Evaluating the Impact of Optimizations for Dynamic Binary Modification on 64-bit RISC-V

DOI:
10.1109/ISPASS57527.2023.00017

Evaluating the Impact of Optimizations for Dynamic Binary Modification on 64-bit RISC-V

John Alistair Kressel, Guillermo Callaghan, Cosmin Gorgovan, and Mikel Luján
Department of Computer Science, *University of Manchester*, M13 9PL, United Kingdom.

***Abstract*—Dynamic Binary Modification (DBM) is an important technique used in computer architecture simulators, virtualization, and program analysis, to name a few examples. The software ecosystem of RISC-V is maturing at pace, but is still missing a high-performance, optimized DBM. Addressing this requirement is key to improving the overall software ecosystem.**

This paper presents a comprehensive performance evaluation study for a DBM (MAMBO) which has been ported and optimized for 64-bit RISC-V. The main optimizations for DBM on RISC architectures have been implemented and tuned for RISC-V to address specific architectural features. For example, *jump trampolines* have been specifically developed to address the short direct branch range specified by the RISC-V ISA.

The evaluation shows that for SPEC CPU2006 the geometric mean overhead is of 14.5%, with SPECint having the largest contribution with a geometric mean of 28.5%, while SPECfp has only an overhead of 5.6%. Concretely, this results in a reduction in runtime for *h264ref* from over 75 hours using the baseline DBM, to 2.2 hours with optimizations applied.

***Index Terms*—Dynamic Binary Instrumentation, Dynamic Binary Modification, Binary Code Profiling, RISC-V**

I. Introduction

Dynamic Binary Modification frameworks are an essential part of the software development ecosystem, they enable powerful applications, such as binary instrumentation (DBI) [1] [2], error detection [3] [4], virtualization [5] and binary translation [6] [7]. By acting directly on compiled application binaries and modifying them at runtime, DBM frameworks are valuable for software development and analysis, facilitating development of high-quality, high-performance software. The RISC-V software ecosystem is maturing, but still has some gaps. There is no DBM optimized for RISC-V.

DBM frameworks generate a performance overhead onto a hosted binary, compared to the same binary running natively on a processing core. This is partly as a consequence of executing additional instructions which are needed to load, execute and modify a binary. Further, micro-architectural inefficiencies caused by the modification of binaries, such as increases in the number of instruction cache misses and branch prediction misses also contribute to the performance overhead.

MAMBO [8], [9] is a high-performance DBM framework that was initially developed and optimized to support 32-bit ARM [10] was subsequently extended and optimized to support the 64-bit ARM architecture [11]. We detail the development of optimizations for the RISC-V port of MAMBO, addressing architectural requirements specific to RISC-V, including the short address range of conditional and unconditional branch instructions.

Handling position dependent instructions (such as branches) is a major cause of overhead in DBM frameworks. DBM frameworks use a software code cache where the instructions from a binary file are loaded. Such position dependent instructions need to be modified to reflect their new position within the DBM code cache. This paper examines how existing optimizations for DBMs (i.e. branch linking, inline hash lookup and Trace Restricted Indirect Branch Inlining (TRIBI)) are adapted and tuned for the RISC-V architecture to reduce the performance overhead associated with running under the control of a DBM framework.

An architectural feature of RISC-V, which affects DBM frameworks, is a relatively short range of ±1MB for direct branches [12]. As a result, direct branches (`JAL`) cannot always be used when implementing optimizations. Sometimes these need to be replaced by direct register jumps (`JALR`) when a target in the code cache exceeds the ±1MB range even when the original address in the original binary did not. We describe *jump trampolines*, a means of addressing this limitation, removing the need for a `JALR` within a configurable range, instead using a series of `JAL`s, thus avoiding the additional overhead associated with register jumps.

Traces have been used successfully in ARM implementations of MAMBO [10] [11] and other DBM/DBI frameworks such as DynamoRIO [13] and PIN [2] to reduce the number of instruction cache misses caused by code cache fragmentation, by organizing frequently executed, hot code into contiguous memory blocks called traces, which are able to make better use of the instruction cache. We describe RISC-V implementations of traces and contiguous traces [11].

The contributions of this paper are summarized as:

- RISC-V specific implementations and tuning of branch linking, inline hash lookup, traces, contiguous traces and TRIBI, resulting in the first optimized DBM framework for RISC-V, compatible with `RV64GC`.
- The inefficiency caused by the short range of direct branches is addressed with jump trampolines. Using this approach results in a 5.6% reduction of geometric mean overhead when traces are disabled, with performance similar to traces for SPECfp.
- The evaluation of the presented optimizations and their

effect using system performance counters, demonstrating a geometric mean overhead of just 14.5% when running SPEC CPU2006.

This paper uses SPEC CPU2006 as the existing literature for DBI/DBM has been evaluated with this benchmark suite. Section II describes the basics of MAMBO. Sections III and IV show how to implement the optimizations addressing RISC-V. Section V presents the performance evaluation. Sections VI and VII introduce the related work and presents the summary of the paper, respectively.

II. Baseline DBM System Overview

MAMBO loads the target binary into memory and then scans and copies its instructions, storing them in a code cache as basic blocks, where modifications can be applied. The RISC-V port of MAMBO uses a modified scanner to support RISC-V instructions. Most instructions can be copied unmodified, but position dependent instructions must first be modified to reflect their new position within the code cache. In addition to storing basic blocks, the code cache also stores metadata which describes basic blocks, including the target and fall-through addresses of conditional branches. Additionally, a hash table data structure maps the address of the source binary or Source Program Counter (SPC) to the corresponding address in the code cache, or Translated Program Counter (TPC). This mapping allows MAMBO to efficiently transfer control between basic blocks whilst maintaining the original addresses used in the application.

Basic blocks end with a branch which may be direct, indirect or conditional, initially these are translated into a branch that transfers control back to the dispatcher of MAMBO. Once control is returned from the hosted application, the dispatcher can perform a hash table lookup to identify if the called basic block is already present in the code cache, or scan and add a new basic block corresponding to the SPC. The baseline used in this paper applies conditional branch linking, this replaces calls to the dispatcher of MAMBO for conditional branches with conditional branches using the corresponding TPC.

III. Branch optimizations

This section describes the branch linking, inline hash lookup and jump trampolines optimizations, which describe how MAMBO handles branch instructions.

A. Direct branch linking

RISC-V specifies two types of unconditional branch instruction – Jump And Link (`JAL`) and Jump And Link Register (`JALR`). A `JAL` instruction has a 20-bit immediate, giving a range of ±1MB. This limited range may sometimes necessitate the use of a `JALR` instruction, which requires that the target address is first loaded into a register, before branching to that address. This gives `JALR` instructions a much greater range.

Direct branch linking exploits the fact that the branch target is unchanging. The original branch instruction had a fixed target address in the source binary, which is replaced with the address of the new target in the code cache (TPC), thus removing the need to transfer control to MAMBO.

In implementing direct branch linking, one must consider that at the time of scanning, the target basic block may not yet exist in the code cache. It may also be the case that the target basic block is outside of the ±1MB range allowed by a `JAL`. In order to handle this gracefully, the following steps are taken:

- The original target address of the branch is recorded in the code cache metadata.
- A branch to the dispatcher is inserted as before.
- The dispatcher looks up or scans the target of the branch.
- The offset of the target from the current location in the code cache is calculated.
- For an offset in range of ±1MB, a `JAL` is used, overwriting the current branch to the dispatcher.
- For an offset greater than ±1MB, it is necessary to use a `JALR`.

When using a `JALR` in place of a `JAL`, the TPC must be loaded into a register and a series of instructions must be emitted into the code cache. First registers `x10` and `x11` are pushed to the stack, then the TPC is loaded into `x10`, finally the `JALR` instruction is executed. When no `JALR` is needed, the TPC is incremented by 12 bytes (3 instructions) in order to skip the pop instructions at the beginning of the target basic block, since no registers were pushed.

This algorithm has been adapted from the 64-bit ARM implementation of branch linking, however it differs in two key ways. Firstly, the 64-bit ARM implementation does not need to consider register branches because the range of a direct branch is greater than the maximum size of the code cache. Secondly, on RISC-V three instructions are skipped when using a `JAL` (two load instructions to pop registers off the stack, and an add instruction to increment the stack pointer), whereas only one must be skipped on 64-bit ARM (a single load pair instruction performs the same task as the three instructions on RISC-V).

B. Inline Hash Lookup

Inline Hash Lookup (IHL) is an optimization designed to reduce the overhead introduced by the handling of indirect branches. An indirect branch uses an address which is only known at runtime. The target address of an indirect branch may vary upon each execution, meaning that the SPC must be translated each time.

The default method for handling this translation is to insert a call to the dispatcher of MAMBO. While this method will produce correct results, it is also inefficient since it requires a context switch. In order to translate a SPC to a TPC, the hash table is needed. A call to the dispatcher results in an expensive hash lookup, this only makes sense if the target has not yet been scanned. It is clear that in order to perform this lookup, a dispatcher call is normally superfluous. Indeed, IHL removes the need for a context switch by performing the hash table lookup, inline. The standard MAMBO calculation for the hash table key is

$$(\mathrm{SPC} >> \mathrm{HT\_SHIFT})\ \&\ \mathrm{CODE\_CACHE\_HASH\_SIZE}$$

where HT SHIFT is the number of bits by which to shift the SPC right. If the RISC-V ‘C’ extension is available, this is set to 1, if it is not available then HT SHIFT is set to 2. This is needed to satisfy memory alignment requirements; either the least significant bit of an address in the case of RISC-V ‘C’, or the two least significant bits of an address when the ‘C’ extension is not available, will always be 0 and thus are not needed when calculating a hash key. This is due to the fact that RISC-V ‘C’ relaxes memory alignment requirements and allows instructions to be aligned on 16-bit boundaries, rather than the usual 32-bit requirement. The implementation of IHL is a direct port of the existing MAMBO IHL algorithm [11].

C. Jump Trampolines

When linking direct branches, all targets would ideally be within range of a `JAL` instruction, meaning that only a single instruction is needed. However, as described in Section III-A, sometimes a `JALR` is needed. Switching from a `JAL` to a `JALR` is costlier than simply loading the target address into `x10`. To quantify the performance deficit resulting from the use of a JALR instead of a JAL, a micro-benchmark was run comparing a JAL to a JALR (including instructions to load the address into a register). A JALR was 11% slower on average running on a SiFive U74 processing core. The two pop instructions at the beginning of the target basic block and the corresponding push instructions needed must also be considered, since these can potentially be the source of overhead. In order to reduce this impact, jump trampolines are introduced. Making use of empty space at the end of existing basic blocks, a number of `JAL` instructions are inserted, acting as trampolines to reach the target. This reduces the number of memory accessing instructions required by avoiding the use of a `JALR` instruction. This optimization has been developed for MAMBO on RISC-V.

Each trampoline could potentially hold as many branch instructions as there is free space at the end of a basic block, but this will vary greatly between basic blocks. To simplify the logic, trampolines are restricted to 6 slots since this represents a good balance between providing sufficient slots in each trampoline and ensuring that more basic blocks have the free space needed for a trampoline. Analysis of SPEC CPU2006 *gcc*, which tends to have a large number of basic blocks, revealed that with 6 trampoline slots, most basic blocks had enough free space available. This number can be increased or decreased as needed.

1) Identifying Trampoline Basic Blocks: Trampoline capable basic blocks have space for at least 6 standard RISC-V instructions following its constituent instructions. These are marked as being available as trampolines, once all instructions have been scanned.

2) Algorithm: When linking a direct branch which has an offset greater than 1MB, it is checked to see if the offset is less than 3MB. This maximum has been chosen to prevent the number of branches growing too large and outweighing the benefit seen from the reduction in the number of memory accessing instructions. If a branch meets these conditions, a series of trampoline jumps are constructed. Starting at the target, the first branch to be inserted into a trampoline slot which is as close to 1MB from the target as possible, with that branch pointing to the target address. The next branch to be inserted is as close to 1MB from the previous trampoline slot as possible and points to the previously inserted branch. This is repeated until a branch is inserted within 1MB of the starting branch. Finally, the original direct branch is set to target the first trampoline slot branch. Figure 1 illustrates this. Here, the algorithm starts at the target which is BB#4, from there it finds the next available trampoline slot which is as close to 1MB away as possible. This corresponds to BB#3 slot 1, so a branch from BB#3 slot 1 to BB#4 is inserted. The algorithm continues working backwards until a trampoline within 1MB of the source basic block is found, which in the diagram corresponds to BB#2 slot 1. This means that all necessary trampolines have been inserted and a branch is now inserted as normal at the end of BB#1 targeting this trampoline.

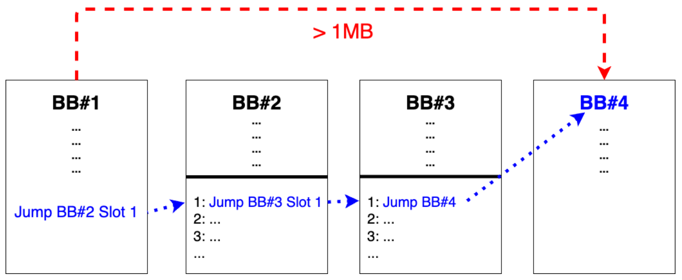


Fig. 1: Example of Jump Trampolining. The red dashed line represents the desired jump.

IV. Advanced Optimizations

This section describes trace and contiguous trace optimizations. These have been applied to reduce software code cache fragmentation of MAMBO and thus reduce the hardware instruction cache misses. Finally, the implementation of TRIBI is detailed which builds upon trace optimizations to reduce the indirect branch handling overhead further. All three optimizations have been adapted from their original implementations in the ARM versions of MAMBO.

A. Traces

When code is scanned and basic blocks are created, they are added to the software code cache in the order in which they are first called. This can cause significant fragmentation within the software code cache, since the order in which basic blocks are first accessed may bear no relation to the paths most frequently taken (hot code path). Modern processors maintain in hardware a small instruction cache which is both physically closer to the processing core than main memory and significantly faster. Software which is optimized to make better use of the instruction cache, potentially has a performance advantage over a program which is less well optimized and would, therefore, spend more time waiting for the required instructions to arrive from memory.

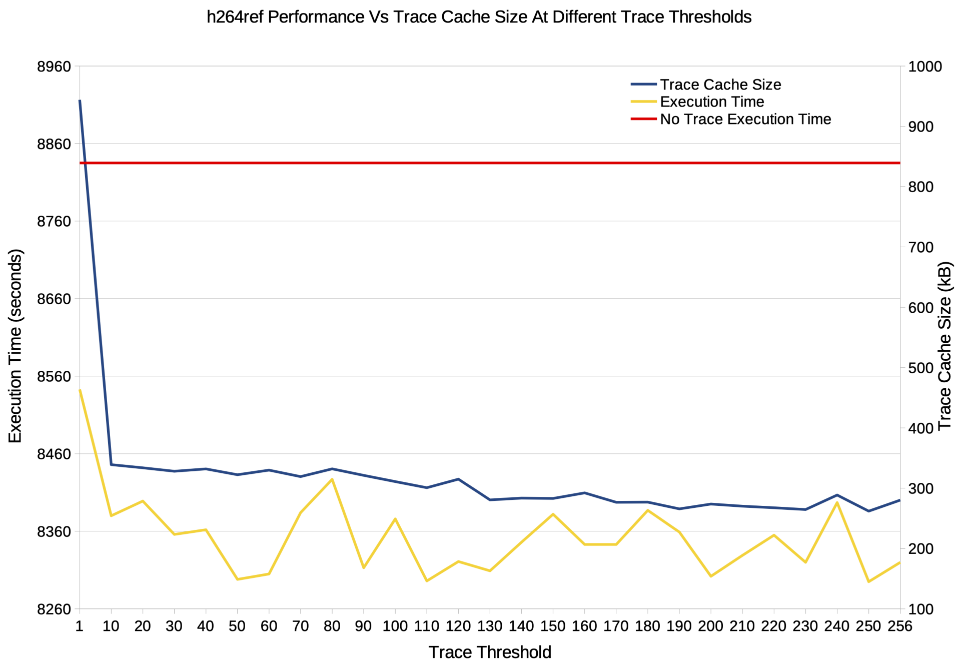


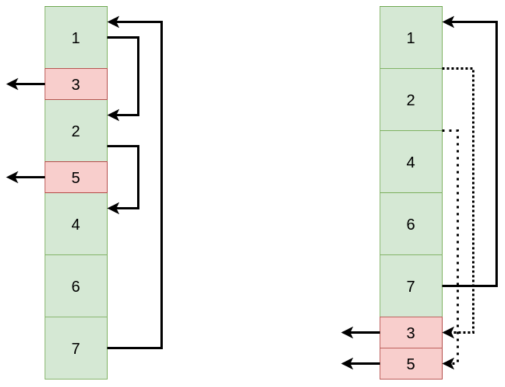
Fig. 2: Size of the trace cache vs. runtime at different trace creation thresholds for *h264ref*.

To improve the instruction cache utilization, MAMBO implements traces with the goal of reducing the size of the critical path by grouping basic blocks together into traces, sometimes referred to as superblocks [3]. Traces are made up of *fragments* which are single entry, single exit units of code. RISC-V processors such as the FU740, also implement a fast instruction cache [14], meaning that a trace cache implementation targeting RISC-V is desirable. The algorithm used is a modified version of the Next Executing Tail (NET) algorithm [15].

Constructing a trace requires each component basic block to be re-scanned, so a significant cost is associated with the creation of cache traces, which needs to be amortised over time through improved performance. In order to increase the chances of this cost being amortised, suitable basic blocks must first be identified.

1) Traces – RISC-V Considerations: MAMBO increments a counter at the beginning of each eligible basic block, recording the number of times it has been executed. Once a certain threshold is reached, a basic block is regarded as being hot and a trace is constructed. This threshold is a parameter which can be adjusted. The counter is implemented as an unsigned 8-bit number resulting in a maximum threshold of 256. For example, Figure 2 shows the runtimes of *h264ref* against the trace cache size at different threshold values. The analysis shows that the increasing threshold value does not produce a trend in the runtime, however the trace cache size observed tends to decrease. The value 256 is used as the maximum for these reasons.

The trace head is the first basic block to be identified in a hot path, this block is scanned and placed into the trace cache



(a) A regular trace. (b) A contiguous trace.

Fig. 3: The difference in layout between a trace and contiguous trace is illustrated. Green sections represent trace fragments, while the red sections are trace exits.

aligned to a 4 byte boundary. Before emitting the first trace fragment, instructions to pop `x10` and `x11` are inserted in the same way as at the beginning of a basic block. However, even though a trace consists of many fragments, it will only have one entry point. The trace represents a path of execution and, therefore, needs only one set of pop instructions. Once the trace head has been scanned, execution resumes with the newly scanned code being executed. As with a newly scanned basic block, the scanner inserts a call to the dispatcher in place of the original exit branch, which is overwritten either by a trace exit or by the next fragment. This allows the trace to follow along the path of execution, with each target along this path added to the trace until a terminating condition is encountered. Returning from the dispatcher requires registers `x10` and `x11` to be popped. Therefore, at the end of each scanned fragment, a set of pop instructions is included. This is followed by a branch to the start of the new fragment, allowing the necessary pops to occur without the need for additional instructions to be inserted into the fragment. The pop instructions are overwritten when the next fragment is scanned or a trace exit is inserted, this results in them being discarded after use.

The maximum number of fragments allowed in a trace is set as 20 as per the analysis presented in Figure 4. This value offers the best performance while still supporting large traces. It also prevents traces from becoming too large, potentially going too far down a particular path of execution. A loop could cause significant code duplication. On ARM, MAMBO also uses the same value 20 as the maximum.

B. Contiguous Traces

The motivation for constructing a trace is the reduction in size of the hot code path. However, a trace as described in Section IV-A can contain many trace exits, which may pollute the hot code path with instructions which may not be frequently executed. Contiguous traces as described by Callaghan *et al.* [11], seek to address this by moving trace exits out of the main path of the trace, instead putting all of the instructions needed to exit the trace, at the end of the trace. Now, the only instruction which remains in the main trace, is the conditional branch which performs the condition check to decide if the execution will continue within the trace or will take the trace exit. Previously, in a standard trace, a trace exit was reached as the fall-through address of the condition check, meaning that the condition for remaining in the trace was false. In a contiguous trace, the condition must be inverted since a trace exit no longer needs to be skipped. Instead, when a trace exit is to be taken, the branch jumps to the trace exit which is at the end of the trace. This also means that the branch target needs to be updated to point to the correct trace exit, rather than the next fragment as before.

```
Frag 1: ...
        ...
        ...
        B.COND Frag 2
        JUMP Exit 1
Frag 2: ...
        ...
        (>=4KB)
        ...
Exit 1: JUMP OUT OF TRACE
```

Listing 1: Contiguous traces when a trace is not restriced to 3KB for the main trace. This means that extra `JUMP` instructions are needed in the main trace path.

1) Contiguous Traces – RISC-V Considerations: Conditional branches on RISC-V have a range of ±4KB [12]. This can cause a problem when implementing contiguous traces on RISC-V. In a standard trace, a branch is required to jump at most 80 bytes (20 instructions; i.e. the maximum size of a trace exit) to skip the trace exit and enter the next trace fragment. The variable size of a trace exit stems from the fact that loading an address into a register can require a variable number of instructions on RISC-V depending on the size of the offset to load. A trace fragment may be of arbitrary length, which means that if a fragment or series of fragments exceeds 4KB in size, it would not be possible to reach the trace exit and an additional unconditional branch would be needed, increasing the number of instructions in the hot code path. The second option and the one which has been implemented, is to restrict the maximum size of a trace to 4KB.

However, the main part of a trace cannot reach 4KB in size, to do so would risk certain branches being unable to reach their trace exit. Instead, the main part of a trace is restricted to at most 3KB, with 1KB allowed for trace exits. An example of an unrestricted and restricted contiguous trace can be seen in Listings 1 and 2. The requirement to restrict the size of a contiguous trace was not needed for the 64-bit ARM implementation because a conditional branch has a range of ±1MB [16]. In practice, the majority of traces observed do not exceed 3KB in size. By restricting the size of a trace, only a single branch is needed per exit within the main trace. Due to the maximum number of fragments having been set to 20, only a maximum of 20 instructions relating to trace exits could now be in the main path of the trace (one instruction per fragment).

```
Frag 1: ...
        ...
        B.COND.INVERTED Exit 1
Frag 2: ...
        ...
        (<=3KB)
        ...
Exit 1: JUMP OUT OF TRACE
```

Listing 2: Contiguous traces when a trace is restriced to 3KB for the main trace. This means that no extra instructions are needed in the main trace path. Notice that the branch condition is now inverted to jump to the trace exit rather than the next fragment.

C. TRIBI

Even with IHL applied, handling indirect branches still represents a significant source of overhead. The SPC must be translated each time, even if the target of the branch is always the same, or there are only a small number of targets. TRIBI, proposed by Callaghan *et al.* [11] is a potential solution to this problem and has been modified to work efficiently with RISC-V branches. With TRIBI enabled, a number of prediction slots are made available before the regular IHL instructions, allowing a configurable number of previously taken branch addresses to be stored and used again in the future without the need for a full lookup. If the IHL routine is reached, then it means that the current target has not been seen before and it is inserted in the next available prediction slot. An example

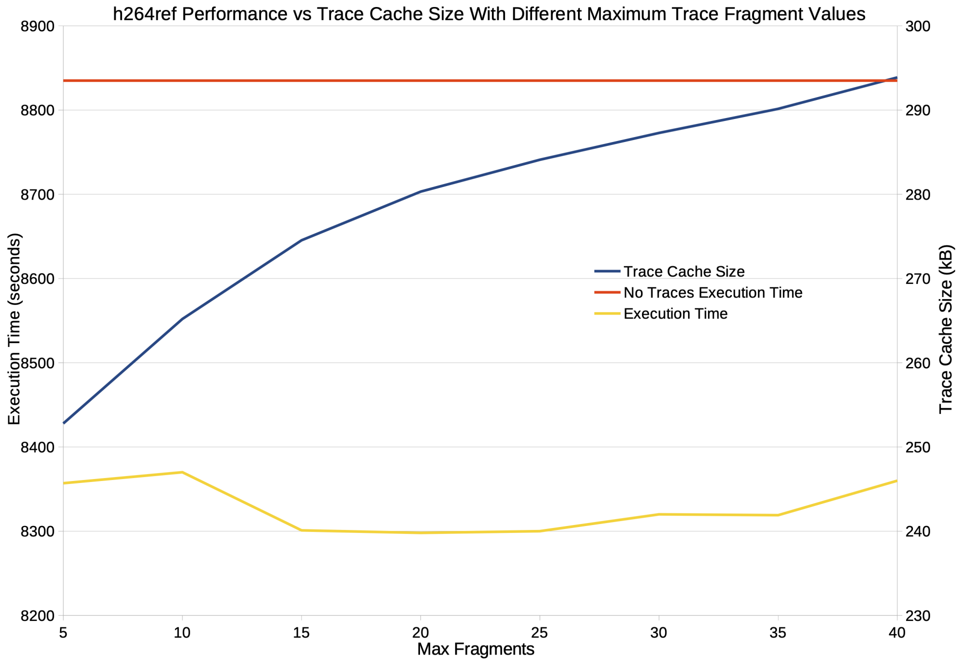


Fig. 4: Size of the trace cache vs. runtime at different maximum trace fragment values for *h264ref*.

of a prediction is shown in Listing 3. If the SPC calculated matches the one stored in a prediction, then the prediction is correct and is used, avoiding a full lookup.

```
          PUSH  x10  and  x11
           ...
slot1:    LOAD  x10,  SPC_1
          BNEQ  x10,  x11,  slot2
          JUMP  TPC_1
slot2:    LOAD  x10,  SPC_2
          BNEQ  x10,  x11,  IHL
          JUMP  TPC_2
IHL:       ...
           ...
```

Listing 3: TRIBI 2 prediction structure, 2 predictions before falling back to IHL

If the IHL routine is reached and the maximum number of predictions has been exceeded, then the prediction mechanism is disabled and the existing predictions are overwritten by the standard IHL routine, and TRIBI is abandoned. Indirect branches with a large number of targets are not suitable for optimization using TRIBI. This is due to the large number of prediction slots that would be necessary to capture all possible targets, resulting in a potential increase in instructions being executed than would be seen in the original case. Alternatively, the predictions made might not be useful if there are a large number of targets, meaning that additional instructions are executed without any benefit. In order to increase the chance that a prediction will be used, predictions are restricted to target traces. This is because a trace has been identified as hot code, which increases the chance that a prediction will be frequently used.

1) TRIBI – RISC-V considerations: When scanning a trace fragment ending in an indirect branch with TRIBI enabled, a gap must be left which is big enough to accommodate all of the prediction slots. For each prediction slot, a gap is left to accommodate, first a load of the SPC corresponding to that prediction into `x10` or `x11`, if `x10` is already in use. Following this, the SPC corresponding to the prediction is compared with the branch target, followed by an instruction to load the TPC address of the prediction into `x10`. This is needed in case the comparison evaluates to true meaning that the prediction is used. Finally a branch instruction is inserted to branch to the prediction TPC address. At the beginning of a series of predictions, `x10` and `x11` are pushed to allow for the manipulation of these registers to perform the comparisons. Ordinarily, when a `JAL` is used, the target address is the start of a basic block or trace +3 instructions. This is done to skip the pop instructions found at the start of each, needed for when a `JALR` is used. However, rather than popping these registers before jumping if a `JAL` can be used for a successful TRIBI prediction, all branches will target the start of a trace, meaning the aforementioned registers are restored by the existing pop instructions at the beginning of a trace. Therefore, space does not need to be wasted with prediction slots holding duplicated pop instructions.

Figure 5 displays benchmarking performed on a subset of SPEC CPU2006 benchmarks, using 1-4 TRIBI predictions. The results found that 4 predictions provides the best performance. For this reason, TRIBI 4 has been used for evaluation in the rest of this paper.

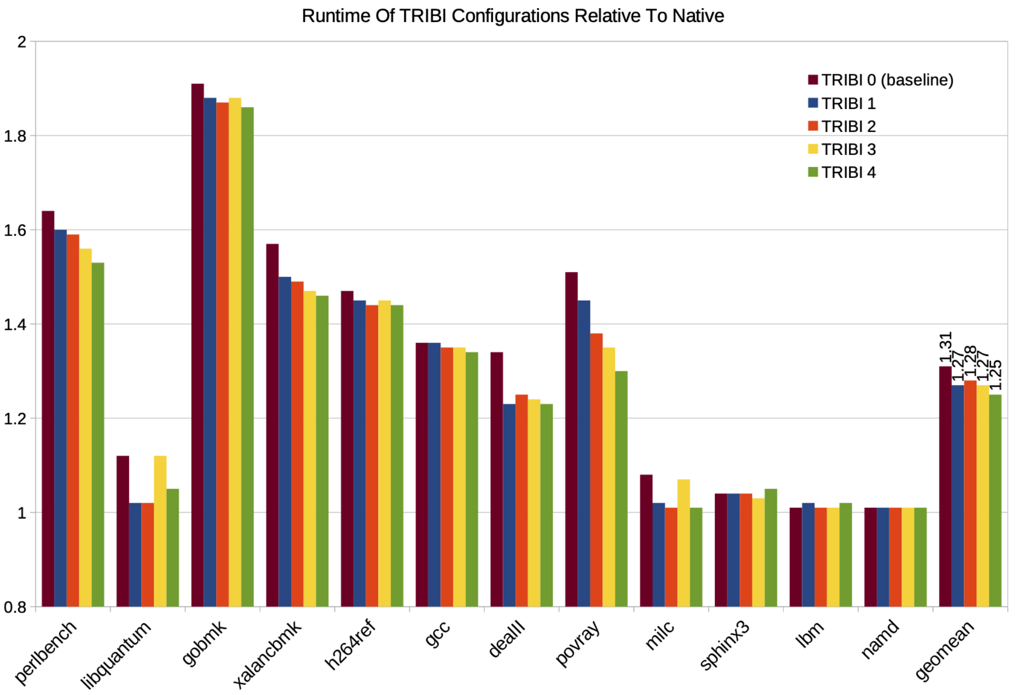


Fig. 5: Performance with 1-4 TRIBI predictions.

V. Performance Evaluation

A. Evaluation Setup

All experiments have been conducted using SiFive Unmatched boards running Ubuntu 20.04.4. These contain the SiFive Freedom U740 SoC [17]. Each U74 application core has a 32 KB private L1 cache. A 2MB L2 cache is shared between cores [14]. The evaluation uses the SPEC CPU2006 suite of benchmarks [18]. This suite has been selected as it tests a wide range of scenarios, and is the main benchmark suite used in the DBI/DBM literature. All benchmarks have been compiled using GCC version 9.4.0 with optimization flag -O2. Additionally, MAMBO was compiled using GCC version 9.4.0, also using -O2.

All reported results are an average of 3 runs. Note that executing one benchmark can take several days for the baseline MAMBO configuration. Thus executing once a benchmark with all the different configurations can take more than one week. Each SPEC CPU2006 benchmark was run using the *ref* workload in order to ensure the most representative runtimes, allowing for marginal improvements or performance degradation to be identified over the longer runs. For clarity, when a trace optimization is applied, branch linking and IHL are also applied. Contiguous traces and traces are mutually exclusive. When TRIBI is applied, so are branch linking, IHL and contiguous traces. Jump trampolines are used in conjunction with branch linking and IHL but never trace optimizations, since this negates the effect of them.

1) Performance Counters: The SiFive Freedom U740 provides two fixed function performance counters and two programmable event performance counters [14]. These count cycles and retired instructions, as well as 34 other events: instruction commit events, micro-architectural events and memory system events. To gain insight into the effect of implemented optimizations, we have chosen he following events: Branch instructions, Branch misses, Level 2 cache references, Level 2 cache misses, Level 1 instruction cache load misses, Level 1 data cache load misses, Instructions retired, and Cycles.

The performance results are broken down into two parts. The first part examines the effect of branch linking and IHL against the baseline. The benchmarks listed in Table I have been selected for this first part and represent a wide range of different workload types following the indirect branch analysis performed by d'Antras *et al.* [19].

The second part uses the full benchmark suite and examines the effect of the remaining optimizations when branch linking and IHL are already applied, which is used as a baseline for comparison with jump trampolines, traces, contiguous traces and TRIBI. The reason for this division relates to the time needed to run the benchmarks. The performance of MAMBO is much improved with branch linking and IHL optimizations applied making it feasible to consider all the benchmarks.

B. Benchmark Results

1) Branch Linking and IHL: Figure 6 displays the relative overhead of the baseline compared to native execution, which represents the starting performance of MAMBO as described

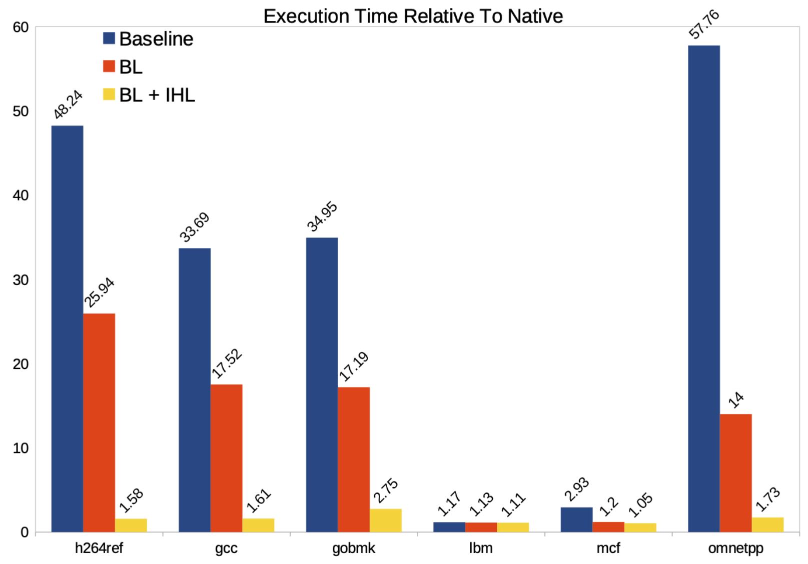


Fig. 6: Effect of branch optimizations.

TABLE I: Benchmarks used for the baseline, branch linking and IHL comparison.

Benchmark	Reason
h264ref	Medium number of indirect branches.
lbm	Small number of indirect branches.
omnetpp	Large number function returns and indirect branches.
gcc	Large number of function returns and table branches.
mcf	Small number of indirect branches.
gobmk	Medium number of indirect branches.

in Section II. Plotted alongside this is the performance overhead of MAMBO after first applying branch linking, and then branch linking combined with IHL. As a result of the large overheads experienced, the runtimes of some benchmarks with the baseline MAMBO are substantial. An example of this is *h264ref*, which takes over 75 hours per run to complete.

Both optimization approaches result in a speedup but to varying degrees. Branch linking targets direct branches and generally has quite a significant effect. In the case of *lbm* and *mcf*, this effect is much smaller, which is due to both benchmarks having a small proportion of branches [19]. Inline Hash Lookup also provides a good performance improvement, with *mcf* and *lbm* showing the least improvement due to their small number of indirect branches. Both of these optimizations reduce the number of context switches needed when handling both direct and indirect branches.

TABLE II: SPEC CPU 2006 overhead relative to native.

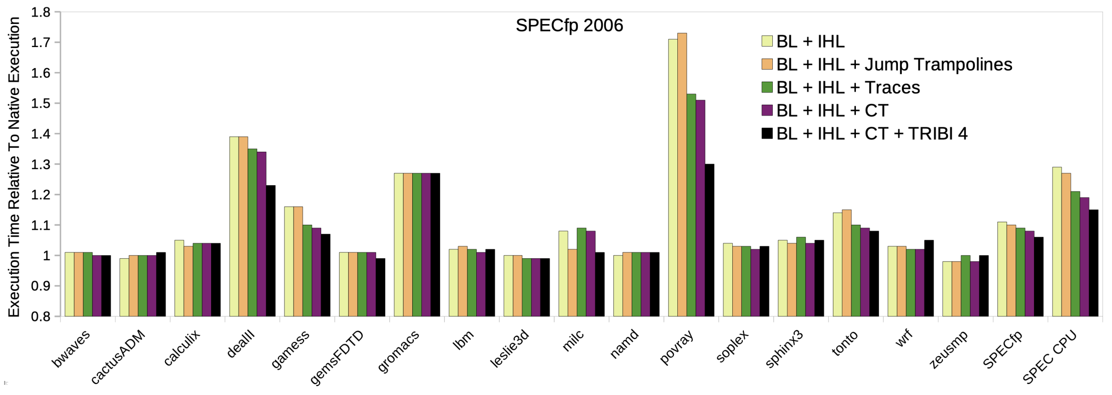
Configuration	SPECint	SPECfp	All
BL+IHL	+59.6%	+10.6%	+28.8%
BL+IHL+Jump Trampolines	+56.6%	+9.8%	+27.2%
BL+IHL+Traces	+39.4%	+9.3%	+21.1%
BL+IHL+Contiguous Traces	+35.0%	+8.0%	+18.5%
BL+IHL+Contiguous Traces+TRIBI(4)	+28.8%	+6.3%	+15.1%
Best Configuration For Each	+28.5%	+5.6%	+14.5%

2) All other optimizations: Table II shows the geomean overheads for SPECint, SPECfp and all benchmarks run, they help to paint a general picture of how effective an optimization is when considering a large range of workloads. Individual benchmark results can be seen in Figure 7. Table III shows the baseline overhead against the overhead achieved with branch linking and IHL enabled, and the overhead achieved when branch linking, IHL, traces, contiguous traces and TRIBI 4 are enabled, for the benchmarks used for branch optimization benchmarking.

TABLE III: Overhead relative to native execution of baseline compared to BL+IHL and TRIBI 4.

	Baseline	BL+IHL	TRIBI 4
h264ref	+4724%	+57.4%	+43.8%
gcc	+3269%	+60.9%	+33.6%
gobmk	+3395%	+175%	+86.3%
lbm	+17.0%	+11.8%	+1.9%
mcf	+193%	+4.7%	+0.5%
omnetpp	+5667%	+73.2%	+23.9%

From a starting overhead of 10.6%, SPECfp benchmarks exhibit the lowest geomean overhead reduction of just 5% compared to running natively. However, this small reduction represents 47.2% of the starting overhead. SPECint benchmarks demonstrate a greater reduction in overhead of 31.1% with the best optimizations applied to each benchmark, compared to native execution, representing 51.2% of the starting overhead, which is in part due to the fact that SPECint benchmarks have higher overheads in general, which may be attributed to the fact that they tend to have more branches, which represent a large proportion of overhead. The geomean overhead of all of the SPEC benchmarks run, falls from 28.8%



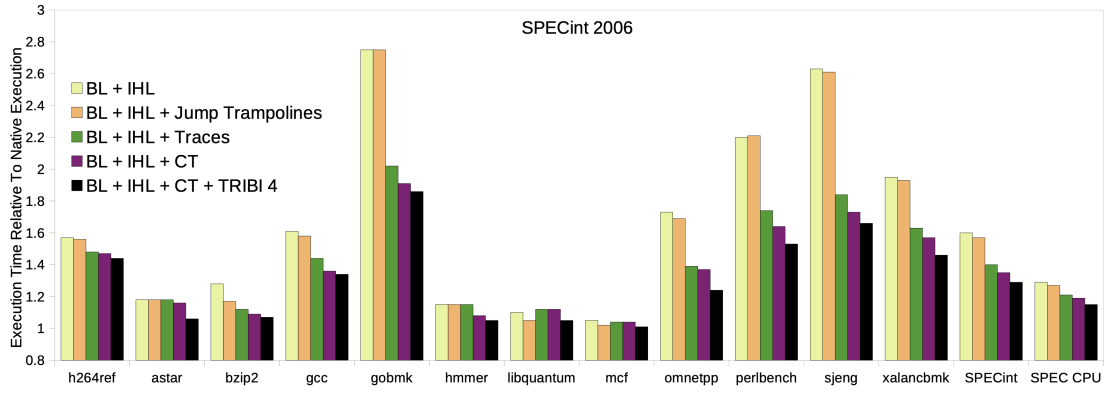


Fig. 7: Execution times of SPEC CPU2006 compared to native execution.

with IHL applied, to 15.1% with TRIBI and 14.5% with the most effective optimizations applied, a 14.3% reduction in runtime compared to native execution times and 49.7% of the overhead at the start.

The reduction in performance overhead compares favourably to the overhead achieved by MAMBO on 32-bit ARM architectures — 12%-21% [10], and on 64-bit ARM architectures — 8%-11% [11].

Results show that although TRIBI 4 provides the best overall reduction of overhead as a blanket policy, the overhead achieved when the best configuration for each benchmark is used, performs better overall. This highlights the importance of tuning and analysing the effect of optimizations applied to each workload individually when using MAMBO with a new and intensive application.

C. Analysis using Performance Counters

Performance counter data has been gathered for the following configurations: IHL, jump trampolines, traces, contiguous traces and TRIBI. Performance counter data for IHL is provided for reference in order to observe the effect of optimizations compared to a more performant and useful point than the baseline. Table IV shows the geometric mean average performance counter change compared to native execution, which will help to visualise what the optimizations are physically affecting. Generally, traces, contiguous traces and TRIBI reduce branch, L1 instruction cache and L2 cache misses, which are the driving force behind the reduced overhead observed. IPC, a measure of overall system performance, which it is desirable to maximise, rises steadily with each trace optimization applied, meaning that the processor is spending less time idle or waiting, which may be explained by the reduced branch and cache misses.

1) Jump Trampolines: Jump trampolines result in an average 1.6% speedup compared to IHL. Jump trampolines have been evaluated without trace optimizations; traces will negate much of the effect observed, as basic blocks will only execute 256 times before execution switches to the trace cache, where jump trampolines have not been implemented. In most cases some benefit is observed, with 14 benchmarks showing reduced overhead with jump trampolines enabled. In some cases however, no effect is observed and in a small number of cases there is a slight performance degradation. One such case is *povray*, where the only significant change observed is the number of instructions retired, possibly indicating that there may still be some inefficiencies in the trampoline insertion mechanism in certain cases. The performance counter data also reveals an increase in branch mispredictions, indicating that the branch predictor has difficulty predicting chained jumps. For 10 workloads, jump trampolines compare well to the overhead achieved by using trace optimizations, for example in the case of *lbm* and *libquantum*, making it a viable alternative in some cases where traces do not provide the best solution, particularly for many SPECfp benchmarks.

TABLE IV: Geometric mean average performance counter values relative to native.

	IHL	Jump Trampolines	Traces	Contiguous Traces	TRIBI
branches	+30.0%	+30.0%	+30.0%	+30.0%	+25.6%
branch-misses	+63.5%	+66.0%	+57.1%	+48.3%	+36.4%
Cache-references (L2)	+87.4%	+79.1%	+60.4%	+51.9%	+46.2%
Cache-misses (L2)	+100.0%	+94.0%	+32.5%	+37.8%	+35.3%
L1-icache-load-misses	+2890.0%	+2866.4%	+661.9%	+609.4%	+618.5%
L1-dcache-load-misses	+16.1%	+50.5%	+46.9%	+46.1%	+36.0%
instructions	+64.4%	+71.7%	+59.2%	+62.5%	+49.7%
IPC	-15.0%	+50.8%	+1.5%	+5.9%	+12.1%

2) Traces: Traces optimize the usage of the processor instruction cache by grouping instructions physically closer together. Indeed, this corresponds to significant reductions in L1 instruction and L2 cache misses which is the intended effect of the optimization. Compared to IHL, traces on average, reduce the amount of L1 instruction cache misses. For all evaluated workloads, traces provide a benefit or do not significantly affect performance which is reflected in the geomean overheads, where traces provide an 8.5% speedup when compared to the geomean overhead for IHL. The largest effect can be seen when running SPECint benchmarks, which contain a larger number of branch instructions [20], suggesting that the better cache locality helps to improve the performance of these control flow instructions when jumping around, compared to a fragmented code cache.

Certain benchmarks do not benefit from traces, including *sphinx3* and *libquantum*. However, even in these cases, the performance is degraded by less than 1.5% which is vastly outweighed by the benefits brought about by traces.

3) Contiguous Traces: Contiguous traces improve upon standard traces by reducing the number of instructions in the hot code path, reducing the strain on the instruction cache. The performance improvement observed in the traces optimization coincides with a lower instruction cache miss rate, with the trend continued by contiguous traces. This result is supported also by the geomean overhead, which shows that contiguous traces improve upon regular traces by 2.6%. When comparing *dealII* and *gobmk*, it can be seen that *gobmk* benefits more from contiguous traces than *dealII* and this is reflected in a further reduction in the L1 instruction cache miss rate. *gobmk* has a trace cache which is twice the size of that of *dealII*, which means that a reduction in the hot code path size will make a larger difference to a larger trace cache, potentially enabling more useful instruction cache entries.

4) TRIBI: The introduction of TRIBI provides a larger performance improvement than the introduction of contiguous traces, with the number of retired instructions, L2 cache accesses, L2 cache misses, retired branches, branch misses and L1 data cache misses falling. The fall in the number of instructions can be attributed to the reduction in the number of instructions needed to handle indirect branches, which TRIBI enables. The ability to bypass hash table data accesses is likely to be a driving force behind the overall reduction in L2 cache accesses and misses as well as L1 data cache misses. The transformation of some indirect branches, which require lookups at run-time, to direct branches, will help to ease strain on the less accurate indirect branch predictor and onto the branch target buffer. This has a positive effect on branch misses. As a result of fewer cache accesses and misses meaning fewer cycles waiting, the IPC, an overall indicator of performance, rises.

However, the number of L1 instruction cache misses rises overall compared to contiguous traces but is overall still lower than traces. This rise can be attributed to the increased size of the trace cache because of the addition of predictions, meaning that fewer instructions from the hot code path can be cached. As an example, the trace cache size of *omnetpp* increases by 39% with TRIBI 4 enabled, compared to contiguous traces.

Although an overall speedup is observed, benchmarks such as *wrf* and *zeusmp* experience a slowdown compared to contiguous traces, likely due to a larger proportion of indirect branches with more than 4 targets. The result being wasted effort setting up TRIBI and inserting predictions, before ultimately disabling it.

VI. Related Work

Hazelwood [21] and Wenzi *et al.* [22] provide background material and surveys for DBM and binary rewriting. Binary modification and instrumentation tools are relatively new to the RISC-V architecture, with MAMBO being the first optimized DBM available. *Instrew* [23] is a tool which enables binary translation and instrumentation by elevating RISC-V source code to LLVM-IR and utilizing the LLVM toolchain to perform translation or modification of a binary. This approach differs from MAMBO since modifications are not performed at runtime. Although little work has been done on binary modification for RISC-V, more work has been done on binary translation. Tools such as *rv8* [24] and QEMU [6] provide emulation of the RISC-V architecture. However, these do not provide the performance benefits of low overhead native execution and this can be prohibitive in more complex applications. The PatchWrx framework developed for the DEC Alpha architecture used a similar optimization to the described jump trampolines, but relying on privileged PAL code [25].

VII. Conclusions

DBM tools provide the fundamental functionality needed for binary analysis. Six DBM performance optimizations on RISC-V have been described and evaluated. The optimizations have been developed and adapted to address challenges specific to the RISC-V architecture. The short range of `JAL` instructions necessitates the use of `JALR` instructions to jump to targets outside of the ±1MB range allowed. *Jump trampolines* address this issue, replacing `JALR` instructions with a series of `JAL` instructions. These result in an overall speedup, especially under SPECfp workloads where the performance is comparable to the performance improvement achieved by trace optimizations. In addition, we have adapted contiguous traces to overcome the short conditional branch range specified by RISC-V. We have restricted contiguous traces to a the maximum size per trace of 4KB, and split each trace into 3KB for its main part and 1KB for trace exits.

The overall result of the implemented optimizations is a reduced geometric mean overhead to 14.5%, with SPECint having 28.5% and SPECfp 5.6%. The implementation of these optimizations results in large reductions in overhead for individual benchmarks, including *h264ref*: a runtime of 2.2 hours, down from over 75 (native runtime is 1.6 hours), and *omnetpp*: a runtime of 1 hour, down from over 48 hours in the baseline configuration (native runtime is 0.8 hours). The overall result compares well to the overhead achieved on existing 32-bit (12%-21%) and 64-bit (8%-11%) ARM systems [10] [11], as well as on Intel/AMD, e.g. Pin (21%) [26]. Thus, MAMBO becomes the first optimized open source DBM tool for the RISC-V ecosystem.

Acknowledgments

This research was partially supported by the UK Industrial Strategy Challenge Fund (ISCF) under the Digital Security by Design (DSbD) Programme delivered by UKRI as part of the MoatE (10017512) and Soteria (75243) projects, and by EPSRC Rain Hub (EP/R026084/1), and LAMBDA (EP/N035127/1) projects. Mikel Luján is supported by a Royal Society Wolfson Fellowship and an Arm/RAEng Research Chair Award. SiFive donated one SiFive Unmatched board used in the experiments.

References

[1] K. M. Hazelwood and A. Klauser, “A dynamic binary instrumentation engine for the ARM architecture,” in *Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2006*. ACM, 2006, pp. 261–270. [Online]. Available: https://doi.org/10.1145/1176760.1176793

[2] C. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. ACM, 2005, pp. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[3] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation,*. ACM, 2007, pp. 89–100. [Online]. Available: https://doi.org/10.1145/1250734.1250746

[4] D. Bruening and Q. Zhao, “Practical memory checking with Dr. Memory,” in *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization*. IEEE, 2011, pp. 213–223. [Online]. Available: https://doi.org/10.1109/CGO.2011.5764689

[5] J. E. Smith and R. Nair, *Virtual Machines – Versatile platforms for systems and processes*. Elsevier, 2005.

[6] F. Bellard, “QEMU, a fast and portable dynamic translator,” in *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*. USENIX, 2005, pp. 41–46. [Online]. Available: http://www.usenix.org/events/usenix05/tech/freenix/bellard.html

[7] A. d’Antras, C. Gorgovan, J. D. Garside, and M. Luján, “Low overhead dynamic binary translation on ARM,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*. ACM, 2017, pp. 333–346. [Online]. Available: https://doi.org/10.1145/3062341.3062371

[8] C. Gorgovan, A. d’Antras, and M. Luján, “Mambo: A low-overhead dynamic binary modification tool for arm,” *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, 2016. [Online]. Available: https://doi.org/10.1145/2896451

[9] C. Gorgovan, G. Callaghan, and M. Luján, “Balancing performance and productivity for the development of dynamic binary instrumentation tools: A case study on Arm systems,” in *Proceedings of the 29th International Conference on Compiler Construction*, ser. CC 2020. ACM, 2020, p. 132–142. [Online]. Available: https://doi.org/10.1145/3377555.3377895

[10] C. Gorgovan, A. d’Antras, and M. Luján, “Optimising dynamic binary modification across ARM microarchitectures,” in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018*, ser. ICPE ’18. ACM, 2018, pp. 28–39. [Online]. Available: https://doi.org/10.1145/3184407.3184425

[11] G. Callaghan, C. Gorgovan, and M. Luján, “Optimising dynamic binary modification across 64-bit arm microarchitectures,” in *VEE’20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE ’20. ACM, 2020, pp. 185–197. [Online]. Available: https://doi.org/10.1145/3381052.3381322

[12] A. Waterman and K. Asanović, “The risc-v instruction set manual, volume i: User-level isa, document version 2019121,” 2019. [Online]. Available: https://riscv.org/specifications

[13] D. Bruening, “Efficient, transparent, and comprehensive runtime code manipulation,” Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[14] SiFive Inc., “Sifive fu740-c000 manual v1p6,” 2022. [Online]. Available: https://www.sifive.com/documentation

[15] E. Duesterwald and V. Bala, “Software profiling for hot path prediction: Less is more,” in *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 2000, pp. 202–211. [Online]. Available: https://doi.org/10.1145/378993.379241

[16] ARM Ltd., “ARM architecture reference manual for A-profile architecture,” 2013.

[17] SiFive Inc., “HF105 Datasheet.” [Online]. Available: https://www.sifive.com/boards/hifive-unmatched

[18] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *SIGARCH Comput. Archit. News*, vol. 34, no. 4, p. 1–17, sep 2006. [Online]. Available: https://doi.org/10.1145/1186736.1186737

[19] A. d’Antras, C. Gorgovan, J. D. Garside, and M. Luján, “Optimizing indirect branches in dynamic binary translators,” *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, pp. 7:1–7:25, 2016. [Online]. Available: https://doi.org/10.1145/2866573

[20] S. Bird, A. Phansalkar, L. K. John, A. E. Mericas, and R. Indukuru, “Performance characterization of SPEC CPU benchmarks on Intel’s Core microarchitecture based processor,” in *SPEC Benchmark Workshop*.

[21] K. Hazelwood, *Dynamic Binary Modification: Tools, Techniques, and Applications*. Morgan Claypool Publishers, 2011.

[22] M. Wenzl, G. Merzdovnik, J. Ullrich, and E. Weippl, “From hack to elaborate technique—a survey on binary rewriting,” *ACM Comput. Surv.*, vol. 52, no. 3, 2019. [Online]. Available: https://doi.org/10.1145/3316415

[23] A. Engelke, “Optimizing performance using dynamic code generation,” Ph.D. dissertation, Technical University of Munich, Germany, 2021.

[24] M. Clark and B. Hoult, “rv8: a high performance risc-v to x86 binary translator,” in *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[25] J. P. Casmira, D. P. Hunter, and D. R. Kaeli, “Tracing and characterization of Windows NT-based system workloads,” *Digit. Tech. J.*, vol. 10, no. 1, pp. 6–21, 1998. [Online]. Available: https://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art1.pdf

[26] D. Bruening, Q. Zhao, and S. P. Amarasinghe, “Transparent dynamic instrumentation,” in *Proceedings of the 8th International Conference on Virtual Execution Environments, VEE 2012*. ACM, 2012, pp. 133–144. [Online]. Available: https://doi.org/10.1145/2151024.2151043