Optimizing Indirect Branches in Dynamic Binary Translators

AMANIEU D'ANTRAS, COSMIN GORGOVAN, JIM GARSIDE, and MIKEL LUJÁN,
School of Computer Science, University of Manchester

Dynamic binary translation is a technology for transparently translating and modifying a program at the machine code level as it is running. A significant factor in the performance of a dynamic binary translator is its handling of indirect branches. Unlike direct branches, which have a known target at translation time, an indirect branch requires translating a source program counter address to a translated program counter address every time the branch is executed. This translation can impose a serious runtime penalty if it is not handled efficiently.

MAMBO-X64, a dynamic binary translator that translates 32-bit ARM (AArch32) code to 64-bit ARM (AArch64) code, uses three novel techniques to improve the performance of indirect branch translation. Together, these techniques allow MAMBO-X64 to achieve a very low performance overhead of only 10% on average compared to native execution of 32-bit programs. *Hardware-assisted function returns* use a software return address stack to predict the targets of function returns, making use of several novel optimizations while also exploiting hardware return address prediction. This technique has a significant impact on most benchmarks, reducing binary translation overhead compared to native execution by 40% on average and by 90% on some benchmarks. *Branch table inference*, an algorithm for detecting and translating branch tables, can reduce the overhead of translated code by up to 40% on some SPEC CPU2006 benchmarks. The remaining indirect branches are handled using a *fast atomic hash table*, which is optimized to work with multiple threads. This last technique translates indirect branches using a single shared hash table while avoiding expensive synchronization in performance-critical lookup code. This allows the performance to be on par with thread-private hash tables while having superior memory scalability.

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

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

ACM Reference Format:
Amanieu d'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. 2016. Optimizing indirect branches in dynamic binary translators. ACM Trans. Archit. Code Optim. 13, 1, Article 7 (April 2016), 25 pages.
DOI: http://dx.doi.org/10.1145/2866573

1. INTRODUCTION

Binary translation is a technology that allows a program to be transparently translated and modified at the machine code level. It has numerous applications, such as dynamic instrumentation [Moseley et al. 2007; Seward and Nethercote 2005], program analysis [Sato et al. 2011; Zhao et al. 2011], virtualization [Adams and Agesen 2006; Watson 2008], and instruction set translation [Bellard 2005; Dehnert et al. 2003]. A binary translator does not need access to the source code of a program, which makes it

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

Authors' addresses: A. d'Antras, C. Gorgovan, J. Garside, and M. Luján, The APT Group, School of Computer Science, The University of Manchester, Oxford Road, Manchester, M13 9PL, United Kingdom; emails: {bdantras, cgorgovan, jgarside, mikel}@cs.man.ac.uk.

1544-3566/2016/04-ART7 $15.00
DOI: http://dx.doi.org/10.1145/2866573

particularly useful in cases where source code is not available or is not portable enough to be simply recompiled.

Binary translation can be either static or dynamic. A static binary translator translates the entirety of the program object code ahead of time. This is not always practical in the general case due to the *code-discovery problem* [Horspool and Marovac 1980]: since it is not always possible to determine which memory locations contain code, all addresses need to be treated as potential branch targets to ensure full transparency. Additionally, in modern systems, not all of the code that will be executed is known ahead of time, such as when a program uses shared libraries or generates new instructions using a Just-In-Time compiler. A Dynamic Binary Translator (DBT) translates code only as it is about to be executed, which avoids these issues but comes at a cost in execution time.

Rather than translating instructions individually, a DBT usually translates instructions in blocks. Since some code, such as loop and function bodies, are likely to be executed many times, it is advantageous to keep the translated blocks so that they can be used again, rather than retranslating them each time they are encountered. Rather than modifying the program code, translated code fragments are stored in a *code cache* separate from the original instructions.

When executing in a code cache, control sometimes needs to be transferred between blocks. A simple solution is to give control back to the DBT so that it can find the next block to execute, but this adds significant overhead because branches occur frequently in programs. Instead, a branch in one block can be *linked* to a different block by modifying the branch to point to the latter block. This technique only works for *direct branches*, where the branch target is known at translation time.

Indirect branches need to be handled differently because the branch target is only known at execution time and can vary from one execution to the next. This requires its own dynamic translation and can impose a serious runtime penalty. Previous research [Kim and Smith 2003; Hiser et al. 2007] has shown that indirect branch handling is the biggest performance bottleneck in a DBT.

Figure 1 shows how indirect branches are handled in a typical DBT: every time an indirect branch is executed in a DBT, a Source Program Counter (SPC) value must be mapped to a Translated Program Counter (TPC) value, which is then branched to. While this approach solves the translation problem, it is still many times slower than a native indirect branch, which only consists of a single instruction. It also interacts poorly with hardware branch prediction mechanisms that are optimized for native code, often resulting in unnecessary branch mispredictions.

In a typical program, indirect branches mainly come from three sources, as shown in Figure 2:

Branch tables: Branch tables are an efficient way to branch to many targets by using an array of code addresses in memory.

Function returns: Because a function may be called from many places, functions must use an indirect branch to return to their caller.

Function pointers: Function pointers and virtual functions are used to dispatch execution to different functions dynamically.

When generated by a compiler, each of these classes has distinctive assembly code signatures that a DBT can detect. This allows the DBT to perform specialized optimizations depending on the branch type. This article presents three indirect branch handling techniques that handle each type of indirect branch efficiently.

The first, *hardware-assisted function returns* (Section 2), uses a stack of translated return addresses to predict the target of function returns, thus avoiding the need for a hash table lookup. While return address stacks have previously been used in

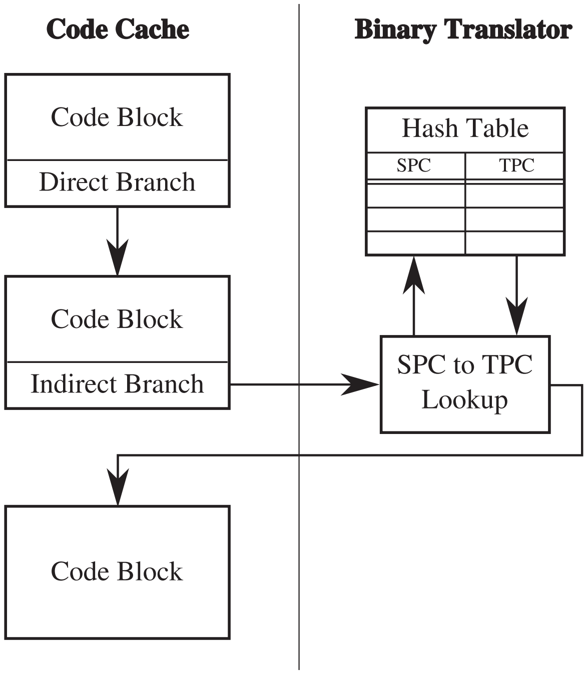


Fig. 1. Direct and indirect branch handling in a DBT.

Function call and return

```
caller:
...
BL callee
...

callee:
...
BX LR
```

Branch table

```
CMP R0, #3
BHI default
ADR R1, table
LDR PC, [R1, R0, LSL #2]
table:
.word case0
.word case1
.word case2
.word case3
```

Function pointer call

```
caller:
...
LDR R0, [R1]
BLX R0
...

callee:
...
BX LR
```

Fig. 2. Indirect branch types generated by GCC when compiling for 32-bit ARM.

some DBTs [Hazelwood and Klauser 2006] to predict function returns, they have not always resulted in performance improvements due to the increased number of memory operations and poor interactions with indirect branch prediction hardware. Hardware-assisted function returns are designed to work with the return address predictor of the target processor, while also including optimizations to eliminate many return address stack operations and handle return address mispredictions efficiently.

The second, *branch table inference* (Section 3), is a pattern-matching algorithm to detect branch tables during translation and generate a corresponding table in the code cache. Although ad hoc branch table detection has been explored in DBTs [Payer and Gross 2010], the proposed inference provides a systematic way of detecting many variants of branch tables and extracting the bounds of the table directly from the source instructions instead of guessing it.

A hash table is still necessary to handle the remaining indirect branches that are not covered by the previously mentioned techniques. However, most existing hash tables used for indirect branch translation are not designed to work with multiple threads and require either duplicating the hash table for each thread or introducing expensive synchronization mechanisms. This article presents *fast atomic hash tables* (Section 4), which take advantage of cheap 64-bit atomic loads and stores to provide a thread-shared hash table that matches the performance of single-threaded hash tables.

These techniques were developed on MAMBO-X64, a DBT that translates 32-bit ARM programs into AArch64 code, which is the new 64-bit instruction set introduced in the ARMv8 version of the ARM architecture. While the current generation of ARMv8 processors is capable of running 32-bit ARM code directly, maintaining this support comes at a cost in silicon space and significantly increases the complexity of hardware verification. MAMBO-X64 has a low average performance overhead of only 10%, which allows future processors to preserve the ability to transparently run 32-bit ARM programs while only supporting the AArch64 instruction set in hardware.

In Section 5, these techniques are evaluated on an ARM Cortex-A57 system using the SPEC CPU2006 benchmark suite. The results show that the hardware-assisted function return optimization has the highest impact on performance, with an average overhead reduction of 40% and up to 90% on some benchmarks compared to hash table lookups. Branch table inference has a significant effect on benchmarks that make frequent use of branch tables, reducing DBT overhead by up to 40% in those benchmarks. Finally, fast atomic hash tables are shown to reduce DBT overhead by 40% compared to existing thread-shared hash table designs, also matching the performance of other indirect branch handling techniques while consuming significantly less memory.

2. HARDWARE-ASSISTED FUNCTION RETURNS

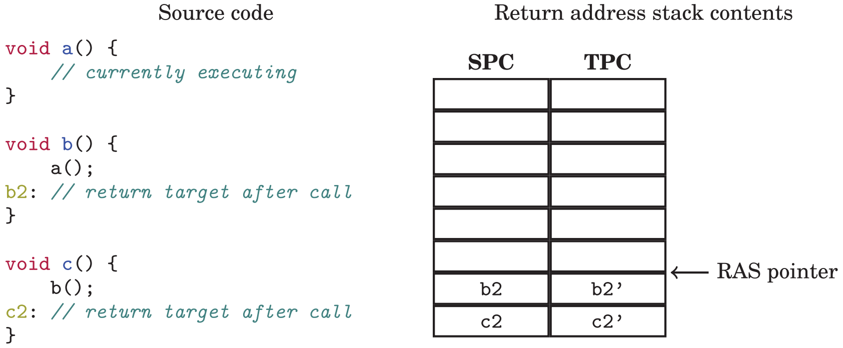
Research has shown that function returns are by far the most common type of indirect branch [Scott et al. 2004]. Function returns are different from other indirect branches in that they usually target the next instruction after a previously executed call instruction. In some cases, a function may not return to the address it was called from. This is atypical, only occurring in exceptional cases such as during stack unwinding after an exception is thrown or if the return address of a function has been modified.

Hardware-assisted function returns take advantage of this property in two ways, first by tracking the addresses of executed call instructions in a software return address stack, and second by laying out the translated code in a way that can take advantage of hardware return address prediction logic built into modern processors. While software return address stacks have previously been used in DBTs to optimize function returns [Hazelwood and Klauser 2006; Hookway and Herdeg 1997], this technique extends them by efficiently handling stack overflows and underflows using memory protection hardware, and by avoiding the return address stack entirely in certain cases (e.g., leaf functions).

2.1. Software Return Address Stack

Hardware-assisted function returns work by maintaining a *Return Address Stack* (RAS) in memory, which tracks previously executed call instructions. Each thread is allocated its own RAS, and the current position in the stack is tracked by a dedicated RAS pointer register. To account for the possibility of mispredicting returns, each RAS entry comprises a pair of values: the SPC of the expected return address and its corresponding TPC. An entry is pushed onto the RAS by a translated call instruction and an entry is popped from the RAS by a translated return instruction. The resulting RAS entries therefore mirror the call stack of the program, as shown in Figure 3.

Translating a call instruction is simple since all it needs to do is push the SPC and TPC of the call return target, which is usually the instruction immediately after the call instruction. Translating a return instruction is more complicated due to the need to handle potential mispredictions and requires four operations:



Source code

```
void a() {
    // currently executing
}

void b() {
    a();
b2: // return target after call
}

void c() {
    b();
c2: // return target after call
}
```

Return address stack contents

SPC	TPC
b2	b2’
c2	c2’

Fig. 3. Return address stack contents while executing nested function calls.

(1) An entry containing an SPC and TPC pair is popped from the return address stack.
(2) The SPC is compared with the program-visible return address used by the return instruction.
(3) If the values match, then control branches to the TPC in the entry.
(4) If the values do not match, then control is returned to the DBT so that it can determine the target of the return.

2.2. Hardware Return Address Prediction

Most modern processors include a return address prediction mechanism in hardware to predict the targets of function returns. This specifically detects “call” and “return” instructions and passes them to the branch prediction system. Unfortunately, this is not used in most DBTs because code before and after the call is treated separately for translation purposes, so they may not place the target of a return immediately after the matching call instruction in the code cache, which is required to exploit the hardware predictor.

Hardware-assisted function returns exploit hardware return prediction by including the return target in the same block as the call instruction, thus ensuring that the return target is located immediately after the translated call instruction. This is effectively done by not ending a basic block when a call instruction is encountered. Translated code can then use native call and return instructions, which take advantage of any return address predictor.

Figure 4 shows how a function call and return are translated in MAMBO-X64. The source BL instruction is translated into a constant move to set the source link register and a BL (call) instruction to branch to the translated function. The code takes advantage of the BL instruction to generate the TPC address in the link register. In the translated function, the source link register and translated link register are both saved to the return address stack. The BX LR (return) instruction is translated into a return address stack pop and compare. If the comparison succeeds, then a RET instruction is used to branch to the translated address from the stack. The RET instruction allows the processor to use its return address predictor for this branch. Because the return address stack contains the link register value generated by the BL instruction, the processor will predict the target of the return correctly.

2.3. Return Address Stack Elision

Rather than pushing an entry to the RAS at the translated call instruction in the caller block, the SPC and TPC of the return target are passed to the callee block in registers. The SPC is passed in the application-visible link register for the target architecture

Original 32-bit ARM code

```
orig_caller:
BL function      ; branch and set LR
orig_ret_target:
...              ; rest of code

orig_function:
...              ; contents of function
BX LR            ; return using LR (R14)
```

Translated AArch64 code

```
translated_caller:
MOV W14, #orig_ret_target      ; calculate return SPC
BL translated_function         ; branch and set TPC
translated_ret_target:
...                            ; rest of code

translated_function:
STP X14, LR, [ras_ptr], #16    ; push SPC and TPC
...                            ; contents of function
...                            ; possibly spread over
...                            ; multiple blocks
LDP X16, LR, [ras_ptr, #-16]!  ; pop SPC and TPC
SUB W16, W16, W14              ; compare SPC with LR
CBNZ W16, return_mispredict    ; handle mispredicts
RET LR                         ; return using TPC
```

Fig. 4. Translated function call and return in MAMBO-X64. `ras_ptr` is a register that holds the return address stack pointer, `X16` is a scratch register, and `W14` contains the ARM link register. Note that on AArch64, the link register is `X30`, whereas on ARM it is `R14`.

(`R14` on ARM), while the TPC is passed in the link register for the host architecture (`X30` on AArch64). This TPC value is hidden from the target application and remains valid as long as the application link register is not modified.

The relationship between these two values is broken when the application link register is modified, either explicitly by overwriting the link register with a different value or implicitly through a call instruction that overwrites this register. In this situation, the SPC and TPC pair needs to be pushed to the RAS just before the register holding the SPC is modified. When a function returns by performing an indirect branch to the address in the target link register, a RAS pop can be avoided if the link register is known to not have been modified since the last executed call instruction. In this situation, the host link register already contains the correct TPC address to return to and can be branched to directly.

The relationship between the host and target link registers is maintained across block boundaries by creating two variants of every block: a *normal* variant and a *callee* variant. The latter variant has the property that, on entry, both registers will contain valid values. Each variant is generated on demand since in practice, most blocks only ever use a single variant: for the SPEC2006 benchmarks, on average only 1.5% of all blocks needed to have both a normal and callee variant generated.

Three rules determine which variant of a block is targeted when a branch instruction is translated:

(1) If the instruction is a call branch, then it will always target a callee variant.
(2) If the instruction is a noncall branch, the block containing the branch is a callee variant, and the target link register has not been modified since the start of the block, then the branch will target a callee variant.
(3) In all other cases, the branch will target a normal variant.

For indirect branches, two separate hash tables are used, one for normal variants and one for callee variants. Although the target address of an indirect branch can't be determined at translation time, the target variant can be determined because indirect calls (`BLX` on ARM) can be distinguished from other indirect branch types. The indirect branch is then translated to use one of the two hash tables depending on the variant that needs to be targeted.

While this optimization aims to avoid RAS overhead for leaf functions, these rules also allow RAS elision to be applied to nonleaf functions for code paths that return

Original ARM code

```
non_leaf:
CBZ R0, non_leaf_ret

STR LR, [SP, #-4]!
BL leaf
LDR LR, [SP], #4
B non_leaf_ret

non_leaf_ret:
BX LR

leaf:
BX LR
```

Translated AArch64 code

```
non_leaf-callee:
CBZ W0, non_leaf_ret-callee      (*)

STR W14, [X13, #-4]!             (*)
STP X14, LR, [ras_ptr], #16
MOV W14, #return_target
BL leaf-callee
LDR W14, [X13], #4
B non_leaf_ret

non_leaf_ret:
LDP X16, LR, [ras_ptr, #-16]!
SUB W16, W16, W14
CBNZ W16, return_mispredict
RET LR

non_leaf_ret-callee:
RET LR                           (*)

leaf-callee:
RET LR                           (*)
```

Fig. 5. Example code showing RAS elision in MAMBO-X64. For instructions marked with (*), W14 contains the return target SPC and LR contains the return target TPC. This property is preserved across branches by making them target callee blocks instead of normal blocks, but is lost when the ARM link register (W14) is modified, such as by the BL instruction, at which point the SPC and TPC values must be saved to the RAS.

without calling any other functions. Figure 5 shows an example of this optimization being applied: when a return instruction is in a callee block and the target register (W14) has not been modified, it can be translated to a single host return instruction that branches to the TPC return address in the host link register (LR). The resulting code therefore only requires a single instruction for returns, effectively matching the performance of a native function return.

2.4. Overflow and Underflow Handling

Because the RAS is allocated as a block of memory of fixed size, it can *overflow* it if a function call is executed when the stack is full. Similarly, it is possible to *underflow* the RAS by attempting to return from a function when the RAS is empty. The former usually occurs when searching through a deep tree structure recursively, while the latter usually occurs when returning from such a recursion. A DBT must handle both of these situations to maintain transparency since they could otherwise potentially result in incorrect code execution.

Many RAS implementations handle overflows and underflows by adding bound-checking instructions to the RAS push and pop operations, but this comes at a significant cost due to the additional instructions required and the high frequency of calls and returns in many programs. A better approach is to use memory protection hardware to trap overflows by allocating a *guard page* at the end of the RAS. Underflows are caught using a *guard entry* that is reserved at the bottom of the stack. Figure 6 shows an example of how RAS overflows and underflows are handled.

When the RAS is full and a push is attempted, the write to the guard page will trigger a page fault. The fault handler will shift the RAS contents down: the top half of the stack is copied to the bottom half and the RAS pointer register is adjusted to point to the new top of the stack. Although this discards the bottom half of the RAS entries, which are the least recently used, correct execution is not affected because the RAS is only used as a prediction mechanism. After the stack contents are moved,

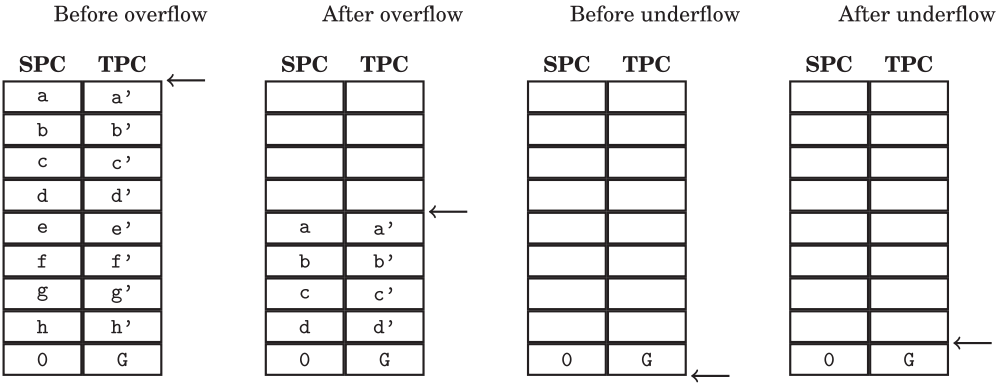


Fig. 6. Overflow and underflow handling for the return address stack. On overflow, the contents of the RAS are shifted down, the RAS pointer is adjusted, and the push instruction is restarted. On underflow, the RAS pointer is moved above the guard entry so that the guard is not overwritten by a later push, and the misprediction is handled by the DBT.

the push instruction is restarted with the adjusted RAS pointer, which will cause it to successfully push a value into the newly freed space.

To catch underflows, a guard entry is reserved at the bottom of the RAS. This entry contains an SPC address of 0 and a TPC address pointing to a stub that returns to the DBT, so that control returns to the DBT even if the return target happens to be address 0. Once control is returned to the DBT, the current RAS pointer is checked and adjusted to ensure that it always points above the guard entry. A misprediction is unavoidable at this point because there is no prediction information available in the RAS.

2.5. Misprediction Handling

There are two reasons that a function return can be mispredicted: either the function did not return to its matching call instruction or the predicted return address was lost due to a RAS overflow and subsequent underflow. These situations can be discerned by checking whether the entry that was just popped from the RAS is the guard entry. Since these situations have different causes, they are best handled separately.

Return Misprediction. A genuine return misprediction can occur for a variety of reasons, such as stack unwinding when an exception is thrown, calling the C `longjmp` function, switching stacks when invoking a coroutine, or simply having a function modify its return address. In many of these cases, program execution will continue normally at an earlier point in the call stack, so it is beneficial to avoid further mispredictions by *unwinding* the RAS to an earlier point. Unwinding is done by scanning the RAS from the top down until an entry matching the current SPC return target is found, and adjusting the RAS pointer to remove that entry and all others above it from the RAS. If a matching entry cannot be found, then the RAS is left unmodified so that its contents are still available for a later attempt at unwinding.

RAS Underflow. A misprediction due to a RAS underflow can occur when the call depth of a program exceeds the size of the RAS, causing the RAS to overflow and lose some entries. This is more common than genuine mispredictions and can happen in algorithms that make heavy use of recursion, such as when searching through a very deep tree structure. In this situation, it is possible to take advantage of the fact that these algorithms tend to only call a limited set of functions recursively: a small hash table is used to predict these function returns, which contains SPC and TPC addresses of returns that were previously mispredicted due to a RAS underflow. The hash table allows function returns to be predicted even when the relevant return address stack entry has been lost due to an overflow. This fallback avoids the need to perform an expensive context switch back to the DBT, which can cost hundreds of cycles.

Note that while a return address misprediction can affect program performance, it will never lead to incorrect code execution. This is guaranteed by always tracking the SPC address for each predicted TPC return address and checking that that SPC value matches the intended return target.

2.6. Unlinking

Hardware-assisted function returns have been designed to work in a *thread-shared code cache* model [Bruening et al. 2006; Hazelwood et al. 2009], where the same translated code is shared among multiple threads, because this model has been shown to scale significantly better than thread-private code caches. One complication with this model is that block invalidation is more complicated: a block that needs to be deleted because its source assembly instructions have been modified may still have other threads concurrently executing it. This is solved by using *lazy deletion*: all incoming links to the block are removed so that it becomes unreachable, and it is freed once all live threads have returned to the DBT at least once since the block was unlinked.

There are three ways through which a block can be reached: direct branches from one block to the next, the indirect branch lookup table, and a thread's return address stack. Direct branches can be unlinked by having each block maintain metadata about which other blocks have incoming direct branches to it. The branch instructions can then be patched to point to an *exit stub,* which returns control to the DBT while passing the current program counter value so that the DBT knows what to execute next. Removing an entry from a thread-shared indirect branch lookup table is discussed in Section 4.1.

Removing entries from the RAS of all live threads is more complicated because the RAS is a thread-private structure, which makes it impossible to safely modify from another thread. Using a lock or atomic operations to achieve this is unacceptable as it would slow down normal RAS operations, which are performance critical. Leaving the RAS unmodified is also unacceptable because it violates memory consistency: code returning into an invalidated function will expect to be executing the newly written code rather than returning into the old function.

Rather than attempting to modify the RAS of all live threads, the invalidated block can be patched so that each instruction located immediately after a call is replaced with a branch to an exit stub. This modification is safe to perform while other threads are concurrently executing code in that block because each thread will either see the new code and return to the DBT or see the old code and continue execution. Since a thread needs to execute a synchronizing instruction (`ISB` on ARM, any branch on x86) to guarantee that it will execute newly generated code, any thread that is expecting to see the new code will always execute the patched branch and return to the DBT. Once a thread has returned to the DBT, a simple scan of its RAS will remove any entries pointing to invalidated blocks. Therefore, once all threads have returned to the DBT at least once since the block was invalidated, the block can be safely freed since it is not pointed to by the RAS of any thread.

3. BRANCH TABLE INFERENCE

Branch tables are used to efficiently support multiway branches by loading a target code address from a table in memory. C compilers commonly generate branch tables for large `switch` statements. Since all targets of a branch table can be discovered at translation time, a DBT can treat it as a multiway direct branch rather than as an indirect branch. Branch table inference is an algorithm to discover (Section 3.1) and

Original C code

```
switch (val) {
    case 0: ...
    case 1: ...
    case 2: ...
    case 3: ...
    default: ...
}
```

ARM branch table

```
CMP R0, #3
BHI default
ADR R1, table
LSL R0, R0, #2
LDR PC, [R0, R1]
table:
.word case0
.word case1
.word case2
.word case3
```

AArch64 translated branch table

```
CMP X0, #3
B.HI translated_default
ADR X16, table
LDR X16, [X16, X0, LSL #3]
BR X16
table:
.quad translated_case0
.quad translated_case1
.quad translated_case2
.quad translated_case3
```

Fig. 7. ARM branch table generated by Clang/LLVM for a switch statement, and an AArch64 translation of that branch table. Note that when executed by a processor, the `LDR` instruction expands to the same micro-operations as the `LDR`/`BR` sequence, so the translation introduces negligible performance overhead.

translate (Section 3.2) branch tables. An example of this optimization is shown in Figure 7.

3.1. Detecting Branch Tables

Code implementing a branch table comprises several operations:

(1) An *index register* is compared to a constant value, which is the number of entries in the table.
(2) If the index register value is higher than the number of entries in the table, control jumps to a default case handler. Otherwise, the index register is known to be within the bounds of the table.
(3) The address of the table is loaded into a register, usually as a PC-relative constant.
(4) The address of the table entry that should be loaded is calculated by multiplying the index register by 4 (by shifting it to the left two places) and adding it to the table base.
(5) A 32-bit value is loaded from the table using the calculated address.
(6) The target address is jumped to using an indirect branch instruction.

While the specific instruction sequence used may vary depending on the target architecture, compiler, and even compiler options, it always consists of the same operations. In particular, depending on the instruction set, more than one of the listed steps may be performed by a single instruction. For example, the ARM branch table code generated by LLVM, shown in Figure 7, combines the last three operations into a single instruction.

One significant variation is when branch tables are compiled as position-independent code: in this case, rather than containing absolute target addresses, the table entries contain offsets from a known position, such as the table address or the address of the branch instruction. In such a table, the loaded value would be added to the base address before being branched to.

Branch table inference can recognize all these variations by scanning the instructions in a block in reverse order once an indirect branch instruction is encountered. During the scan, several conditions are checked:

(1) The target of the indirect branch is the result of a 32-bit load instruction, optionally added to a constant base address.
(2) The address operand of the load instruction is the sum of two values A and B.
(3) A is a constant value known at translation time. This is the base address of the table in memory.
(4) B is the value of a register R shifted to the left by two places.

ARM fixed branch table

```
CMP R0, #3
LDRLS PC, [PC, R0, LSL #2]
B default
.word case0
.word case1
.word case2
.word case3
```

ARM position-independent branch table

```
CMP R0, #3
ADDLS PC, PC, R0, LSL #2
B default
B case0
B case1
B case2
B case3
```

Fig. 8. ARM branch tables generated by GCC, which do not match generic branch table patterns. These exploit several features of the ARM instruction set: almost all instructions can be predicated, the PC register reads as a value 8 bytes past the current instruction, and writing to the PC register causes an indirect branch.

(5) Before the load, there is an exit branch if the condition code indicated a greater-than result.

(6) The condition flags used by the branch are generated by comparing R to a constant value. This constant value is the size of the table.

While this algorithm will detect most compiler-generated branch tables, such as those generated by LLVM, some compilers use nonstandard branch table structures on some architectures, which follow a different pattern. One particular example of this is the branch tables generated by GCC for ARM code, which are shown in Figure 8. The generated code makes use of several features specific to the ARM architecture to make the branch table more efficient. The branch table code generated in this case can be recognized by simply looking for certain hard-coded instruction sequences.

3.2. Translating Branch Tables

Once a branch table code sequence has been identified, the following information needs to be extracted from the sequence so that it can be translated:

—The branch table type: fixed or position independent
—The index register and its upper limit, which determines the table size
—The branch table base address
—For position-independent tables, the indirect branch base address

Once the address and size of a table are known, all possible targets can be found at translation time, so there is no need to perform an SPC-to-TPC translation every time the branch is executed. A new branch table is generated that contains the addresses of translated blocks, shadowing each target of the original branch table. Initially, the table only contains pointers to exit stubs that return control to the DBT to translate a block, but these are gradually replaced by the TPC addresses of the blocks as they are translated.

With branch table inference, a translated branch table performs exactly the same operations as a native branch table: a compare, a load, and a branch. This results in branch table inference completely eliminating any DBT overhead for this type of indirect branch. Another benefit is that branch table targets are eliminated from the set of indirect branch targets that need to be considered for generic indirect branch lookup. The latter typically uses a hash table lookup for SPC-to-TPC translation, which has poor performance when the target of the indirect branch varies a lot from one execution to the next, as is often the case with branch tables.

To ensure that the copy of the branch table in the code cache remains consistent if the branch table entries are modified, this optimization is only performed when the pages containing the branch table are mapped with read-only permissions. This covers all compiler-generated branch tables since these are located in the code or read-only data segments of the executable. Attempts to change the permissions of pages containing a

```
AND W18, table_mask, W15, LSR #3          ; hash the SPC and apply the hash table mask
ADD X18, table_base, X18                  ; get a pointer to the hash table entry

loop:
LDR X16, [X18], #8                        ; use a 64-bit atomic load to read the entry
SUB W17, W16, W15                         ; compare the low 32 bits of the entry with the SPC
CBZ W17, found                            ; break out of the loop if they match
CBNZ W16, loop                            ; otherwise loop while the entry is not empty
B indirect_branch_miss                    ; return to the DBT to handle misses

found:
ADRP X17, code_cache_base                 ; get the base address of the code cache
ADD X17, X17, X16, LSR #32                ; extract the TPC from the high bits of the entry
BR X17                                    ; branch to the TPC
```

Fig. 9. AArch64 implementation of the indirect branch lookup algorithm. On entry, `W15` holds the SPC target of the branch. `table_mask` and `table_base` are registers that hold the hash table mask and hash table base, respectively. All other registers are scratch registers.

branch table to read-write are caught and the code cache block containing the branch table is invalidated.

4. FAST ATOMIC HASH TABLES

When an indirect branch is encountered in translated code, the SPC target of the branch must be translated into a TPC address to continue execution. The standard method for doing this is to use a hash table to store a mapping of SPC-to-TPC addresses. In a *thread-shared code cache* model, which has been shown to scale significantly better than thread-private code caches [Bruening et al. 2006; Hazelwood et al. 2009], this hash table is shared among all running threads. Synchronization of hash table accesses is complicated by the strict performance requirements of indirect branch lookup: because these lookups are very frequent, adding any kind of locking for synchronization comes at an unacceptable performance cost.

4.1. Hash Table Operations

Fast atomic hash tables are based on open-addressing hash tables with linear probing. To avoid the need for locks while reading, this technique makes use of the fact that aligned 64-bit loads and stores are guaranteed to be atomic on many architectures. This is true for all 64-bit architectures and even some 32-bit architectures, such as ARMv7 with the Large Physical Address Extension (LPAE). To take advantage of these instructions, each hash table entry, consisting of an SPC and TPC pair, is packed into a 64-bit value that can be loaded or stored together atomically.

The hash table supports four operations:

Lookup. Hash table lookup performance is by far the most critical since it is the most common operation: typical programs will have billions of hash table reads for each hash table write. The hash table lookup algorithm, shown in Algorithm 1, is simple: it loads entries from the hash table one at a time using a 64-bit atomic load, stopping only when an entry with a matching SPC or an empty entry is reached. The need for bound-checking or wrap-around is eliminated by simply adding a terminating empty entry after the end of the table. For correct execution, the lookup algorithm requires that, at any time, all hash table entries must contain either a valid SPC and TPC pair or be empty. This is guaranteed by having all hash table modifications use 64-bit atomic stores. This algorithm can be implemented very efficiently, as shown in Figure 9: the implementation in MAMBO-X64 requires only 10 instructions, of which eight are executed if a matching entry is found on the first iteration. These instructions are

ALGORITHM 1: Indirect Branch Lookup Algorithm with Fast Atomic Hash Tables

```
Input: $SPC$ address of the block that should be executed next.
$Index$ = Hash($SPC$) mod $HashTableSize$;
repeat
    $Entry$ = AtomicLoad64($HashTableBase, Index$);
    $Entry_{SPC}$, $Entry_{TPC}$ = Unpack($Entry$);
    if IsEmpty($Entry_{SPC}$) then
        BranchToDBT();
    end
    $Index$ = $Index$ + 1;
until $Entry_{SPC}$ == $SPC$;
BranchTo($Entry_{TPC}$);
```

inlined directly into the translated block to avoid call overhead and allow the processor's indirect branch predictor to track each translated indirect branch separately.

Insertion. Inserting an entry into the hash table requires only a single 64-bit atomic store to add the entry, but it also requires holding a lock to prevent other threads from concurrently modifying the table. This lock is acceptable because unlike lookups, hash table additions or removals are relatively rare in typical programs, and using a lock while writing avoids the need for memory barriers in lookup code. A thread concurrently reading the table will see either the new entry or an empty entry. Potential races due to two threads attempting to insert the same entry into the table are resolved once the hash table lock is taken by checking if a matching entry already exists.

Removal. An entry needs to be removed from the hash table when a block is invalidated, such as when the instructions it is sourced from have been modified. Entry removal is similar to insertion in that it also requires holding the hash table lock but differs in its effect on concurrent lookups. While the target entry can simply be replaced with a *poisoned* entry, which will always be skipped by lookups, this causes lookup times to grow over time as the number of poisoned entries increases. A better solution is to shift the entries after the target backward, thus keeping lookup times low. In practice, this shifting usually only consists of one or two swap operations. The shifting can induce spurious failures in concurrent lookups since a lookup might miss its intended target as it is shifted past. This will result in a lookup failure and a return to the DBT, but the lookup can then be resolved by searching the hash table again while holding its lock to prevent concurrent writes. This case rarely occurs in practice and therefore has an insignificant impact on performance.

Growth. Growing the hash table is necessary to allow a DBT to scale to a wide range of applications. Since other threads may still be reading the table, it cannot be freed immediately. Resizing instead creates a new larger table into which the entries of the previous table are copied. Just before a thread starts executing translated code, it will copy a pointer to the latest version of the indirect branch hash table into thread-local storage, which is used for lookups, and increment the table's reference count. Once it returns to the DBT, the thread will decrement the reference count and free the table if it reaches zero, since that means that no more threads are using the table.

4.2. SPC and TPC Packing

For fast atomic hash tables to work, both an SPC and a TPC address must be packed into a 64-bit value. There are several ways of achieving this, depending on two factors: the pointer size of the *target* architecture, which determines the size of the SPC, and the pointer size of the *host* architecture, which determines the size of the TPC.

The simplest case is when both the host and the target use 32-bit pointers, in which case they can simply be appended to form a 64-bit value. For a 64-bit host emulating a 32-bit target, the TPC can be turned into a *code cache offset* from the start of the code cache, which can then fit in 32 bits, with the limitation that the code cache cannot exceed 4GB in size.

When both the host and the target are 64 bit, the situation is more complicated. While the TPC can be compressed as a code cache offset, the SPC needs to be fully represented to ensure transparency. Fortunately, most 64-bit architectures do not support a full 64-bit address space, instead only using a subset of those bits for virtual addresses. For example, the default configuration of Linux on AArch64 restricts the virtual address space of a process to 39 bits, which leaves 25 bits for a code cache offset, allowing a maximum code cache size of 32MB. The TPC can then be reconstructed by simply adding the 25-bit offset to the starting address of the code cache.

If the target architecture still uses too many bits for virtual addresses, a DBT can still artificially reduce the address space of a process by controlling memory allocation system calls such as `mmap`. By restricting all virtual memory allocations with an upper address limit, the DBT can guarantee that all valid SPC addresses can fit in a limited number of bits.

The case of a 32-bit host architecture emulating a 64-bit target architecture is not discussed here because it is rarely used.

5. EVALUATION

In this section, the performance of the indirect branch handling techniques is evaluated. Because the optimizations described in this article are designed to reduce the runtime overhead of running a program under a DBT, performance results are described in terms of "overhead reduction." Overhead is defined as the additional time a benchmark takes when running under a DBT compared to running the 32-bit program natively on the CPU, and overhead reduction is the percentage by which this overhead is reduced when the optimization is applied.

5.1. Experimental Setup

All experiments were conducted on a Juno ARM Development Board with two Cortex-A57 cores running at 1.1GHz and four Cortex-A53 cores running at 850MHz. The board comes with 8GB of RAM and runs Debian Unstable with Linux kernel version 3.17. To keep results consistent, all experiments were run on one of the Cortex-A57 cores, which has 48KB of L1 instruction cache, 32KB of L1 data cache, and 2MB of shared L2 cache.

The performance of the three techniques was analyzed using the SPEC CPU2006 benchmark suite. All benchmarks were compiled with GCC 4.9.1, optimization level `-O2`, and targeting 32-bit and 64-bit ARMv8-A. Figure 10 shows the distribution of indirect branch types in the SPEC CPU2006 benchmarks as a fraction of the total dynamic instruction count.

Indirect branches account for fewer than 3% of dynamic instructions executed, but they are a significant source of overhead for DBTs because they need to be translated into a lookup routine that translates an SPC address to a TPC address. While a native indirect branch instruction (`BR`) only requires a single cycle to execute on a Cortex-A57, a DBT's indirect branch lookup code usually requires 10 to 20 cycles. This explains why so few instructions can have such a significant impact on DBT performance.

These results also show that in all the benchmarks, the majority of indirect branches are function returns, which makes effective handling of returns an important factor in DBT performance. In contrast, intensive use of branch tables and other indirect

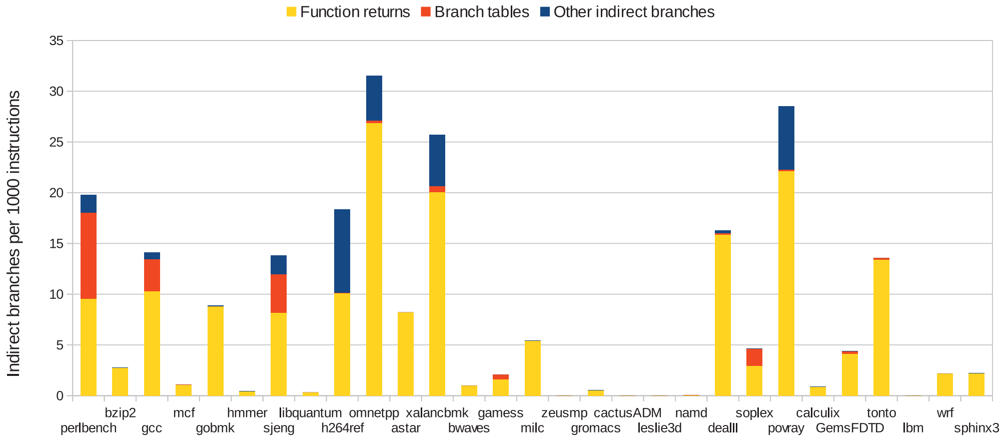


Fig. 10. Dynamic distribution of indirect branch types in SPEC CPU2006.

branches is limited to only a few benchmarks and optimization thereof is expected to have less impact on performance.

Because the ARMv8 processors used in these experiments are capable of running 32-bit ARM code directly, all benchmarks were executed natively on the processor and the results are used as a baseline for the experiments. All other results are normalized to this baseline, showing the relative performance of the DBT compared to native execution.

5.2. MAMBO-X64

To evaluate the three techniques, they were implemented on the MAMBO-X64 DBT, which translates 32-bit ARM programs to AArch64 (64-bit ARM) code. MAMBO-X64 aims to be a sufficiently fast DBT to eliminate the need for hardware support of the 32-bit ARM instruction set in AArch64 processors. MAMBO-X64 can even allow some translated ARM programs to run faster than they would if executed natively by the processor, for example, by exploiting the new instructions and additional registers available in AArch64.

Figure 11 shows the performance of the benchmarks translated from ARM to AArch64 compared to executing the ARM code directly. Three different translation methods are shown:

—The benchmarks can be translated using QEMU [Bellard 2005], a generic DBT that supports translating programs among many architectures.
—The benchmarks can be recompiled to AArch64 from source.
—The benchmarks can be translated using MAMBO-X64.

While recompiling the code for AArch64 results in the best performance in most benchmarks, this requires that the source code be available and portable to the new architecture, which may not always be the case. Additionally, pointers in AArch64 use two times more space than on 32-bit ARM, which can degrade performance due to increased memory usage and cache pressure despite the additional registers and new instructions in AArch64.

QEMU supports a large number of architectures, and therefore does not use many architecture-specific optimizations and emulates all floating-point operations in software. Together, these design choices cause QEMU's performance to suffer significantly compared to native execution.

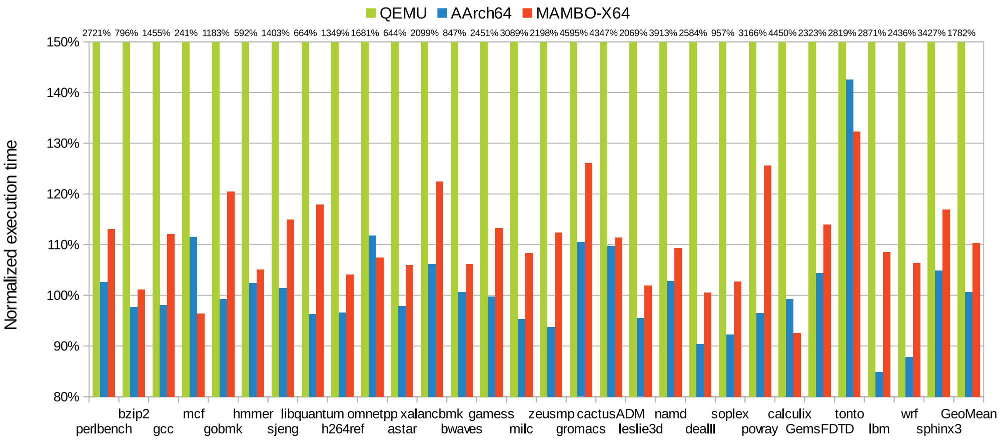


Fig. 11. Performance of various systems on SPEC CPU2006. These are QEMU translating ARM to AArch64, MAMBO-X64 translating ARM to AArch64, and recompiling the benchmarks for AArch64. Performance numbers are relative to the benchmark running natively in 32-bit mode.

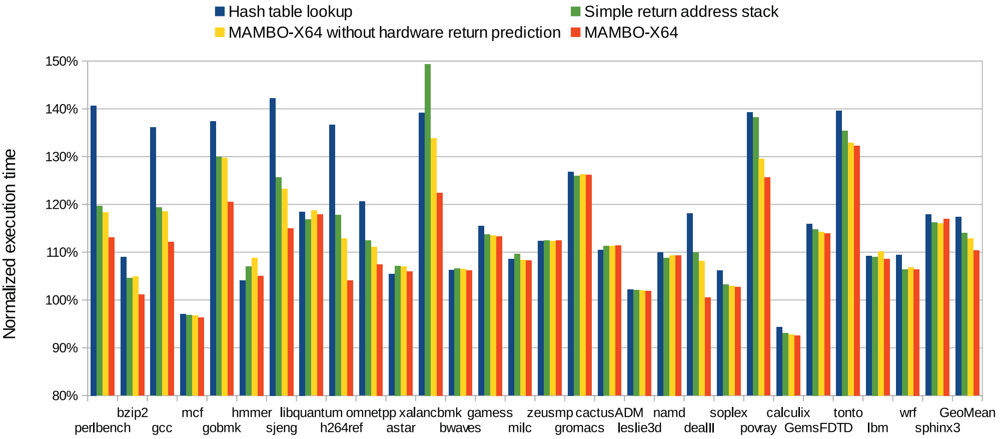


Fig. 12. MAMBO-X64 performance on SPEC CPU2006 with different ways of handling function returns. Performance numbers are relative to the benchmark running natively in 32-bit mode.

MAMBO-X64 is specialized for ARM-to-AArch64 translation, which allows it to reach near-native performance: it is only 10% slower than native execution on average, and even achieves faster speeds than native execution in some cases. This is achieved by taking advantage of the new features of the AArch64 instruction set that allow a wider range of immediate values to be encoded in instructions and allow certain operations to be done in fewer instructions. MAMBO-X64 continues to be improved and this overhead is likely to reduce further in the future.

5.3. Hardware-Assisted Function Returns

Hardware-assisted function returns give a significant performance improvement on about half of the benchmarks, as shown in Figure 12. To better understand the impact of hardware return address prediction on performance, the benchmarks were run in four configurations:

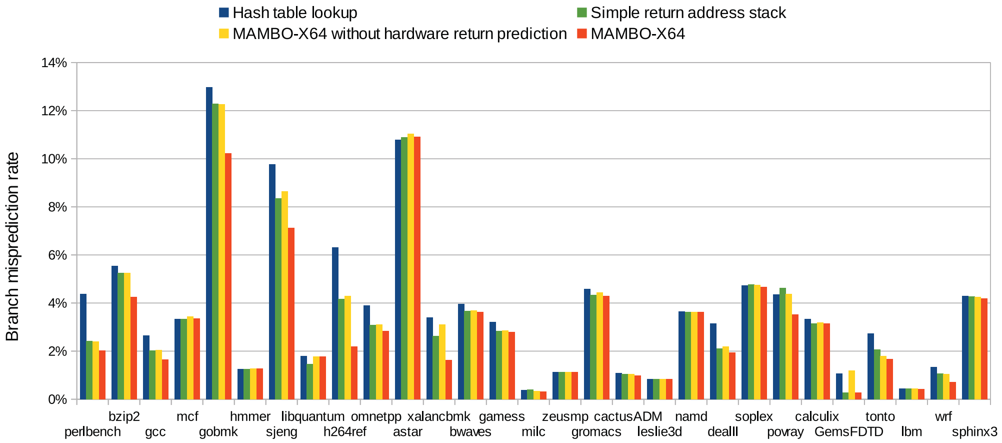


Fig. 13. Hardware branch misprediction rate of MAMBO-X64 on SPEC CPU2006 with different ways of handling function returns. The numbers show the number of branch mispredictions divided by the total number of branches executed. Note that the misprediction rate includes both direct and indirect branches.

Hash table lookup. Function returns are handled using a hash table lookup, just like other indirect branches.

Simple return address stack. A return address stack is used, similar to the one used by Pin on ARM: it does not support RAS elision or support advanced handling of mispredictions.

MAMBO-X64. All of the techniques described in Section 2 are used to optimize function return handling.

MAMBO-X64 without hardware return prediction. Similar to the previous, but normal indirect branch instructions are used instead of return instructions, effectively not utilizing the hardware return address predictor.

The results show that while most of the performance improvement comes from the use of a return address stack instead of a hash table, taking advantage of hardware return prediction still accounts for about a third of the overall speedup. Over all of the SPEC CPU2006 benchmarks, the use of a return address stack reduces overhead by 27%, and the use of hardware return address prediction reduces overhead by a further 14%. Of the former 27%, 18.5% is due to the use of a RAS while the remaining 8.5% comes from the RAS elision and misprediction handling optimizations. On benchmarks that make heavy use of function calls, such as *h264ref*, the difference is even more significant: using a return address stack reduces overhead by 65%, and hardware return address prediction reduces it by a further 23%.

Figure 13 shows the hardware branch misprediction rate (fraction of branch instructions that were mispredicted) when running MAMBO-X64 under the same four configurations. Unfortunately, the Cortex-A57 does not use separate performance counters for direct and indirect branches, so only the combined branch misprediction rate is shown. These results match the previous ones, which shows that both aspects of hardware-assisted function returns improve DBT performance by reducing branch mispredictions. The additional mispredictions when using a hash table instead of a RAS are due to cases where the hardware fails to predict the direct branch in the hash table lookup loop in addition to the indirect branch after the loop.

The RAS can very accurately predict the targets of function returns: the majority of benchmarks do not have any RAS mispredictions. Some benchmarks (*perlbench*, *omnetpp*, and *povray*) have RAS mispredictions due to the use of stack unwinding

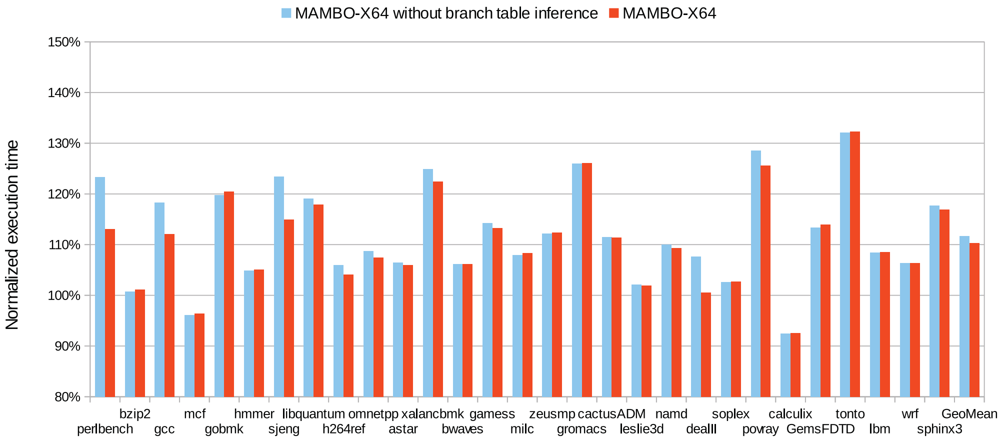


Fig. 14. MAMBO-X64 performance on SPEC CPU2006 with and without branch table inference. Performance numbers are relative to the benchmark running natively in 32-bit mode.

through C++ exceptions and the C `longjmp` function, but these are still extremely rare: the misprediction rate did not exceed one in 10 million for any benchmark, which makes the overhead insignificant.

Another set of benchmarks (*perlbench*, *gcc*, *gobmk*, and *xalancbmk*) suffer from RAS mispredictions due to RAS overflow. MAMBO-X64 uses a 4KB RAS for each thread, which can hold 512 entries. When the RAS overflows, its contents are shifted down, which causes it to lose some of its entries. This is most visible in *xalancbmk*, which overflows the RAS 668,664 times, causing a RAS misprediction rate of 0.3%. To avoid context-switching back to the DBT to handle each misprediction, MAMBO-X64 uses the hash table mechanism described in Section 2.4 to handle overflow-related mispredictions efficiently. This significantly improves performance on *xalancbmk*, where a third of the DBT overhead was due to time spent handling the misprediction in the DBT.

A significant factor in the performance of MAMBO-X64 is the use of RAS elision, which was applied to 51% of all function returns executed in the benchmarks. This optimization effectively eliminates the overhead of translated function returns compared to native execution. On two benchmarks that make heavy use of function returns, *h264ref* and *dealII*, over 80% of function returns were optimized with RAS elision.

5.4. Branch Table Inference

Figure 14 shows the impact of branch table inference on the benchmarks when run under MAMBO-X64. This optimization mainly affects benchmarks that make heavy use of branch tables: *perlbench* benefits the most because it is an interpreter that makes heavy use of `switch` statements, where branch table inference reduces DBT overhead by 40%. Despite this, the overhead reduction over all of the benchmarks is only 10% because few benchmarks make heavy use of branch tables.

5.5. Fast Atomic Hash Tables

To evaluate the performance of fast atomic hash tables, three alternative indirect branch handling mechanisms were implemented on MAMBO-X64. The first, called "megatables," is based on SPIRE [Jia et al. 2013] but is simplified by exploiting the fact that MAMBO-X64 runs in a 64-bit address space while the program it is translating only uses a 32-bit address space. Instead of using a small hash table, a huge 16GB table

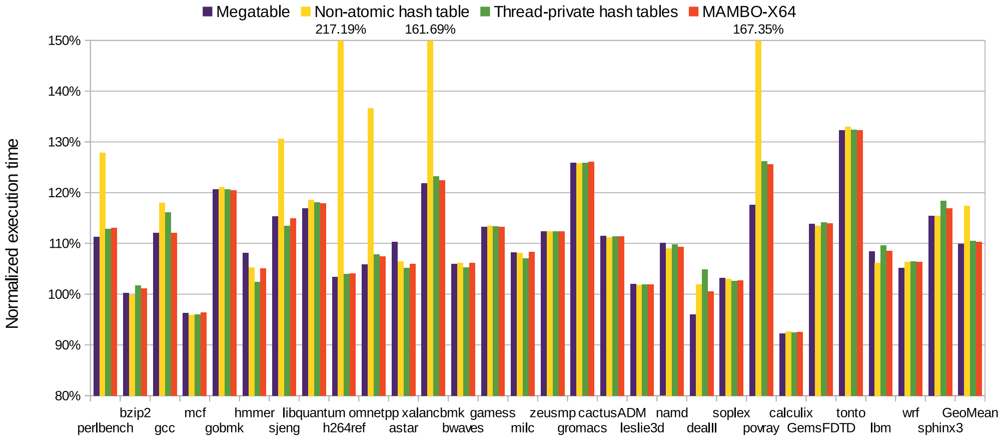


Fig. 15. MAMBO-X64 performance on SPEC CPU2006 with various indirect branch handling techniques. Performance numbers are relative to the benchmark running natively in 32-bit mode.

is allocated, which contains a 4-byte code cache offset for every possible 32-bit address. Handling an indirect branch then simply consists of loading the entry in the table for the given target address and branching to the translated block it points to. While the table consumes a large amount of virtual memory, the operating system will only allocate memory for pages that contain entries, while pages with no entries will simply be mapped to a common zero-filled page. Table modifications are atomic because adding or removing an entry is simply an aligned 32-bit store. As with fast atomic hash tables, entries are added to the megatable lazily, only when an indirect branch lookup misses.

The second mechanism is a hash table that does not require 64-bit atomic operations, similar to the one used by DynamoRIO [Bruening et al. 2006]. Like fast atomic hash tables, it also uses a single hash table shared among all threads, but only uses 32-bit loads and stores when accessing the hash table, writing to the SPC and TPC parts of a hash table entry separately. Because of this, backward shift deletion cannot be used to compact the hash table when an entry is deleted; instead, the deleted entry must be "poisoned" by replacing the TPC of the entry with the address of a routine that returns control to the DBT. Once an entry has been poisoned, it cannot be reused since another thread may be concurrently reading that entry in a lookup. This causes the average number of entries scanned during a hash table lookup to increase as entries are added and removed, which can degrade performance. Another disadvantage of this method is that, because of the ARM architecture's weakly ordered memory model, a memory barrier is needed between reading the SPC of an entry and reading its TPC. This memory barrier is part of the performance-critical lookup code and therefore has a significant effect on performance.

The last mechanism is thread-private hash tables, which use a separate hash table for each thread and therefore do not require any synchronization during lookups. This is the mechanism most commonly implemented in DBT due to its simplicity and the fact that it works with thread-private code caches. Despite these advantages, it has been shown to not scale well to large numbers of threads [Bruening et al. 2006; Hazelwood et al. 2009] since each thread requires its own table. Block invalidation is also more complicated because a block needs to be removed from the hash tables of all live threads, although unlike thread-shared hash tables, this can be done without needing a memory barrier in the lookup code.

Figure 15 shows the performance of MAMBO-X64 with fast atomic hash tables, nonatomic hash tables, thread-private hash tables, and megatables. In most

Table I. Memory Usage of Megatables Compared to Fast Atomic Hash Tables on SPEC CPU2006

Benchmark	Indirect Branch Targets	Hash Table Size (KB)	Megatable Size (KB)	Ratio
perlbench	316	7.1	431.6	60.7
bzip2	32	1.0	52.0	52.0
gcc	748	14.5	771.2	53.1
mcf	38	1.0	52.0	52.0
gobmk	1021	21.6	347.9	16.1
hmmer	51	1.0	75.9	75.9
sjeng	43	1.0	68.0	68.0
libquantum	29	1.0	48.0	48.0
h264ref	67	1.3	94.4	72.0
omnetpp	497	9.0	388.0	43.1
astar	38	1.0	60.0	60.0
xalancbmk	1120	18.0	1208.0	67.1
bwaves	46	1.0	60.0	60.0
gamess	59	1.0	80.0	80.0
milc	40	1.0	56.0	56.0
zeusmp	48	1.0	52.0	52.0
gromacs	59	1.0	96.0	96.0
cactusADM	127	2.5	200.0	80.0
leslie3d	55	1.0	76.0	76.0
namd	53	1.5	112.0	74.7
dealII	193	5.0	280.0	56.0
soplex	182	4.5	233.9	52.0
povray	175	4.0	248.0	62.0
calculix	65	1.5	68.0	45.3
GemsFDTD	53	1.0	68.0	68.0
tonto	69	1.5	104.0	69.3
lbm	36	1.0	56.0	56.0
wrf	64	1.5	76.0	50.7
sphinx	49	1.0	68.0	68.0
Geometric mean ratio				58.7

benchmarks, the performance of these techniques is similar, but some benchmarks that make heavy use of indirect branches, such as *h264ref*, suffer a very large performance degradation due to the use of a memory barrier in thread-shared hash tables. The other three techniques all have very close performance, but some benchmarks, such as *povray* or *dealII*, suffer from hash table collisions that megatables do not suffer from. These results show that fast atomic hash tables have the performance of thread-private hash tables while preserving the superior scalability of thread-shared hash tables.

While they have similar performance and both only use a single shared table, the memory usage of megatables is much higher than fast atomic hash tables, as shown in Table I. This is because megatables cause the operating system to allocate a full 4KB page for every page of the table that contains entries, whereas the size of a hash table is proportional to the total number of entries in that table. This shows that fast atomic hash tables are competitive with existing indirect branch handling techniques while consuming significantly less memory.

6. RELATED WORK

Because of its significance, many approaches have been designed to reduce the overhead of indirect branches in DBTs. Some of these are generic and can be applied to all types of indirect branches, while others only apply to a single type.

6.1. Indirect Branch Handling

The most common way that DBTs handle indirect branches is by using a hash table to map SPC addresses to TPC addresses. The lookup is typically done using a heavily optimized assembly code routine, which can be either called like a function or inlined directly inside a block. In some DBTs [Bruening et al. 2012], these routines also need to save and restore registers to provide scratch registers to work with.

DynamoRIO [Bruening et al. 2006] implements support for thread-shared indirect branch hash tables, but these are disabled by default because they result in lower performance than thread-private hash tables. This implementation is also specific to x86 and will not work on architectures with a weak memory model such as ARM without expensive memory barriers in the performance-critical lookup code.

The Indirect Branch Translation Cache (IBTC) [Scott et al. 2004] mechanism uses per-branch hash tables instead of a global hash table shared by all indirect branches. While it can potentially offer better hit rates, this approach suffers from increased memory usage compared to a global hash table.

A different approach to handling indirect branches is to use software prediction [Luk et al. 2005; Bruening et al. 2003]. This technique consists of comparing the branch target with a predefined SPC and branching to the corresponding TPC if the comparison succeeds. Multiple predicted targets can be checked this way, eventually falling back to a hash table lookup if none of the comparisons succeeds. Software Prediction with Target Updating (SPTU) [Jia et al. 2014b] is an improvement on this technique, which updates the predicted targets according to their frequency. While this approach works well on architectures like x86, which can include an entire word-sized immediate operand with a compare instruction, it is less effective on RISC architectures such as ARM and AArch64, which only support a limited set of values as immediate operands.

HDTrans [Sridhar et al. 2005] uses a technique called SIEVE, which combines software prediction and hash tables. First, the branch target SPC is hashed and used to index a branch table. Each entry in the branch table leads to a chain of compare and branch instructions for all targets in that hash bucket. While SIEVE requires fewer registers than a standard hash table lookup and can take advantage of hardware branch prediction, it suffers from the same issues as software prediction on RISC architectures.

SPc-Indexed REdirecting (SPIRE) [Jia et al. 2013] handles indirect branches by patching the instruction at the SPC in the original program to branch to the TPC, thus allowing an indirect branch to jump to the SPC directly, avoiding the need for address translation. While this scheme can efficiently handle indirect branches, it is complicated to implement efficiently and, as mentioned by the authors, can lead to incorrect code execution in some edge cases.

The fastBT DBT [Payer and Gross 2010] uses shadow jump tables to optimize handling of branch tables, which are a subset of indirect branches. When an indirect branch is recognized as a branch table instruction, fastBT will create a shadow table in the code cache containing the TPC addresses of all the branch table targets. The main downside of this technique is that it sometimes ends up creating shadow tables that are too large or too small because it uses fixed-size tables instead of inferring the size from the instructions.

Direct-TPC Tables (DTTs) [Jia et al. 2014a] extend shadow jump tables to apply to all indirect branches that load an address from an array, such as virtual function calls. This is done by shadowing large blocks of program memory and treating them as large branch tables. An indirect branch that loads from an array can then be translated into a load from the shadow table, which will contain the TPC address for the entry.

The Jump Target Lookup Table (JTLT) [Kim and Smith 2003] proposed hardware extensions that add a small hardware cache containing a mapping of SPC addresses to TPC addresses. When the processor executes an indirect branch instruction, the JTLT is used to find the TPC address for the SPC branch target. If a matching entry is found in the JTLT, then the processor branches to the TPC address in the entry. Unfortunately, this technique requires special instructions and therefore is not usable on existing architectures.

6.2. Function Return Handling

Research shows that function returns are by far the most common type of indirect branch [Scott et al. 2004]. Function returns are different from most other indirect branches in that they almost always return to a matching call instruction. Many techniques have been designed to take advantage of this property to accelerate return handling in DBTs.

While return address stacks have been used in existing DBTs [Hazelwood and Klauser 2006], they do not take advantage of hardware return prediction, instead only using generic indirect branches that are not predicted as returns by the processor. They also do not implement the optimizations described in this article for eliding return address stack operations and for efficient handling of return address stack mispredictions.

An experimental version of DynamoRIO [Bruening 2004] did attempt to combine a software return address stack with hardware return prediction, but the resulting performance was worse than that of a hash table lookup. This was caused by excessive memory operations and because call return targets were not kept in the same block as the call instruction itself.

Pin [Luk et al. 2005] uses *function cloning* to create a different translated copy of a function for each site it is called from. Because each clone is specialized for a single caller, which is known at translation time, a function return can be translated to a compare and branch to the return target. While this provides an accurate prediction of the return address in most cases, it comes at a significant cost in memory and instruction cache locality due to code duplication since a new clone needs to be generated for each site a function is called from.

Fast returns [Scott et al. 2004] consist of translating call instructions such that they generate the TPC of the return target instead of the SPC for the program-visible return address. This allows return instructions to avoid an SPC-to-TPC lookup because the return address is a TPC value instead of an SPC value. Moore et al. [2009] propose a variant of this called *checked fast returns,* which adds a check to the return instruction to ensure the address is a valid TPC value. While these techniques offer near-native performance for translated return instructions, they can cause applications to malfunction if they attempt to read the return address of a function, for example, when generating stack traces or when unwinding the stack.

The *return cache* [Sridhar et al. 2005; Payer and Gross 2010] is a different approach that works by using a direct-mapped hash table that holds return target TPC addresses and is indexed by a hash of the return target SPC address. A return instruction is handled by simply branching to an entry in the return cache. Hash collisions are handled at the return target by comparing the SPC used for the lookup with the SPC of the return target. The downsides of this approach are that hash collisions are not handled efficiently and that it interacts poorly with hardware branch prediction mechanisms.

The *dual-address return address stack* [Kim and Smith 2003] is a technique that requires modifying the processor hardware. It works by extending the hardware return address predictor to track both SPC and TPC addresses. Since this techniques requires hardware support, it is not usable on existing architectures.

7. CONCLUSIONS

This article has presented three novel techniques that improve the performance of indirect branches in dynamic binary translators. These techniques have been implemented in MAMBO-X64, a dynamic binary translator that translates 32-bit ARM programs into 64-bit ARMv8 code, and their performance impact was evaluated using the SPEC CPU2006 benchmarks. Together, these techniques allow MAMBO-X64 to achieve a very low performance overhead of only 10% on average compared to native execution of 32-bit programs. This makes processors that only support 64-bit execution mode more viable since legacy applications are still able to run at an acceptable performance.

The first technique, *hardware-assisted function returns*, tracks the SPC and TPC addresses of executed call instructions in a software return address stack so that subsequent return instructions can use the last entry on the stack as a predicted branch target, thus avoiding the overhead of SPC-to-TPC translation. This extends previous work on return address stacks by combining it with a novel layout for translated code that allows the use of the hardware return address predictor in translated code, as well as optimizations to elide return address stack operations and to better handle return address stack mispredictions. This technique has the highest impact when applied to SPEC CPU2006 running under MAMBO-X64, reducing DBT overhead by 40% on average and by up to 90% on some benchmarks. This approach has significant benefits over techniques based on hash tables because it significantly reduces the number of CPU branch mispredictions.

The second technique, *branch table inference*, is an algorithm for recognizing and translating certain code patterns that are used for branch tables. This allows branch tables to be translated as multiway direct branches rather than as indirect branches by reading the source branch table and generating a corresponding branch table in the translated code. This optimization completely eliminates DBT overhead on branch tables and significantly improves the performance of benchmarks that make extensive use of branch tables, achieving an overhead reduction of 40% on some benchmarks. While detection of branch tables has previously been used in DBTs, branch table inference provides a systematic way of detecting many variants of branch tables and extracting the bounds of the table directly from the source instructions instead of guessing it.

The last technique, *fast atomic hash tables*, takes advantage of the fact that aligned 64-bit loads are guaranteed to be atomic on 64-bit architectures and some 32-bit architectures to perform fast indirect branch lookups on a thread-shared hash table. This avoids the need for memory barriers in performance-critical lookup code, which reduces DBT overhead by 40% compared to a DBT that uses memory barriers. It matches the performance of thread-private hash tables while scaling much better to a large number of threads. It also matches the performance of SPIRE, an existing indirect branch handling technique, while consuming 50 times less memory.

While these techniques have been shown to be effective on MAMBO-X64 for translation of 32-bit ARM executables to AArch64, they are also applicable on a wider range of architectures. Branch table inference can work on any architecture as long as a branch table instruction sequence can be recognized. Hardware-assisted function returns can also be generalized, although some additional overhead may be introduced by the need to have scratch registers. Finally, fast atomic hash tables can work on 64-bit architectures but have limitations on 32-bit architectures, as described earlier.

REFERENCES

Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for*

Programming Languages and Operating Systems (ASPLOS'06). ACM, 2–13. DOI:http://dx.doi.org/10.1145/1168857.1168860

Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*. USENIX, 41–46.

Derek Bruening, Timothy Garnett, and Saman P. Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the 1st IEEE/ACM International Symposium on Code Generation and Optimization (CGO'03)*. IEEE Computer Society, 265–275. DOI:http://dx.doi.org/10.1109/CGO.2003.1191551

Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji. 2006. Thread-shared software code caches. In *Proceedings of the 4th IEEE/ACM International Symposium on Code Generation and Optimization (CGO'06)*. IEEE Computer Society, 28–38. DOI:http://dx.doi.org/10.1109/CGO.2006.36

Derek Bruening, Qin Zhao, and Saman P. Amarasinghe. 2012. Transparent dynamic instrumentation. In *Proceedings of the 8th International Conference on Virtual Execution Environments (VEE'12)*. ACM, 133–144. DOI:http://dx.doi.org/10.1145/2151024.2151043

Derek Lane Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Massachusetts Institute of Technology.

James C. Dehnert, Brian Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The transmeta code morphing - software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the 1st IEEE/ACM International Symposium on Code Generation and Optimization (CGO'03)*. IEEE Computer Society, 15–24. DOI:http://dx.doi.org/10.1109/CGO.2003.1191529

Kim M. Hazelwood and Artur Klauser. 2006. A dynamic binary instrumentation engine for the ARM architecture. In *Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'06)*. ACM, 261–270. DOI:http://dx.doi.org/10.1145/1176760.1176793

Kim M. Hazelwood, Greg Lueck, and Robert Cohn. 2009. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *Proceedings of the 8th International Symposium on Memory Management (ISMM'09)*, Hillel Kolodner and Guy L. Steele Jr. (Eds.). ACM, 20–29. DOI:http://dx.doi.org/10.1145/1542431.1542435

Jason Hiser, Daniel W. Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. 2007. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Proceedings of the 5th International Symposium on Code Generation and Optimization (CGO'07)*. IEEE Computer Society, 61–73. DOI:http://dx.doi.org/10.1109/CGO.2007.10

Raymond J. Hookway and Mark A. Herdeg. 1997. DIGITAL FX!32: Combining emulation and binary translation. *Digital Technical Journal* 9, 1 (1997), 3–12. http://www.hpl.hp.com/hpjournal/dtj/vol9num1/vol9num1art1.pdf.

R. Nigel Horspool and Nenad Marovac. 1980. An approach to the problem of detranslation of computer programs. *Computer Journal* 23, 3 (1980), 223–229. DOI:http://dx.doi.org/10.1093/comjnl/23.3.223

Ning Jia, Chun Yang, Yu He, and Xu Cheng. 2014a. DTT: Program structure-aware indirect branch optimization via direct-TPC-table in DBT system. In *Proceedings of the Computing Frontiers Conference (CF'14)*. ACM, 12:1–12:10. DOI:http://dx.doi.org/10.1145/2597917.2597944

Ning Jia, Chun Yang, Yu He, and Xu Cheng. 2014b. SPTU: Improving dynamic binary translation through software prediction with target updating. In *Proceedings of the International Conference on Systems and Storage (SYSTOR'14)*. ACM, 2:1–2:12. DOI:http://dx.doi.org/10.1145/2611354.2611368

Ning Jia, Chun Yang, Jing Wang, Dong Tong, and Keyi Wang. 2013. SPIRE: Improving dynamic binary translation through SPC-indexed indirect branch redirecting. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'13*). ACM, 1–12. DOI:http://dx.doi.org/10.1145/2451512.2451516

Ho-Seop Kim and James E. Smith. 2003. Hardware support for control transfers in code caches. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*. ACM/IEEE Computer Society, 253–264. DOI:http://dx.doi.org/10.1109/MICRO.2003.1253200

Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. ACM, 190–200. DOI:http://dx.doi.org/10.1145/1065010.1065034

Ryan W. Moore, José Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason Hiser. 2009. Addressing the challenges of DBT for the ARM architecture. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'09)*. ACM, 147–156. DOI:http://dx.doi.org/10.1145/1542452.1542472

Tipp Moseley, Daniel A. Connors, Dirk Grunwald, and Ramesh Peri. 2007. Identifying potential parallelism via loop-centric profiling. In *Proceedings of the 4th Conference on Computing Frontiers*. ACM, 143–152. DOI:http://dx.doi.org/10.1145/1242531.1242554

Mathias Payer and Thomas R. Gross. 2010. Generating low-overhead dynamic binary translators. In *Proceedings of SYSTOR 2010: The 3rd Annual Haifa Experimental Systems Conference*. ACM. DOI:http://dx.doi.org/10.1145/1815695.1815724

Yukinori Sato, Yasushi Inoguchi, and Tadao Nakamura. 2011. On-the-fly detection of precise loop nests across procedures on a dynamic binary translation system. In *Proceedings of the 8th Conference on Computing Frontiers*. ACM, 25. DOI:http://dx.doi.org/10.1145/2016604.2016634

Kevin Scott, Naveen Kumar, Bruce R. Childers, Jack W. Davidson, and Mary Lou Soffa. 2004. Overhead reduction techniques for software dynamic translation. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*. IEEE Computer Society. DOI:http://dx.doi.org/10.1109/IPDPS.2004.1303224

Julian Seward and Nicholas Nethercote. 2005. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the 2005 USENIX Annual Technical Conference*. USENIX, 17–30. http://www.usenix.org/events/usenix05/tech/general/seward.html

Swaroop Sridhar, Jonathan S. Shapiro, and Prashanth P. Bungale. 2005. HDTrans: A low-overhead dynamic translator. In *Proceedings of the 2005 Workshop on Binary Instrumentation and Applications*. IEEE Computer Society.

Jon Watson. 2008. Virtualbox: Bits and bytes masquerading as machines. *Linux Journal* 2008, 166 (2008), 1.

Qin Zhao, David Koh, Syed Raza, Derek Bruening, Weng-Fai Wong, and Saman P. Amarasinghe. 2011. Dynamic cache contention detection in multi-threaded applications. In *Proceedings of the 7th International Conference on Virtual Execution Environments (VEE'11)*. ACM, 27–38. DOI:http://dx.doi.org/10.1145/1952682.1952688

Received June 2015; revised December 2015; accepted December 2015