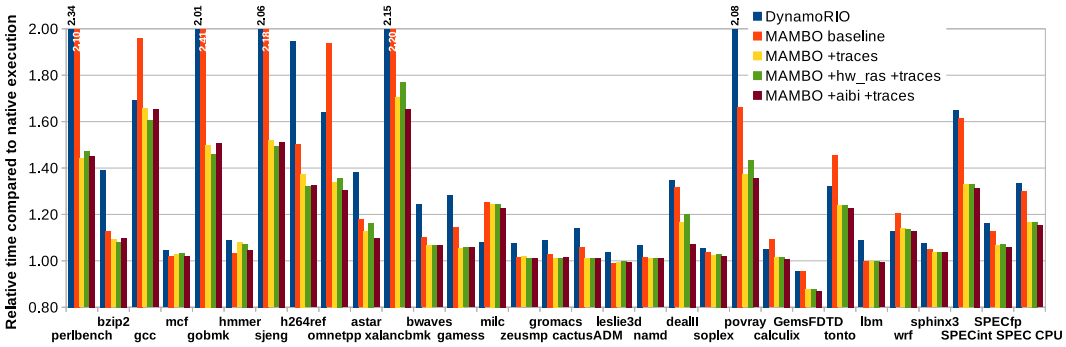


Figure 8: Relative execution time for SPEC CPU2006 on ODROID-X2 (Cortex A9 out-of-order).

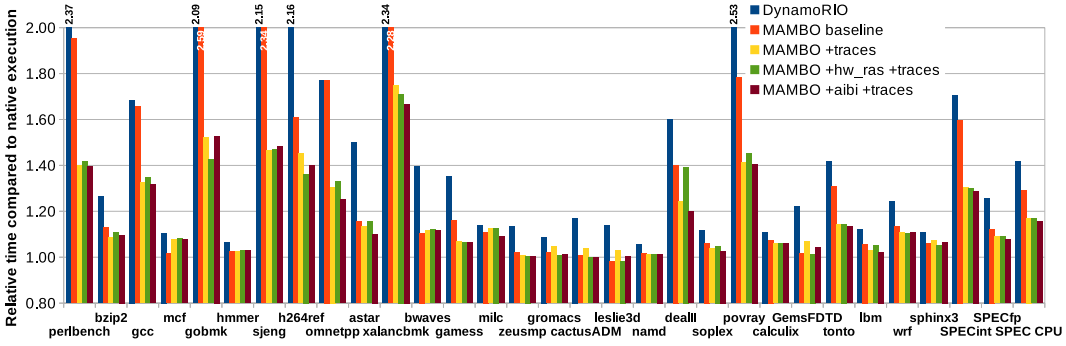


Figure 9: Relative execution time for SPEC CPU2006 on Tronsmart R28 (Cortex A17 out-of-order).

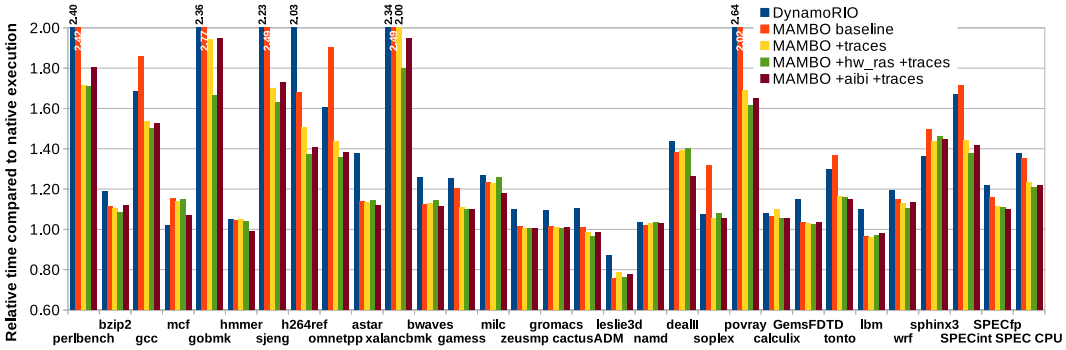


Figure 10: Relative execution time for SPEC CPU2006 on Jetson TK1 (Cortex A15 out-of-order).

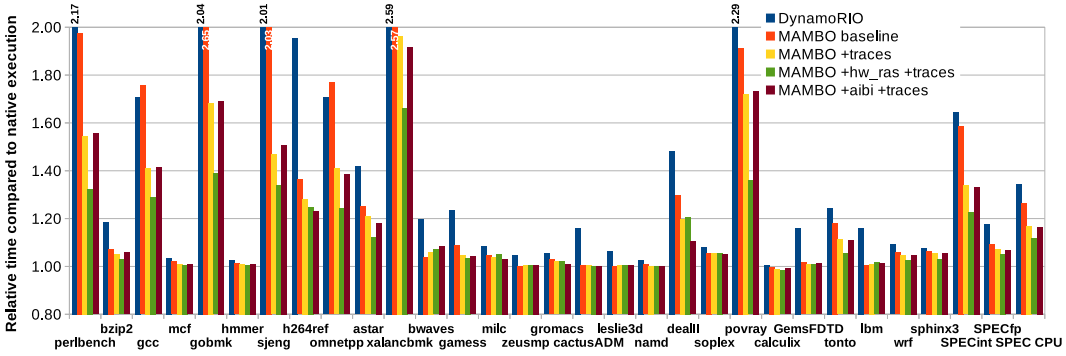


Figure 11: Relative execution time for SPEC CPU2006 on APM X-C1 (X-Gene1 out-of-order).

		ODROID-XU3		ODROID-X2		Tronsmart R28		Jetson TK1		APM X-C1	
Config	Benchmark	IPC_r	Time	IPC_r	Time	IPC_r	Time	IPC_r	Time	IPC_r	Time
Baseline	perlbench	0.73	2.12	0.76	2.10	0.79	1.95	0.64	2.42	0.73	1.98
	gobmk	0.59	2.36	0.58	2.41	0.53	2.59	0.50	2.77	0.59	2.65
	sjeng	0.70	2.14	0.66	2.18	0.62	2.34	0.59	2.49	0.70	2.03
	xalancbmk	0.93	1.94	0.91	2.20	0.79	2.28	0.73	2.49	0.93	2.57
+traces	perlbench	0.88	1.73	1.01	1.44	1.08	1.40	0.89	1.71	0.88	1.54
	gobmk	0.78	1.73	0.86	1.50	0.89	1.52	0.70	1.94	0.78	1.68
	sjeng	0.84	1.74	0.93	1.52	0.99	1.47	0.86	1.70	0.84	1.47
	xalancbmk	0.93	1.75	1.07	1.71	1.00	1.75	0.88	2.00	1.01	1.96
+hw_ras (+)	perlbench	0.93+	1.68+	1.00*	1.45*	1.09	1.40*	0.92+	1.71+	0.93+	1.32+
or	gobmk	0.87+	1.60+	0.90+	1.46+	0.97+	1.43+	0.84+	1.67+	0.87+	1.39+
+aibi (*)	sjeng	0.90+	1.65+	0.94+	1.49+	1.00+	1.47+	0.91+	1.63+	0.90+	1.34+
+traces	xalancbmk	0.97*	1.65*	1.00*	1.66*	0.96*	1.67*	1.02+	1.80+	1.07+	1.66+

Table 5: Relative IPC and slowdown for the benchmarks with the highest relative IPC

baseline configuration by 27% on ODROID-XU3, 41% on Jetson TK1, 45% on Tronsmart R28, 50% on ODROID-X2 and 54% on APM X-C1.

5.3 Performance counter analysis

Using the *perf* Linux tool, which monitors architectural and microarchitectural events using the hardware performance counters, we can gain an insight into 1) the performance differences between the various microarchitectures and 2) the effects of each optimisation introduced in this paper. The following events were counted on each system[1] for native execution and each MAMBO configuration, using the benchmarks with significant overhead: cycles, retired instructions, L1 data, L1 instruction and L2 unified cache accesses and misses, L1 data and instruction TLB misses, architecturally executed branches, mispredicted or not predicted branches speculatively executed.

5.3.1 Predicting speed-up: instructions per cycle. Increasing the dynamic instruction count is an expected effect of DBM. However, by designing the generated code to avoid pipeline stalls (i.e. by avoiding cache misses, branch mispredictions, etc.) the performance overhead should be disproportionately low (i.e. DBM execution should have an equal or higher Instructions Per Cycle — IPC — rate than native execution). Therefore, the relative IPC ($IPC_r = IPC_{DBM}/IPC_{native}$) can be used to identify the workloads which are the least efficient at a microarchitecture level when executed under a given DBM system, and which can be expected to benefit the most from the optimisations presented in this paper. Table 5 shows the IPC_r and relative execution time under the baseline, *+traces* and optimal MAMBO configurations, for the benchmarks with the lowest IPC_r. The results in this table show that the microarchitectural optimisations are effective in reducing the overhead of the benchmarks with a low IPC_r, as intended. For example, looking at the last two columns, we can see that *gobmk* on APM X-C1 has a slowdown of 2.65x under baseline MAMBO, with a IPC_r of 0.59. When traces and then hardware-assisted return address prediction are enabled, the slowdown is reduced to 1.68x with an IPC_r of 0.78 and 1.39x with an IPC_r of 0.87 respectively.

5.3.2 The indirect branch optimisations. The two indirect branch optimisations (*hardware-assisted return address prediction* and *AIBI*) have a varying degree of effectiveness between each benchmark and each system. As a case study, we analysed *xalancbmk*. Compared to the *+traces* configuration, the performance on this benchmark was improved by *AIBI* on all systems and by *hardware-assisted return address prediction* on all systems except on ODROID-X2. *Hardware-assisted return address prediction* has better performance than *AIBI* on Jetson TK1 and APM X-C1. On this benchmark, *hw_ras* increases the number of retired instructions by around 3%, while *AIBI* reduces it by around 9%, therefore the performance difference is not determined by microarchitectural effects alone. The significant performance counter changes, relative to the *+traces* configuration, are summarised in Table 6.

We can observe that *hw_ras* is more effective than *AIBI* at reducing the number of branch mispredictions, L1 instructions cache misses and L2 cache loads. *AIBI* tends to introduce additional L1

[1] with several exceptions due to unsupported events: L1 instructions loads, L2 loads and L2 misses on ODROID-X2 and branch mispredictions on APM X-C1

Opt.	System	Positive effects	Negative effects	Speedup
hw_ras	ODROID-XU3	-14% icache misses -7% L2 loads	+8% icache loads +8% dTLB misses	3.3%
	ODROID-X2	-24% icache misses -13% iTLB misses	+25% branch mispredictions	-3.5%
	Tronsmart R28	-29% branch mispredictions -26% icache misses -20% iTLB misses -14% L2 loads	-	2.2%
	Jetson TK1	-48% branch mispredictions -41% iTLB misses -24% icache misses -16% L2 loads -9% icache loads	-	11.4%
	APM X-C1	-22% icache misses -14% L2 loads -6% icache loads	-	18.0%
AIBI	ODROID-XU3	-15% icache loads -10% dcache loads -10% icache misses -9% dTLB misses -7% iTLB misses	+15% branch mispredictions	6.1%
	ODROID-X2	-8% branch mispredictions -7% dcache loads	+14% iTLB misses +9% dcache misses +7% dTLB misses	3.7%
	Tronsmart R28	-18% icache loads -13% icache misses -13% branch mispredictions	+14% iTLB misses +12% dTLB misses +10% dcache misses	6.9%
	Jetson TK1	-13% icache loads -9% branch mispredictions	+20% dTLB misses +12% iTLB misses +12% dcache loads	8.4%
	APM X-C1	-16% L2 loads -11% icache loads -10% icache misses -7% dcache loads	+11% dcache misses	2.5%

Table 6: Performance counter changes for *xalancbmk* compared to the *+traces* MAMBO configuration

data cache misses, caused by accessing the predicted SPC and TPC. However, on the microarchitectures with no or less advanced out-of-order executing capabilities (Cortex-A7, Cortex-A9 and Cortex-A17), the reduced dynamic instruction count of *AIBI* allows it to achieve better performance.

The performance degradation caused by *hw_ras* on ODROID-X2 appears to be caused by an increased number of branch mispredictions. Unfortunately, the return address predictors are not documented in enough detail to determine why this is happening or the relevant differences between the return address predictors of these microarchitectures. A possible explanation is that the return address predictor of Cortex-A9 compares the predicted address and the actual address early during execution, causing it to sometimes use the SPC before it has been replaced by the TPC, however we have not been able to verify this hypothesis.

5.3.3 The effect of traces. The *traces* optimisation proved to be very effective and almost always improved or did not significantly affect performance. However, in three cases (*bzip2* on ODROID-XU3, *hmmer* on ODROID-X2 and *mcf* on Tronsmart R28) execution was measurably slowed down (by around 6% for all three). In the case of *bzip2* on ODROID-XU3, two of the events we monitor show a change which is potentially relevant: an increase in the rate of branch mispredictions (from 9.2 to 9.5 per 1000 instructions) and L2 cache misses (from 5.0 to 5.5 per 1000 instructions). This is likely caused by the increased code size. However, because both of these changes are relatively small, there is a strong possibility that there are other microarchitectural effects contributing to the slower execution speed, which are not captured by the performance counter events we have monitored. For *hmmer* on ODROID-X2, only the number of branch mispredictions is significantly increased (by around 460 million). However, on its own, the penalty of the additional branch mispredictions cannot account for the additional 167 billion execution cycles. For *mcf* on Tronsmart R28, there was no significant change in the number of any of the monitored performance counter events. Therefore, it appears that the main cause of this rare performance regression caused by the *traces* optimisation is not captured by the performance counter events we have monitored.

6 RELATED WORK

DBT and DBM are a popular research area, with a number of available tools [8, 10, 20, 22, 24, 25]. The strength of MAMBO is in prioritising the performance of a DBM implementation for ARM.

IBI is a common software target prediction scheme for indirect branches [7, 10, 19, 22, 25, 29]. However, IBI is limited by the high misprediction rate and the high penalty for mispredictions. Kim and Smith go as far as calling this technique a *performance limiter* [21]. This paper introduces AIBI as a replacement for IBI, which improves the prediction rate by allowing the predicted address to be updated after every miss, with low overhead. AIBI is similar in concept to the MRU algorithm introduced by Dhanasekaran and Hazelwood [14], however AIBI is a replacement for IBI, while MRU is used in *addition* to IBI. Furthermore, IBI handles prediction misses differently from MRU, which likely contributes to its better performance. AIBI and MRU are compared in detail in Section 4.2.

Dynamically building traces of the hot code path was first enabled by the NET [15] profiling algorithm. NETPlus [12] was later proposed as an improvement, which allows building longer traces by working across backwards branches. However, both of these algorithms create traces across indirect branches, using static software target prediction in the form of IBI. Because of the previously discussed limitations of IBI, this paper proposes an improvement of NET, which avoids software target prediction in traces altogether.

Efficient translation of return instructions is critical for achieving low overhead in DBM systems [21]. Some of the proposed solutions for optimising returns include: modifying the ISA to allow explicit manipulation of the hardware RAS [21], however this change has not been implemented on general purpose architectures such as x86 or ARM; maintaining a software RAS [18], however this is only beneficial on modern microarchitectures if certain transparency guarantees are relaxed [17], or in the case of DBT when the target architecture provides additional registers which can be directly used as a RAS pointer [11]. In this context, this paper introduces hardware-assisted return address prediction, which was developed to allow use of the hardware mechanisms for return address prediction, while forgoing the use of a software RAS.

7 SUMMARY AND CONCLUSIONS

MAMBO is an open source implementation of a DBM framework for the ARM architecture. In this paper, we have introduced three optimisations to address some of its performance limitations: *traces* increase the code cache locality by grouping together basic blocks which are likely to execute sequentially; *hardware-assisted return address prediction* is a technique which enables use of the hardware return address prediction without maintaining a software return address stack; and *adaptive indirect branch inlining* is a software indirect branch prediction scheme which allows quick and frequent updates of the predicted address. By using the right combination of these optimisations on each system, the geometric mean overhead of MAMBO is reduced by at least 27% (on ODROID-XU3) and by as much as 54% (on APM X-C1) compared to the *baseline* MAMBO configuration.

The performance of the various optimisations is analysed on five different ARM platforms, which allows us to show that 1) whether an optimisation for a DBM system is effective or not depends on multiple factors, including the microarchitecture of the processor on which it is running and the type of workload, and that 2) the optimal combination of optimisations can be different between multiple systems.

We have shown that some optimisations can improve performance on one system and decrease performance on other systems while running the same workload. For example, *hardware return address prediction* reduced the overhead of MAMBO on the *perlbench* benchmark on ODROID-XU3 and APM X-C1 and increased it on Tronsmart R28. These results are due to the wide range of ARM microarchitectures commercially available. Therefore, we recommend that future evaluations of DBM overhead use a similar wide range of hardware platforms. Furthermore, this shows that runtime or deployment time selection of optimisations can be desirable to achieve consistent performance.

With regards to the three optimisations presented in this paper, we recommend that the *traces* optimisation is always used for SPEC CPU type workloads. *Hardware return address prediction* appears to be most effective on high performance cores (APM X-Gene) or cores with limited prediction support for generic indirect branches (Cortex-A7) because it trades off a higher dynamic instruction count for improved hardware branch prediction. The effectiveness of *AIBI* is dependent on the workload. For microarchitectures with shorter pipelines and low hardware branch misprediction penalties, AIBI appears to be more effective for translating returns than *hardware return address prediction*, as long as hardware indirect branch prediction is supported, because it reduces the dynamic instruction count.

REFERENCES

[1] Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices* 41, 11 (2006), 2–13.

[2] ARM. 2013. *ARM® Cortex®-A15 MPCore™ Processor Technical Reference Manual, Revision r4p0.*

[3] ARM. 2013. big.LITTLE Technology: The Future of Mobile. (2013). https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf (Visited on 13/07/2016).

[4] ARM. 2013. *Cortex™-A7 MPCore™ Technical Reference Manual, Revision r0p5.*

[5] ARM. 2014. *ARM® Cortex®-A17 MPCore Processor Technical Reference Manual, Revision r1p1.*

[6] ARM. 2016. *ARM® Cortex®-A57 MPCore Processor Technical Reference Manual, Revision r1p3.*

[7] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, Vol. 35. ACM, 1–12.

[8] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator.. In *USENIX Annual Technical Conference, FREENIX Track*. 41–46.

[9] Darrell Boggs, Gary Brown, Nathan Tuck, and K Venkatraman. 2015. Denver: NVIDIA's First 64-bit ARM Processor. (2015).

[10] Derek Lane Bruening. 2004. *Efficient, transparent, and comprehensive runtime code manipulation.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[11] Amanieu d'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. 2016. Optimizing indirect branches in dynamic binary translators. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 7.

[12] Derek Davis and Kim Hazelwood. 2011. Improving region selection through loop completion. In *ASPLOS Workshop on Runtime Environments/Systems, Layering, and Virtualized Environments (RESoLVE).*

[13] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The Transmeta Code Morphing™ 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.

[14] Balaji Dhanasekaran and Kim Hazelwood. 2011. Improving indirect branch translation in dynamic binary translators. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering, and Virtualized Environments.* 11–18.

[15] Evelyn Duesterwald and Vasanth Bala. 2000. Software Profiling for Hot Path Prediction: Less is More. *SIGPLAN Not.* 35, 11 (Nov. 2000), 202–211. https://doi.org/10.1145/356989.357008

[16] Cosmin Gorgovan. 2016. MAMBO: A Low-Overhead Dynamic Binary Modification Tool for ARM. (2016). https://github.com/beehive-lab/mambo.

[17] 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. https://doi.org/10.1145/2896451

[18] 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, 261–270.

[19] Jason D Hiser, Daniel Williams, Wei Hu, Jack W Davidson, Jason Mars, and Bruce R Childers. 2007. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Proceedings of the International Symposium on Code Generation and Optimization.* IEEE Computer Society, 61–73.

[20] Jeffrey K Hollingsworth, Barton Paul Miller, and Jon Cargille. 1994. Dynamic program instrumentation for scalable performance tools. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the.* IEEE, 841–850.

[21] 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.

[22] 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, 190–200.

[23] 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, 143–152.

[24] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan Notices*, Vol. 42. ACM, 89–100.

[25] Mathias Payer and Thomas R Gross. 2010. Generating low-overhead dynamic binary translators. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference.* ACM, 22.

[26] 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, 25.

[27] 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.

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

[29] Emmett Witchel and Mendel Rosenblum. 1996. Embra: Fast and Flexible Machine Simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '96).* ACM, New York, NY, USA, 68–79. https://doi.org/10.1145/233013.233025

[30] 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, 27–38.