Open access • Proceedings Article • DOI:10.1145/3381052.3381322

Optimising dynamic binary modification across 64-bit Arm microarchitectures

Guillermo Callaghan, Cosmin Gorgovan, Mikel Luján

Institutions: University of Manchester

Published on: 17 Mar 2020 - Virtual Execution Environments

Topics: Overhead (computing) and Indirect branch

Optimising Dynamic Binary Modification across 64-bit Arm Microarchitectures

DOI:
10.1145/3381052.3381322

Document Version
Accepted author manuscript

Published in:
Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20)

Optimising Dynamic Binary Modification across 64-bit Arm Microarchitectures

Guillermo Callaghan
Department of Computer Science
University of Manchester
United Kingdom
guillermo.callaghan@manchester.ac.uk

Cosmin Gorgovan
Department of Computer Science
University of Manchester
United Kingdom
cosmin.gorgovan@manchester.ac.uk

Mikel Luján
Department of Computer Science
University of Manchester
United Kingdom
mikel.lujan@manchester.ac.uk

Abstract

A common optimisation used in most Dynamic Binary Modification (DBM) systems is *trace generation* as these traces improve locality and code layout. We describe an optimised code layout for traces as well as present how to adapt the runtime algorithm to generate it. In this way, we manage to reduce the overhead on all the Arm systems evaluated; 5 different microarchitectures.

A major source of overhead for DBMs comes from handling indirect branches. Indirect Branch Inlining (IBI) is a mechanism that attempts to avoid this overhead by using predictions about the target of the indirect branch. We analyse the behaviour of the indirect branch inlining and propose a new predictor, *Trace Restricted IBI (TRIBI)*, and how to optimise IBI given the new trace generation algorithm.

Our evaluation shows a geometric mean overhead for SPEC CPU2006 of 9% for a Cortex-A53 (in-order core), and for out-of-order cores 11% on an X-Gene-2, 10% on a Cortex-A57, 7% on a Cortex-A72 and 8% on a Cortex-A73, when compared to native execution. This is a reduction of the overhead between 30% to 50% compared to the publicly available DBM systems MAMBO, and, even higher, against DynamoRIO. Using PARSEC 3.0, we evaluate the scalability across threads on the X-Gene-2 system (server machine with the highest number of cores) and show a geomean overhead between 6-8%.

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

Keywords Dynamic Binary Modification, Dynamic Binary Instrumentation, Aarch64, 64-bit Arm

VEE '20, March 17, 2020, Lausanne, Switzerland

ACM Reference Format:
Guillermo Callaghan, Cosmin Gorgovan, and Mikel Luján. 2020. Optimising Dynamic Binary Modification across 64-bit Arm Microarchitectures. In *16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20), March 17, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3381052.3381322

1 Introduction

Dynamic Binary Modification (DBM) is a software technique that allows the binary of an application to be modified at runtime transparently. DBM systems are used in many areas such as virtualisation [26], binary instrumentation [17], program analysis [22] and translation [23] [7] [10].

DBM systems have an associated execution time overhead when compared with running the same application directly on the processor. Most DBMs run in the same process as the application they modify. A DBM scans and translates the application binary code before its execution. In addition, Arm systems present an extra challenge due to the variety of available microarchitectures; from low-power in-order processors, to more aggressive out-of-order processors. Optimisations for DBM systems usually are reliant on specific aspects of the microarchitecture on which they run. An optimisation relying on extending the execution path by increasing the number of instructions, such as trace unrolling will, in turn, have a negative impact on a microarchitecture with a small instruction cache. For example, in the evaluation, Section 4, the Arm systems have the L1 instruction cache from 32KiB to 64KiB. Providing optimisations that perform well, or at least do not produce a negative impact, on a group of diverse implementations is a non-trivial task.

A common optimisation used in most DBM systems (such as Pin [21] and DynamoRIO [8]) is the creation of traces. In general, a *trace* is a grouping of often executed basic blocks (the application critical path) and tries to improve the code locality and layout. We describe the code layout for traces, as well as adapt the generation algorithm (based on Next Executing Tail (NET) online profiling [14]) which manages to reduce the overhead on all the Arm systems considered in this paper. The code layout, referred to as *Contiguous Traces*, avoids having interleaved exit branches in the critical path. This reduces the critical path size and decreases the number of branches observed by the fetch stage of a processor.

A major source of overhead for DBM systems comes from handling indirect branches. Most DBM systems use an optimised hash table lookup to map application addresses to translated addresses; where the code has been relocated. Indirect Branch Inlining (IBI) is a mechanism that attempts to avoid this overhead by using predictions about the target of the indirect branch. As with previous studies targeting x86 [12] [18], we observe that many indirect branches have a low number of targets. Constrained by the contiguous trace layout generation, we implement an optimised indirect branch inlining mechanism that only uses as prediction, those targets which are themselves part of a trace. In other words, we restrict the prediction to indirect branches which are *hot* and to targets which are also part of *hot* code. The predictor is called *Trace Restricted IBI* (TRIBI). Previous approaches have applied IBI either to all indirect branches, or have not discriminated and used as predictions any target [16] [8].

The main contributions are summarised as:

- an optimised trace code layout generation algorithm which reduces the critical path size and increases the performance for MAMBO on a range of Arm microarchitectures;
- an efficient prediction mechanism, TRIBI, for indirect branch inlining;
- a reduction of the geometric mean overhead on SPEC CPU 2006 between 30% to 50% compared to baseline MAMBO.

The next sections of the paper are organised as follows. Section 2 provides an introduction to the baseline MAMBO DBM system, on which we have implemented the two optimisations. Section 3 describes the trace layout, and new algorithm to generate it, as well as TRIBI. Section 4 presents the performance evaluation which includes results for MAMBO and DynamoRIO. The Arm systems used in the evaluation provide a comprehensive coverage of the evolution of different Arm 64-bit microarchitectures. Section 5 compares the two optimisations with the main related work and Section 6 summarises the conclusions.

2 System Overview

MAMBO takes an application binary as input, then scans it and executes it. This is done in the same process as the application it modifies. MAMBO executes the application in the units of basic blocks — single-entry single-exit pieces of code. MAMBO maintains control throughout the execution by modifying basic block exits to branch to itself.

The modified application code is likely to be executed numerous times. To avoid multiple re-translations of the same application code, the first time a code fragment is translated, it is stored in a memory region owned by the DBM, called a *software code cache*. The software code cache is a structure in memory which holds the modified application code and its metadata, including a hash table that holds the *Source Program Counters* (SPC) and the corresponding *Translated Program Counters* (TPC). Other metadata used by the DBM system includes the location of the next free location to store a basic block. The size of the software code cache in MAMBO is configurable, however, it must be smaller than the maximum range of a direct branch (see Section 2.1.1) to avoid extra trampolines. A trampoline is a location in memory that holds a direct branch used to extend the range of direct branches.

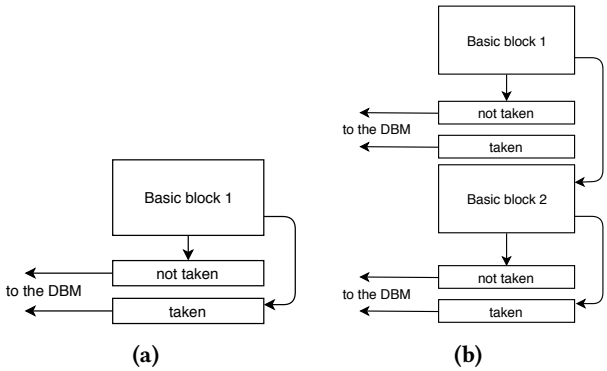

Figure 1. Basic blocks — (a) a newly created basic block; (b) the *taken* path directly linked to Basic Block 2.

MAMBO uses thread-private code caches, which minimises synchronisation when modified applications use multiple threads. The baseline implementation and optimisation of MAMBO were presented by Gorgovan *et al.* [15] [16].

2.1 Branches

There are two types of branches in the Arm 64-bit (A64) instruction set [4]: direct branches and unconditional indirect branches. The former jump to an offset relative to the Program Counter (PC) that is included in the instruction encoding and the latter jump to an address stored in a register. Handling direct and indirect branches differ and are presented separately.

2.1.1 Direct Branches

The offset of a direct branch is included in the instruction encoding, which also means that it is known at compile time and it does not change. There are two types of direct branches: *conditional direct branches* and *unconditional direct branches*. Conditional direct branches have two paths; the taken path and the fall through path. The A64 ISA has three conditional direct branches: `tb(n)z` (test bit and branch on (non) zero), `cb(n)z` (conditional branch on (non) zero) and `b.cond` (branch conditional). On the other hand, unconditional direct branches have only one path which is always taken. The A64 has two unconditional direct branches: `b` (branch) and `bl` (branch link). The path taken by a direct branch from a basic block leads to the creation of a new basic block starting from the address pointed by the branch.

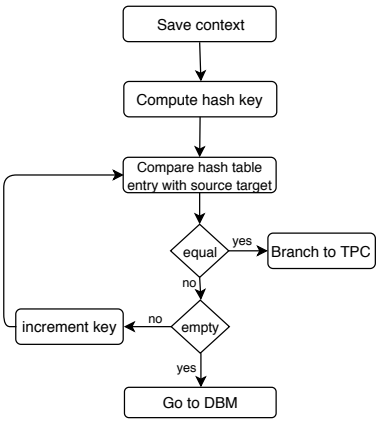

Figure 2. Inline hash lookup routine.

The first time a basic block is executed, the control is transferred back to the DBM. The exit stubs of basic blocks (one for each path of the branch) set the parameters needed by the DBM, such as the application address of the executed path and the basic block number to retrieve its metadata. This results in a context switch, which is a costly operation, as the complete state of the application needs to be stored. Once the new basic block is created the state of the application needs to be restored. To avoid context switches in future executions of the basic blocks, an optimisation technique called *linking* (sometimes referred as chaining) is applied. This optimisation consist in modifying the executed branch path to link directly to the newly created basic block. The fall through path is linked only if a basic block for that path is already present in the code cache. In case a basic block for the specific address is absent in the code cache, the path is left unmodified. Figure 1a shows a newly created basic block with its translated branches transferring control to the DBM. Figure 1b shows basic block 1 with one of its branches directly linked to basic block 2.

Direct branches have a limited number of bits dedicated to store the offset and it varies depending on the type of branch. `cb(n)z` and and `b.cond` have a 19-bit offset allowing a range of ± 1MB. `tb(n)z` have an offset of 14 bit with a range of ± 32KB. And `b` and `bl` have a 26-bit offset with a range of ± 128MB. Basic blocks are executed and stored in the code cache in the order in which they are executed and sometimes their location in memory is beyond the reach of a conditional direct branch. To handle this situation in a coherent manner all conditional direct branches have a trampoline direct branch to extend the range of the branch and reach any address in the code cache.

2.1.2 Indirect Branches

Indirect branches in the A64 ISA only jump to an address stored in a register. The *linking* optimisation cannot be applied to indirect branches. This is because the address in the register is only known at runtime. Moreover the same branch will potentially branch to many different targets. The translation of the *SPC* to the *TPC* is carried out at runtime for every execution of the translated indirect branch. Performing the translation means that its cost is never amortised and is always paid on every execution. Handling indirect branches is responsible for a significant percentage of the overhead of dynamic modification/translation systems [18] [20] [12] [9] [10].

Whenever a translated indirect branch is executed, control is transferred to the DBM in order to find the translation of the target address. A mapping of the SPC to the TPC is stored in the hash table. This causes a context switch and, if the target is not found, the corresponding basic block will be created. This means that whenever an indirect branch is executed, there is a context switch. To minimise the number of context switches, most DBM systems use a highly optimised hash table lookup routine. Figure 2 shows the inline hash lookup routine which is appended in place of the original indirect branch in MAMBO.

The first step saves the context. Then the hash key is computed. This is done using an AND operation between the SPC and a configurable hash mask:

```
key = (target >> 2) & HASH_MASK
```

The key is used to access the hash table entry corresponding to the target address. If the address in the hash table is equal to the target address it means there is a translation in the code cache. The translation is loaded and the execution branches to the translated address. If the address comparison is not equal then it is checked if the value was zero. A zero represents that a translation is not present in the code cache, so execution is transferred to the DBM. If the value in the hash table was not zero, the key is incremented and the next location is probed. This means there is a collision. The loop exits if there is no translation or a translation is found.

2.2 Traces

The layout of the basic blocks in the code cache introduces a greater fragmentation compared to the source application. This is because basic blocks are scanned and stored in the code cache as they are executed, which potentially differs from the original layout of the source application. Moreover, as shown in Figure 1b, both paths of conditional branches are stored as different basic blocks. This means that there is an extra branch for each translated conditional branch. An optimisation to overcome the fragmentation incurred by basic blocks in the code cache is to group basic blocks together that are part of the critical path. This group of basic blocks is called a trace, where each basic block is a trace fragment. Traces are also known as *superblocks* in the literature [8] [26]. Traces are single-entry multiple-exits pieces of code.

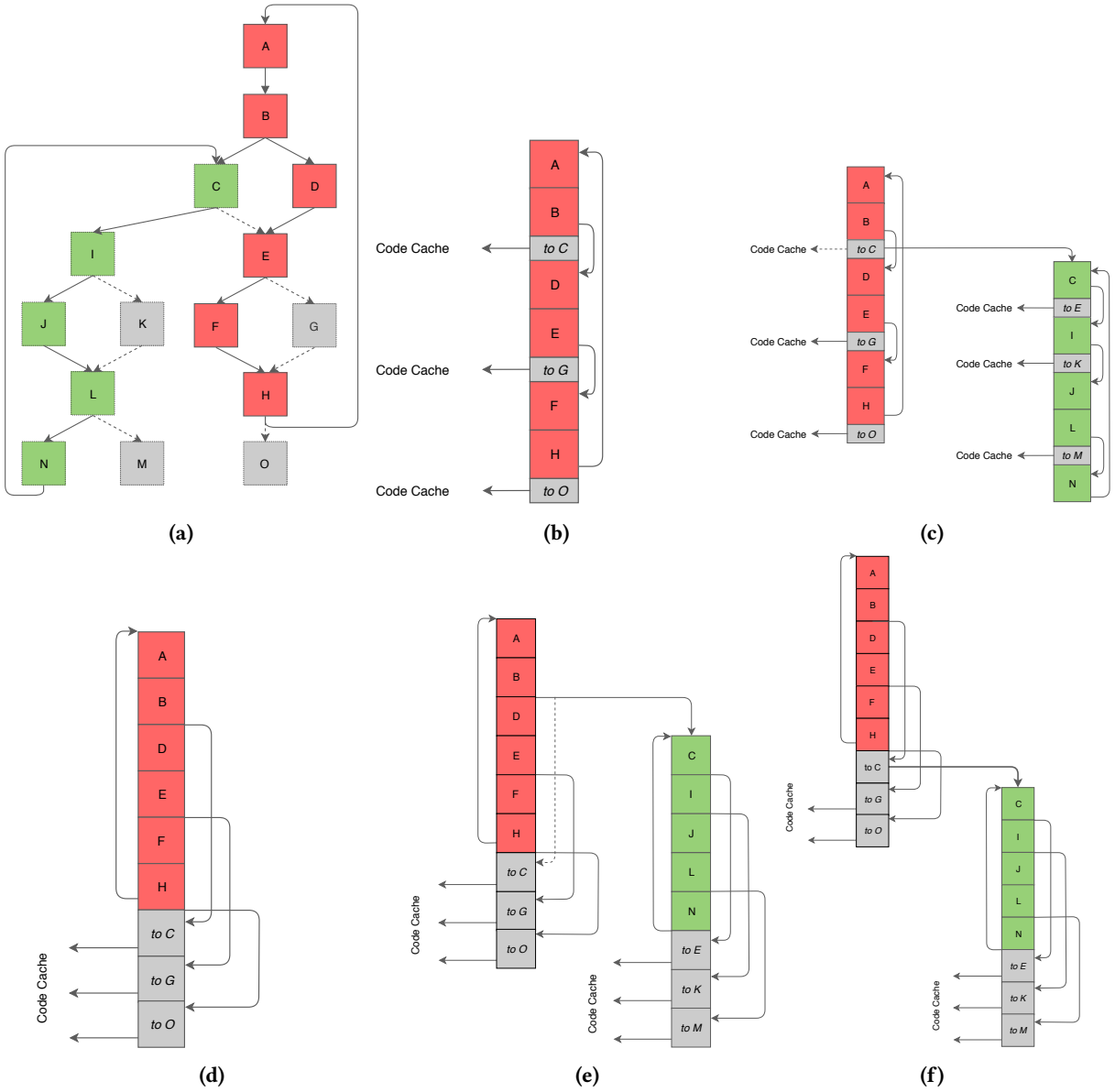

Figure 3. (a) Example control of flow from an application. Basic block are represented by boxes. **(b)** Trace layout of the first *hot* loop with the baseline scheme (interleaved). **(c)** Trace linking: once the second *hot* is identify and a trace is built, an exit from the first trace is linked to the trace entry of the second loop. The replaced path of the branch after linking is shown with a dotted line. **(d)** Contiguous Trace layout of the first *hot* loop. **(e)** Contiguous Trace linking scheme. The replaced path of the branch after linking is shown with a dotted line. **(f)** Linking a trace when the target is outside the range of a conditional direct branch.

2.2.1 Trace Selection

There are two challenges for trace creation. First, traces need to be created only for the *hot* code of the application. Second, the online profiling should be done with low overhead.

Creating traces for some *cold* code is acceptable, but this needs to be kept to a minimum. This is because the overhead of creating a trace would not be amortised due to the low numbers of executions. Creating traces adds overhead as execution is stopped and the code has to be re-scanned to create the trace. It also increases the software code cache size. At the same time the benefit of traces is maximised the sooner they are created. However, these two aims are incompatible. The sooner code is identified as *hot* code the more *cold* code could potentially be promoted to a trace. MAMBO uses a modified Next Executing Tail (NET) online profiling [14]. NET produces a limited number of blocks to

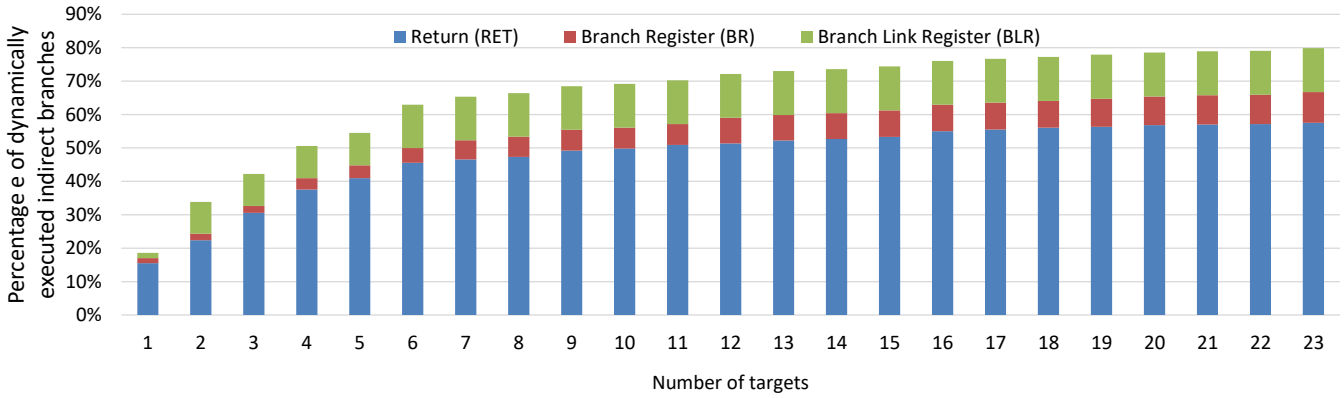

Figure 4. Percentage of dynamic indirect branches captured by inlining target predictions across SPEC CPU 2006.

be instrumented. Only basic blocks which are the target of backwards branches and the target of trace exits are instrumented. The instrumented basic blocks are called *path heads*. A *path head* is a basic block identified as a potential *trace head*. Once the execution times of a *path head* reaches a set threshold, the following *path tail* is predicted as the *hot* path. A *path tail* is the next pieces of code executed after the *path head*. The *path tail* is recorded until a backward branch is encountered. The prediction is that the *path head* suggest the application is executing *hot* code and the *path tail* is also part of that *hot* code. In the implementation of NET in Dynamo [6], some indirect branches are transformed into direct branches where the target was predicted as being the target first taken by the branch. Also NET records the path of traces over indirect branches.

In MAMBO, the existing implemented profiling scheme is a modified version of NET. A full description of the selection algorithm is described by Gorgovan *et al.* [16]. This comprises two main differences. The first is that most of the basic blocks are profiled, except those terminating with an indirect branch. The second change is that indirect branches are a termination condition that stops path recording. In fact, a basic block terminating in an indirect branch would, otherwise, create a single fragment trace, unnecessarily incurring the overhead of recreating this basic block as a trace with a single basic block.

In other words, instead of adding a prediction to the target seen by an indirect branch at translation time, as NET does, the trace is terminated. The reason for this is that, especially in the case of return instructions for functions that are called from multiple locations, this prediction could be a source of overhead when a high number of misspredictions occur.

3 Optimisations

This section describes the two optimisations introduced by this paper and implemented in MAMBO. First, we explain the change in the code layout of traces to avoid having interleaved exit branches in the critical path. Then we explain the implementation of Indirect Branch Inlining, a software prediction mechanism to reduce the number of hash table lookups at runtime.

3.1 Contiguous Trace Construction

Figure 3a shows a control flow graph from an example application where boxes represent basic blocks, and arrows between basic blocks represent a branch and its targets. A single arrow below a basic block is an unconditional direct branch, double arrows represent the two paths of a conditional branch. Solid lines are branch targets which are part of the *hot*-code. Dashed lines are targets which are part of the *cold*-code. Figure 3b shows a trace constructed from the control flow graph presented in Figure 3a, and starting at fragment A, using the baseline trace generation part of MAMBO. Following the execution of fragment A, B is appended to the trace. Since the branch from A to B is a direct branch, the actual branch instruction is omitted. In the case of conditional branches, the taken path target jumps one instruction and continues the execution in the trace. The fall through path is the exit to the translation of the *cold* path, shown as *to C* in Figure 3b. If the basic block does not exist it will be created. The process continues until the taken path of fragment H points to A, which is a termination condition. At this point, the trace ABDEFH has been formed.

3.1.1 Trace Code Layout

Figure 3d shows the layout of traces hereafter referred to as Contiguous Traces. Exits are recorded during trace creation and installed at trace termination as exit stubs. The new layout reduces the number of branches and the size of the critical path. A similar layout was described by Dynamo [6].

Figure 3a shows two pieces of *hot* code: ABDEFH (in red) and CIJLN (in green). Figure 3b shows the first loop constructed into a trace and Figure 3c shows the layout when the second loop becomes *hot*, both using the baseline MAMBO (interleaved). The direct branch (dashed arrow) between B

and D is patched to point to the entry of the new trace (solid line).

In the proposed layout, Figure 3e, the conditional branch taken path (dashed arrow) is modified to target the new target (solid line). Sometimes the new trace could be in an unreachable position in the code cache for a conditional direct branch. In this case the stub is modified and used as a trampoline to reach the trace. This is shown in Figure 3f.

In NET, correctly maintaining the exit points and including extra counters for profiling adds expensive overhead and complexity. Our adapted trace generation and layout do not suffer from this.

3.1.2 Trace Exits

In the baseline scheme, trace exits are composed of direct branches interleaved in the trace. In Arm, direct branches have a range of ±128MB and they can reach any address in the code cache. With Contiguous Traces, trace exits consist of a conditional direct branch where the fall through path leads to the next fragment in the trace and the taken path branches to an exit stub at the end of the trace, as shown in Figure 3d. This scheme elides branch instructions in the critical path and decreases the hot path size. Fewer branches in the critical path also helps the branch predictor [3] [2] [5].

Conditional direct branches have a limited range and they might need *trampolines*. This is specially important because trace exits at construction potentially target basic blocks, which are stored in a location outside the range of conditional direct branches.

To avoid having a conditional branch targeting an exit stub with only a direct branch, exit stubs are composed of the first two instructions of the target basic block and then a direct branch.

Also, exit stubs are 16-byte aligned as recommended by the Arm software optimisation guides [3] [2] [5], improving the utilisation of the fetch stage bandwidth for the target of branches. An exception to this rule is when the first or the second instruction are not position-independent.

3.2 Indirect Branch Inlining

The hash table lookup (shown in Figure 2), is used for translating SPCs to TPCs at runtime. The size of the inline hash table lookup routine is around 52 bytes (13 instructions). The hash lookup routine is composed of an indirect branch (`br`), data memory instructions, conditional branches, arithmetic instructions and in the case of a collision in the hash table a loop.

A mechanism developed in an attempt to avoid the execution of the hash table lookup is Indirect Branch Inlining (IBI) [11] [6]. The IBI is usually composed of a number of comparisons between the target of the translated indirect branch and a number of predicted targets. The predictions are selected from previous taken targets observed at runtime by the same branch. On a hit, the execution jumps directly to the translation without going through the hash table lookup. If a prediction fails, the next prediction is tested. If all the predicted targets fail, the execution falls back to the hash table lookup. If the hash table lookup find the translation, execution jumps to the TPC. If a translation is not found control goes back to the DBM system.

```
            save context
            ...
            adr     x0, spc_0
            sub     x1, x0, target_reg
            cbnz    x0, next_or_fallback
            restore context
            b       tpc_0
next_or_fallback:
```

Code 1. Branch Inlining target prediction. The predicted SPC (`spc_0`) is loaded into `x0` and compared with the current target. In a match, the context is restored and execution is transferred to the predicted translation of the target (`tpc_0`).

Previously, the implementations of the prediction mechanism have been done in two ways: using instruction immediate values to create the predicted targets [6]; or storing the predicted and the translated targets in memory [16] [12]. The first option avoids data memory access instructions, however it adds conditional branches and changing a prediction must flush the instruction cache, which is an expensive operation. On Arm systems the instruction cache and the data cache are not coherent and flushing the instruction cache has to be done explicitly [4]. On the other hand, for Intel systems the instruction and data cache are coherent [19]. Storing the predicted and translated value in memory makes it easy to change a prediction without flushing the instruction cache,

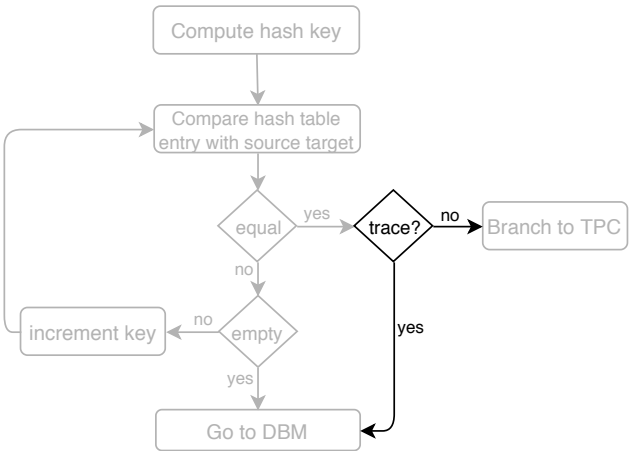

Figure 5. Fall back mechanism. The inline Hash Table Lookup is modified to go back to the DBM in the case where the translated target is a trace. The added steps are shown with black borders.

but at the expense of polluting the data cache and adding memory operations. There are other factors that need to be taken into account [18]: the number of targets to be inlined; whether the number of targets is decided statically or using runtime information; the range of target per type of indirect branch; and how to handle mispredictions (fall back).

Figure 4 shows the percentage of all dynamic executed indirect branches in the SPEC CPU 2006 benchmark suite [27] that would be predicted by IBI from 1 to 23 predictions. From 6 up to 23 predictions the additional number of branches predicted decreases to less than 5%. Capturing more than 80% of indirect branches would required a very high number of prediction. In relation to the total number of dynamic executed indirect branches, branch register and branch link register represent a small percentage. Gathering information at runtime to decide the number of targets to be inlined adds performance and memory overhead for each indirect branch.

We have implemented IBI only on indirect branches in traces which are also branching to a trace. The optimisation is called *Trace Restricted IBI* (TRIBI). Code in traces have been determined to be *hot* code. Using IBI predictions only in traces targeting traces raises the possibilities of making a prediction that will be reused. Moreover, IBI applied only on traces avoids the overhead of keeping track of inline targets to basic blocks as they may potentially be promoted to a trace and its code becomes stale.

In case the prediction fails, there are two outcomes: a) there are empty slots for new predictions; or b) the number of inline predictions is exhausted. In the former case, the DBM adds the new prediction and execution continues. In the latter, all the predictions are cancelled and only the hash table lookup is used.

When an indirect branch is first encountered in a trace, the stub transfers the execution to the fall back mechanism. We have implemented the fall back mechanism (Figure 5) as a modified version of the inline hash table lookup.

In the case when a translation is found for an SPC, the TCP is checked to see if the translation had been promoted to a trace. If the translated target is a basic block, execution is transferred to the basic block. If the translated target is a trace, execution is transferred to the DBM and a prediction is inserted. Code 1 shows an example of a prediction.

4 Evaluation

4.1 Setup

The performance is evaluated using five 64-bit Arm platforms with different microarchitectures. Table 1 shows the microarchitecture for all the systems. All the systems are running GNU/Linux with the manufacturer supported kernel. SPEC CPU2006 has been compiled with GCC 4.8 using the `-O2` optimisation level and the `-static` flag. Dynamic Voltage and Frequency Scaling (DVFS) has been disabled in all platforms. The Rock 960 (Cortex-A53/Cortex-A72) and the Hikey 960 (Cortex-A53/Cortex-A73) systems feature a *big.LITTLE* [1] architecture, however only the *big* cores are used for the evaluation. The *LITTLE* cores on both systems are a cluster of 4x Cortex-A53, which are already evaluated as part of the PineA64+ system.

The reported results are run three times each and the average of the three runs is presented. The workloads used are the *ref* input for SPEC CPU2006 and the *native* input for PARSEC 3.0. The benchmarks have run using multiple configurations of MAMBO. A configuration is a concrete set of optimisations enabled when compiling MAMBO. The baseline MAMBO used is git commit `d3856aa`[1].

The optimisation configuration evaluated are:

- *Contiguous Traces* generation MAMBO with the modified traces layout;
- *RAIBI*, a modified scheme of the state-of-the-art IBI for Arm, AIBI, proposed by Gorgovan *et al.* [16]; and
- *TRIBI* `X`, where `X` represents the number of targets inlined.

AIBI, Adaptive Indirect Branch Inlining, works by always predicting the last taken target of a branch. This target could be in any location of the code cache, basic blocks included. RAIBI, Restricted AIBI, is only applied to indirect branches which target traces, i.e., the prediction is only updated if this target is a trace. Note that *TRIBI* and the *RAIBI* optimisations are implemented on top of the *Contiguous Traces* layout; *TRIBI* is not compatible with the baseline trace layout used by MAMBO.

For completeness, DynamoRIO [8] is also part of the evaluation. DynamoRIO is a publicly available DBM system for Intel and Arm, covering both 32- and 64-bit Arm. Beyond MAMBO, DynamoRIO is the main DBM with support for 64-bit Arm and has received code contributions from Arm. The DynamoRIO version used is the git commit `d71feb5`[2], but it does not contain support for traces. We also tried the development branch which is implementing the trace support for Arm, but this cannot run SPEC CPU2006. In other words, the results using DynamoRIO provide an independent reference point to understand the low overhead achieved by the Baseline MAMBO, which is the start point for the optimisations introduced in this paper.

4.2 Performance evaluation

Table 2 shows a summarised geometric mean overhead performance when running SPEC CPU2006 on all the configurations: baseline MAMBO, MAMBO with the Contiguous Traces, MAMBO RAIBI, MAMBO TRIBI 1, MAMBO TRIBI 6 and DynamoRIO for each system. The table presents the geometric mean overhead execution for each system compared to native execution for the group of integer benchmarks (SPECint), floating point benchmarks (SPECfp) and

[1] https://github.com/beehive-lab/mambo/commit/d3856aa
[2] https://github.com/DynamoRIO/dynamorio/commit/d71feb5

Table 1. Systems used for evaluation

System	PineA64+	Applied Micro X-Gene 2	Jetson TX1	Rock 960[a]	HiKey 960[b]
SoC	Allwinner A64	APM883408-X2	NVIDIA T210	Rockchip RK3399	Kirin 960
Core	4x Cortex-A53	8x X-Gene 2[c]	4x Cortex-A57	2x Cortex-A72	4x Cortex-A73
Frequency	1.2GHz	2.4 GHz	1.73GHz	1.8GHz	2.36GHz
L1d cache size	32KiB	32 KiB	32KiB	32KiB	64KiB
L1i cache size	32 KiB	32 KiB	48KiB	48KiB	64KiB
L2 cache size	512 KiB	256 KiB	2MiB	1MiB	2MiB
L3 cache size	N/A	8 MiB	N/A	N/A	N/A
Out Of Order	N	Y	Y	Y	Y
Pipeline length	8	Not Disclosed	15+	14-16	11-12+
Linux distribution	Debian 8.11	Debian 9.8	Ubuntu 18.04	Ubuntu 18.04	Debian 9.8
Linux kernel[d]	3.10	4.9	4.4	4.4	4.14

[ab] The Rock 960 and the Hikey 960 systems feature a *big.LITTLE* architecture, with 4x Cortex-A53 as the little cores. However, the *LITTLE* cores were not used for this evaluation. The Cortex-A53 core was evaluated using the PINE A64+ system. [c] The X-Gene 2 cores are a custom implementation of Armv8 cores developed by Applied Micro. [d] The kernel versions used are the ones provided by the manufacturers.

the whole SPEC CPU2006 (CPU). The new MAMBO optimisation is labelled as TRIBI 1 and TRIBI 6. TRIBI 1 allows a direct comparison with RAIBI which also only remembers one indirect branch as a prediction.

Figure 6 shows detailed results for each benchmark on the Rock 960 system, for all the optimisation and DynamoRIO for comparison. Figure 7 shows detailed results for each benchmark on the PineA64+ system, for all the optimisation and DynamoRIO for comparison. Detailed graphs for the other systems are not shown due to space restrictions. Figure 8 shows the overhead of MAMBO (baseline) when running the multi-threaded benchmark suite PARSEC 3.0, while Figure 9 presents the overhead of Contiguous Traces and the TRIBI 6 optimisation on MAMBO. The MAMBO (baseline) implementation has a geometric mean overhead of 122% on 1 thread, 124% on 2 threads, 127% on 4 threads, and 133% on 8 threads. The geometric mean overhead for Contiguous Traces and the TRIBI 6 on 1 thread is 6%, for 2 threads is 6%, 4 threads is 6% and 8 threads is 8%.

Table 2 shows a clear difference in the overhead of integer and floating point benchmarks. Using the set of optimisations with the lower overheads, integer benchmarks range from 21% on the Applied Micro X-Gene 2 down to 14% on the Rock 960. On the other hand, floating point benchmarks are in the range 4-3%. This difference between the integer and floating benchmarks is directly correlated with the number of indirect branches executed by the benchmarks.

4.2.1 Contiguous Traces Generation

The Contiguous Traces generation manages to reduce the overhead in all systems. The biggest impact is on the Applied Micro X-Gene system, reducing the geometric mean overhead from 20% to 12%. The highest overheads per benchmark for each system are: Pine A64+ 42% (*400.perlbench*); Applied Micro 55% (*400.perlbench*); Jetson TX1 42% (*453.povray*); Rock 960 46% (*453.povray*); and Hikey 960 35% (*464.h264ref*). The benchmark with the lowest overhead is *481.wrf* on the Jetson TX1, with a 14% speedup.

Table 2. Performance overhead of all optimisations in MAMBO, and DynamoRIO running SPEC CPU2006 on all the systems compared to native execution.

		SPEC CPU2006		
System	DBM configuration	SPECint	SPECfp	CPU
Pine A64+	DynamoRIO	48.8%	14.4%	27.5%
	MAMBO	23.8%	6.4%	13.3%
	Contiguous Traces	17.5%	4.7%	9.8%
	MAMBO RAIBI	20.2%	5.3%	11.2%
	MAMBO TRIBI 1	16.2%	4.4%	9.3%
	MAMBO TRIBI 6	**16.2%**	**4.0%**	**8.9%**
Applied Micro X-Gene 2	DynamoRIO	60.4%	15.7%	32.4%
	MAMBO	36.5%	10.2%	20.4%
	Contiguous Trace	22.3%	5.4%	12.1%
	MAMBO RAIBI	22.7%	4.7%	11.8%
	MAMBO TRIBI 1	21.4%	5.0%	11.5%
	MAMBO TRIBI 6	**20.6%**	**4.2%**	**10.7%**
Jetson TX1	DynamoRIO	58.7%	15.0%	31.4%
	MAMBO	34.6%	9.6%	19.3%
	Contiguous Trace	20.2%	5.3%	11.2%
	MAMBO RAIBI	23.3%	4.8%	12.1%
	MAMBO TRIBI 1	20.0%	4.4%	10.6%
	MAMBO TRIBI 6	**18.7%**	**3.6%**	**9.6%**
Rock 960	DynamoRIO	62.7%	18.5%	35.1%
	MAMBO	25.3%	7.2%	14.4%
	Contiguous Trace	17.2%	4.1%	9.3%
	MAMBO RAIBI	16.7%	3.5%	8.8%
	MAMBO TRIBI 1	14.9%	2.7%	7.6%
	MAMBO TRIBI 6	**14.3%**	**2.5%**	**7.2%**
Hikey 960	DynamoRIO	47.0%	12.2%	25.5%
	MAMBO	23.3%	4.8%	12.1%
	Contiguous Trace	19.1%	4.0%	10.0%
	MAMBO RAIBI	23.4%	4.1%	11.7%
	MAMBO TRIBI 1	**16.1%**	**2.2%**	**7.7%**
	MAMBO TRIBI 6	16.4%	2.6%	8.1%

The lowest overhead for each benchmark appears in **bold**.

The microarchitecture plays a significant role in the performance of the benchmarks. The Contiguous Traces generation manages to shorten the critical path and reduces the number

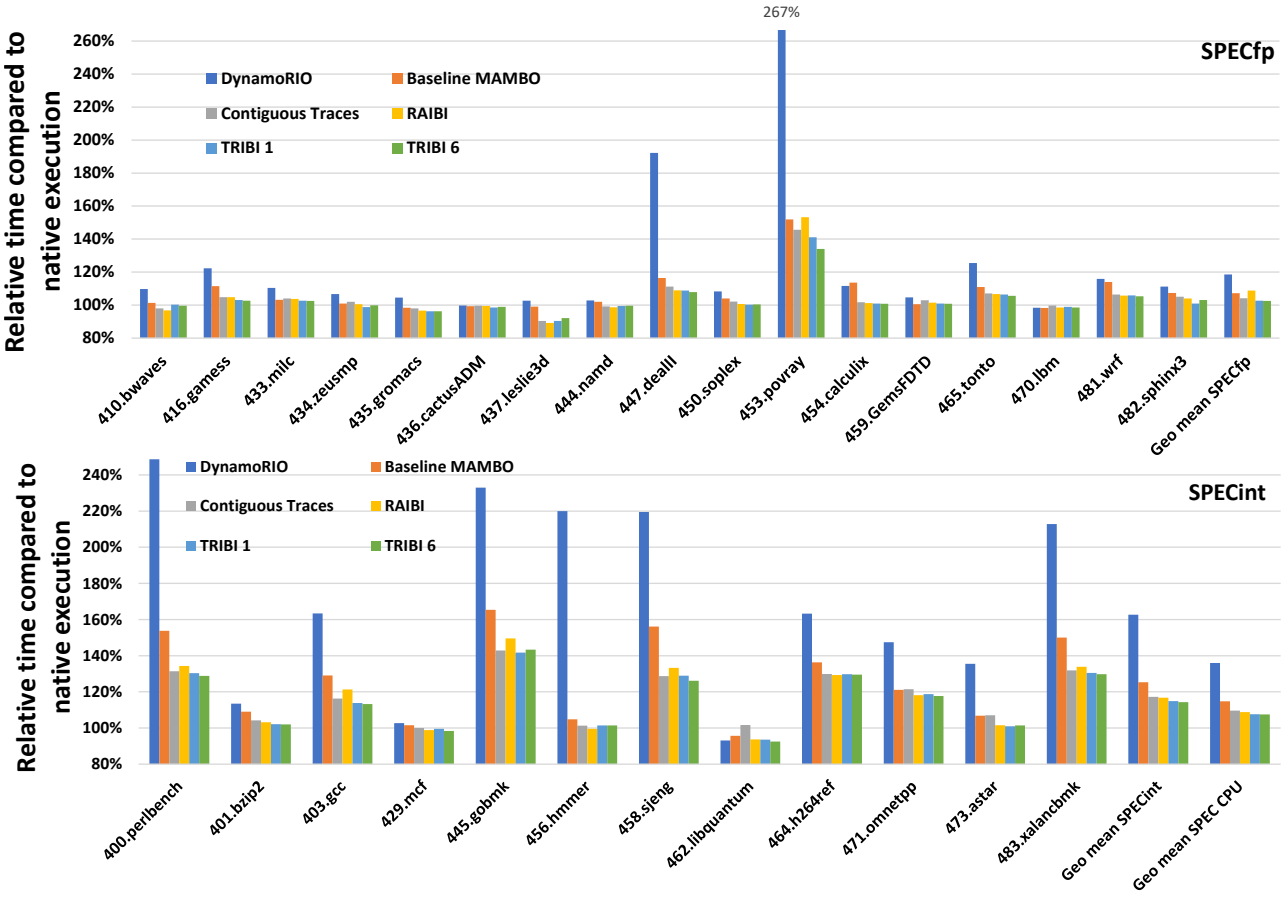

Figure 6. Relative execution time for SPEC CPU2006 on Rock 960 (Cortex-A72) system.

of branches with respect to the baseline implementation. A compact critical path and also a lower number of branches in the critical path even when they are not architecturally executed are crucial for improving the performance.

4.2.2 TRIBI

Table 3 shows an overall of the geometric mean overhead for TRIBI for 1 to 6 targets. The MAMBO TRIBI optimisation also manages to reduce overhead on all systems. The performance improvement ranges from 10% to 30% for the configuration with the lowest overhead compared to MAMBO Contiguous Traces. The improvement in performance highlights the high cost paid for SPC to TPC translations at runtime for indirect branches.

Inlining 1 target has the biggest impact and as more targets are inlined the gains become smaller. The reason for this is that inlining 1 target does not mean the first target seen by the branch. Because the prediction is only inserted when the translated target has been promoted to a trace. Basic blocks are promoted to a trace when they execute above a threshold. The default threshold in MAMBO is 256 times, and how to select it was explored in [16]. In case a branch has a relative high number of targets, but only one is dominating the execution, only the dominating target would be inserted.

Moreover, the difference between the TRIBI 6 and the RAIBI, is that TRIBI 6 inserts predictions using only direct

Table 3. Performance overhead of RAIBI and TRIBI with 1 to 6 indirect branch inlining targets running SPEC CPU2006 on all the systems.

			TRIBI Targets					
System	Benchmark	RAIBI	1	2	3	4	5	6
Pine A64+	SPECint	20.2%	16.2%	16.4%	16.3%	16.2%	16.3%	**16.2%**
	SPECfp	5.3%	4.4%	4.4%	4.3%	5.3%	**4.0%**	4.1%
	CPU	11.2%	9.3%	9.2%	9.1%	9.6%	**9.0%**	**9.0%**
Applied Micro X-Gene 2	SPECint	22.7%	21.4%	21.2%	20.9%	20.7%	20.7%	**20.6%**
	SPECfp	4.7%	5.0%	4.6%	4.5%	4.3%	4.4%	**4.2%**
	CPU	11.8%	11.5%	11.2%	11.0%	10.8%	10.8%	**10.7%**
Jetson TX1	SPECint	23.3%	20.0%	19.0%	19.7%	20.0%	19.0%	**18.7%**
	SPECfp	4.8%	4.4%	4.1%	4.3%	3.8%	3.8%	**3.6%**
	CPU	12.1%	10.6%	10.0%	10.4%	10.2%	9.9%	**9.6%**
Rock 960	SPECint	16.7%	14.9%	14.7%	15.0%	14.9%	15.0%	**14.3%**
	SPECfp	3.5%	2.7%	3.0%	2.4%	2.5%	**2.3%**	2.5%
	CPU	8.8%	7.6%	7.7%	7.4%	7.4%	7.4%	**7.2%**
Hikey 960	SPECint	23.4%	16.1%	17.0%	16.0%	**15.5%**	16.8%	16.4%
	SPECfp	4.1%	**2.2%**	**2.2%**	2.5%	2.7%	**2.2%**	2.6%
	CPU	11.7%	**7.7%**	8.1%	7.9%	7.8%	8.0%	8.1%

The lowest overhead of each benchmark appears in **bold**.

Moreover, the difference between the TRIBI 6 and the RAIBI, is that TRIBI 6 inserts predictions using only direct

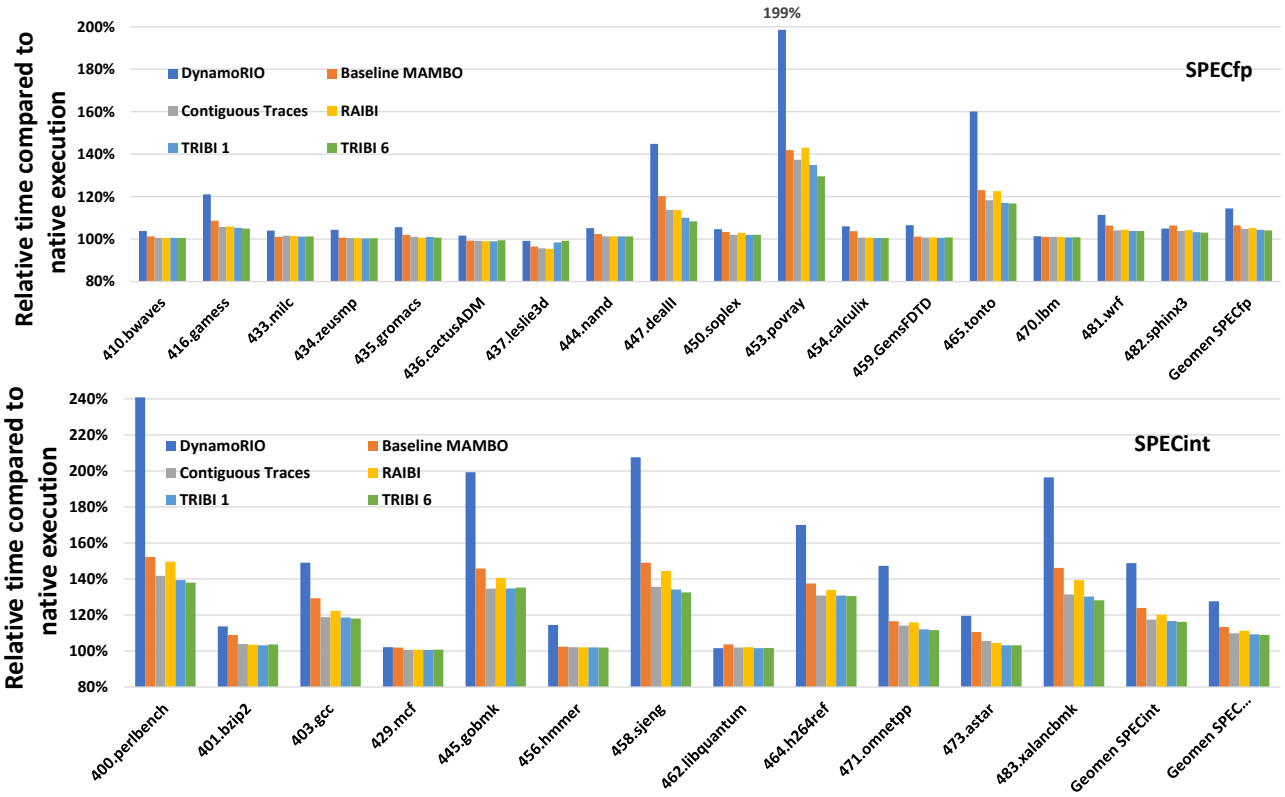

Figure 7. Relative execution time for SPEC CPU2006 on PineA64+ (Cortex-A53) system.

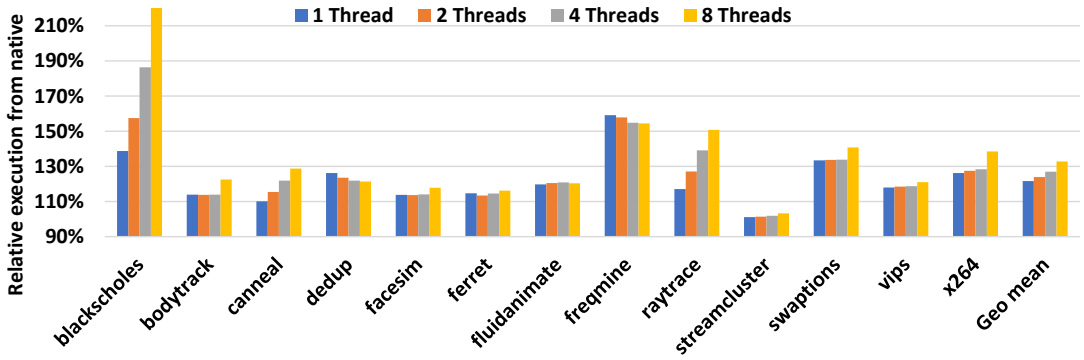

Figure 8. Scalability results — Relative execution time for PARSEC 3.0 on the Applied Micro X-Gene 2 system with baseline MAMBO.

branches and RAIBI uses a single prediction that relies on one indirect branch. This means that, while the former can rely on a branch history pattern-based prediction scheme using a Global History Buffer (GHB) [13], the latter is limited to a prediction mechanism based solely on the last taken branch, for which a Branch Target Address Cache is usually used [25]. For example, if the execution follows a short pattern (equals or shorter than the number of predictions inserted), the branch predictor, after a brief training period can successfully predict the branches in the predictions. This could be the main factor behind the reduced branch mispredictions observed on TRIBI 6. Increasing the number of target predictions have minor improvements up to 6 targets. Adding more than 6 targets prediction produces higher overhead on most of the systems except for the Cortex-A72 and A73 where it stays the same up to 9 target predictions.

Table 4 presents the relative performance counters for a group of benchmarks from SPEC CPU2006 against MAMBO (baseline) on the PineA64+. The 7 benchmarks have been selected as they have the more than 30% overhead with baseline MAMBO (see Figure 7). (400.perlbench: 52%, 445.gobmk:

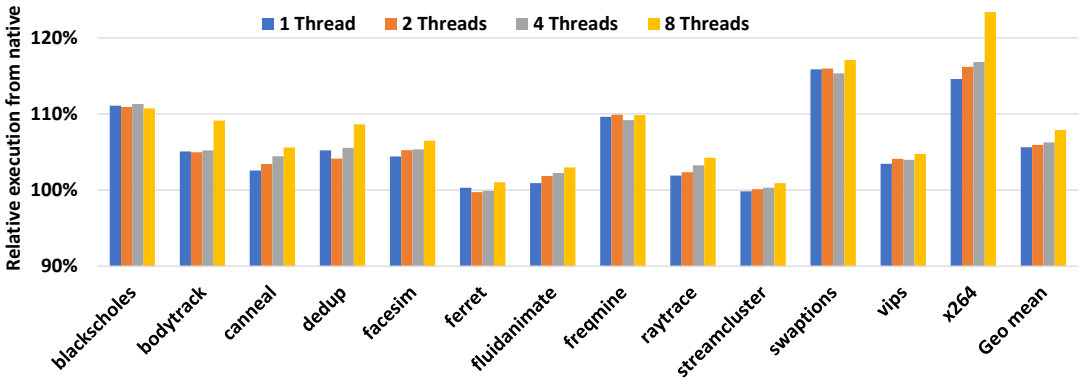

Figure 9. Scalability results — Relative execution time for PARSEC 3.0 on Applied Micro X-Gene 2 system with Contiguous Traces and the TRIBI 6 optimisation enabled for MAMBO.

Table 4. For a subset of SPEC CPU 2006 benchmarks the performance counters recorded for optimised MAMBO (TRIBI 6) relative to MAMBO baseline on the PineA64+ system.

Benchmark	Overhead	Cycles	Instructions	L1I Access	L1I Miss	L1D Access	L1D Miss	Branches	Branch Miss
400.perlbench	52% to 42%	0.91	0.95	0.84	0.93	0.98	0.93	0.39	0.84
445.gobmk	46% to 35%	0.93	0.96	0.81	1.11	0.99	0.93	0.45	0.90
458.sjeng	49% to 36%	0.89	0.95	0.83	1.40	0.95	0.81	0.48	0.92
464.h264ref	37% to 31%	0.95	0.97	0.91	0.95	0.99	0.95	0.58	0.97
471.omnetpp	17% to 14%	0.96	0.92	0.78	1.13	0.95	0.91	0.43	1.09
483.xalancbmk	46% to 31%	0.88	0.95	0.83	1.10	0.95	0.93	0.37	0.91
453.povray	42% to 37%	0.92	0.93	0.88	1.26	0.94	0.91	0.56	1.06

For the performance counter columns, a value of 1 means no improvement. A value below 1 shows an improvement for optimised MAMBO. The Overhead column shows the overhead with MAMBO baseline and the overhead with MAMBO TRIBI 6 and contiguous traces.

46%, 458.sjeng: 49%, 464.h264ref: 37%, 471.omnetpp: 17%, 483.xalancbmk: 46% and 453.povray: 42%.) The Overhead Δ field in Table 4 shows the reduction in overhead from baseline MAMBO. The performance counters show a reduction on the majority of the reported counters. The layout of the Contiguous Traces reduces the number of branches in the hot path, which is reflected in the reduction on the number of branches reported. TRIBI 6 transforms indirect branches into direct branches, which are easier to predict for a branch predictor resulting in a lower number of branch misses. Also, the fewer executions of the Indirect Branch Table Lookup means a reduction on the number of instructions and fewer access to data memory, reducing the accesses and misses on the level 1 data cache.

5 Related Work

Traces are a common optimisation introduced in Dynamo [6] by the NET scheme [14]. The baseline MAMBO uses a modified version of NET, in which traces are terminated on indirect branches. The baseline MAMBO uses an interleaved trace layout with trace exits within the instruction stream. We have optimised the trace code layout by having a contiguous stream of instructions with exits located outside the critical path. The code layout is similar to that described in Dynamo [6], but we have adapted the runtime algorithm for the trace generation. In NET, correctly maintaining the exit points and including extra counters for profiling adds expensive overhead and complexity. Our adapted trace generation and layout do not suffer from this. In addition, NET constructs traces across indirect branches and optimistically transforms them into conditional indirect branches [6]. In MAMBO, an indirect branch is a termination condition when building traces, thus, avoiding the optimistic assumption of NET. In other words, indirect branches are addressed by the TRIBI optimisation, rather than being compounded into the trace construction.

IBI is an optimisation for predicting the targets of indirect branches used by many Dynamic Binary Modification/Translation systems [6] [8] [21]. However, if not implemented efficiently, mispredictions and the high cost of updating the predicted targets can overcome the performance improvement and hurt performance.

DynamoRIO [8] evolved from Dynamo keeping the NET scheme for traces, and is a well-established DBM system supporting Linux, Windows and Android as well as the x86, x86_64 and Arm-32 and -64 bit architectures. The effort of supporting the Arm architecture started in 2014, but for Arm-64 it does include traces. The results using DynamoRIO provide a reference point to understand the performance of

MAMBO. The evaluation has shown that DynamoRIO has an overhead of 25.5%-35.1% geometric mean overhead for SPEC CPU2006, always more than three times the overhead of the new optimisations in MAMBO. Considering the worst case overheads, between 50% and 267%, we have observed them in 10 of the SPEC CPU 2006 benchmarks. The worst observed overheads for MAMBO are below 55%.

Hiser *et al.* [18] evaluates an IBI approach using different strategies for different types of indirect branches, including online profiling and dynamically setting the number of predictions at runtime. They used the Strata dynamic translation infrastructure [24] on Intel, AMD and SPARC architectures. Our implementation uses a fixed number of predictions set at compile time. The actual predictions (based on the execution history) are dynamically inserted only for targets which are *hot*; already promoted to a trace. Hise *et al.* [18] did not consider this restriction for the targets. Furthermore, whereas Hise *et al.* consider 3 architectures (not including Arm), we evaluate the performance of TRIBI on the Arm architecture by using different microarchitecture implementations, from in-order to more aggressive out-of-order cores.

Dhanasekaran *et al.* [12] proposed the Most Recently Used algorithm (MRU) with inline prediction checks done at the start of traces, producing a chain of predictions set to a configurable threshold, but using this they did not managed to gain any performance. In MRU the source target and a prediction are stored in a memory location which is checked on branch execution. On a hit, execution branches to the predicted translation. On a miss, the execution is transferred to a chain of checks stored at the top of a trace. The fallback mechanism uses a private per branch hash table. Our fallback mechanism is different as we use a modified implementation of the inline hash table lookup. Once the maximum number of targets is reached the prediction mechanism is replaced with an unmodified inline hash table lookup.

In [16], Gorgovan *et al.* evaluated IBI on 32-bit Arm systems and introduced the Adaptive Indirect Branch Inlining (AIBI); the state-of-the art IBI on Arm. However, AIBI for 64-bit MAMBO produces a performance improvement in a small group of benchmarks and a slowdown in others, only slightly reducing the overall geometric mean overhead for SPEC CPU 2006. We modified the AIBI implementation to only store a SPC-TPC prediction if the TPC has been promoted to a trace. We called this implementation RAIBI (Restricted AIBI). TRIBI has been evaluated directly against RAIBI showing a measurable lower execution overhead.

6 Conclusions

We have presented two optimisations which have been designed for Arm 64-bit and implemented in MAMBO. The first optimisation is a trace generation algorithm to create a Contiguous Traces code layout which reduces the critical path size and decreases the number of branches seen by the fetch stage of the processor pipeline. This optimisation manages to reduce the overhead on all the evaluated systems relative to baseline MAMBO running SPEC CPU 2006 by 21% to 72% depending on the system. Also we have presented a new predictor and efficient implementation of IBI. TRIBI manages to reduce the overhead relative to the Contiguous Traces optimisation in all the systems running SPEC CPU 2006 by 10% to 29% when compared with RAIBI, a modified version of the the state-of-the-art AIBI.

The Arm systems used in the evaluation provides a comprehensive coverage of the evolution of different Arm 64-bit microarchitectures (e.g. Arm Cortex-A53, A57, A72, A73). The evaluation has showed that the behaviour of the optimisations vary significantly depending on the characteristics of the processor implementation. This is noticeable by observing the performance of some of the benchmarks. An example of different behaviour is *481.wrf* benchmark, which has the lowest overhead on the Jetson TX1 system (Cortex-A57). Another example is *462.libquantum* which has the lowest overhead on the Rock 960 system (Cortex-A72).

Our evaluation has showed a geometric mean overhead for SPEC CPU2006 of 9% for a Cortex-A53 (in-order microarchitecture, first generation 64-bit Arm), 11% for an X-Gene-2 (server) and 10% for a Cortex-A57 (mobile processor) — both first generation out-of order 64-bit Arm microarchitectures — 7% for a Cortex-A72, and 8% for a Cortex-A73 (recent out-of-order processors for mobile devices) when compared to native execution. This is a reduction of the overhead of between 30% to 50% compared to the publicly available version of MAMBO which already included traces (interleaved) and optimisations for indirect branches. Using PARSEC 3.0, we have also evaluated the parallel scalability on the X-Gene-2 (the system with the highest number of cores) and show a geometric mean overhead between 6-8%. These results show the lowest overhead for a DBM system on the Arm architecture.

Acknowledgments

This work is partially supported by an Arm/EPSRC iCASE PhD scholarship (Guillermo Callaghan), and EPSRC Rain Hub EP/R026084/1, and LAMBDA EP/N035127/1 projects. Mikel Luján is supported by an Arm/RAEng Research Chair award and is a Royal Society Wolfson Fellow.

References

[1] ARM. 2013. big.LITTLE Technology: The Future of Mobile. https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf. [Online; accessed 7-April-2019].
[2] ARM. 2015. Cortex®-A72 Software Optimization Guide.
[3] ARM. 2016. Cortex-A57 Software Optimization Guide.
[4] ARM. 2017. Architecture Reference Manual: ARMv8 for ARMv8-A architecture profile.
[5] ARM. 2018. ARM® Cortex®-A75 Software Optimization Guide.
[6] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of*

the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. 1–12. https://doi.org/10.1145/349299.349303

[7] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 41–46. http://www.usenix.org/events/usenix05/tech/freenix/bellard.html

[8] Bruening, Derek L. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Cambridge, MA, USA.

[9] Amanieu d'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. 2016. Optimizing Indirect Branches in Dynamic Binary Translators. *TACO* 13, 1 (2016), 7:1–7:25. https://doi.org/10.1145/2866573

[10] Amanieu d'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. 2017. Low overhead dynamic binary translation on ARM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 333–346. https://doi.org/10.1145/3062341.3062371

[11] Deutsch, L Peter and Schiffman, Allan M. 1984. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 297–302.

[12] Dhanasekaran, Balaji and Hazelwood, Kim. 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.

[13] Driesen, Karel and Holzle, Urs. 1998. Accurate indirect branch prediction. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*. IEEE, 167–178.

[14] Duesterwald, Evelyn and Bala, Vasanth. 2000. Software profiling for hot path prediction: Less is more. *ACM SIGOPS Operating Systems Review* 34, 5 (2000), 202–211.

[15] Gorgovan, Cosmin and d'Antras, Amanieu and Luján, Mikel. 2016. MAMBO: a low-overhead dynamic binary modification tool for ARM. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 14.

[16] Gorgovan, Cosmin and d'Antras, Amanieu and Luján, Mikel. 2018. Optimising Dynamic Binary Modification Across ARM Microarchitectures. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 28–39.

[17] Hazelwood, Kim and Klauser, Artur. 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.

[18] Hiser, Jason D and Williams, Daniel W and Hu, Wei and Davidson, Jack W and Mars, Jason and Childers, Bruce R. 2011. Evaluating indirect branch handling mechanisms in software dynamic translation systems. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 2 (2011), 9.

[19] Intel Corporation. 2016. Intel® 64 and IA-32 architectures software developer's manual. *Volume 3 (3A, 3B, 3C & 3D): System programming Guide* (2016).

[20] Kim, Ho-Seop and Smith, James E. 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.

[21] Luk, Chi-Keung and Cohn, Robert and Muth, Robert and Patil, Harish and Klauser, Artur and Lowney, Geoff and Wallace, Steven and Reddi, Vijay Janapa and Hazelwood, Kim. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[22] Nethercote, Nicholas. 2004. *Dynamic Binary Analysis and Instrumentation*. Ph.D. Dissertation. PhD thesis, University of Cambridge.

[23] Probst, Mark. 2002. Dynamic Binary Translation. In *UKUUG Linux Developer's Conference*, Vol. 2002.

[24] Scott, Kevin and Davidson, Jack. 2001. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation*.

[25] Jurij Šilc, Theo Ungerer, and Borut Robic. 2007. Dynamic branch prediction and control speculation. *International Journal High Performance Systems Architecture* 1, 1 (2007), 12–13.

[26] Jim Smith and Ravi Nair. 2005. *Virtual Machines: versatile platforms for systems and processes*. Elsevier.

[27] Spradling, Cloyce D. 2007. SPEC CPU2006 benchmark tools. *ACM SIGARCH Computer Architecture News* 35, 1 (2007), 130–134.