OPTIMISING DYNAMIC BINARY MODIFICATION ACROSS ARM MICROARCHITECTURES

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Science and Engineering

2017

By
Cosmin Gorgovan
School of Computer Science

Contents

Abstract **11**

Declaration **12**

Copyright **13**

Acknowledgements **14**

1 Introduction **16**

1.1 Dynamic binary modification 16

1.1.1 General principles 16

1.1.2 Uses 17

1.1.3 Overhead 17

1.1.4 DBM for ARM 18

1.2 The ARM architecture 19

1.2.1 Emergence 19

1.2.2 The ARM hardware ecosystem 19

1.3 Motivation 20

1.4 Contributions 21

1.5 Publications 22

1.6 Thesis structure 23

2 Background and related work **25**

2.1 Binary modification 25

2.1.1 Static and dynamic binary modification 26

2.2 Transparency 26

2.3 The implementation of dynamic binary modification 27

2.3.1 Tool injection / application loading 28

2.3.2 Code scanners . . . 29
2.3.3 Code caches . . . 30
2.4 Branch linking . . . 33
2.4.1 Direct branches . . . 33
2.4.2 Indirect branches . . . 34
2.5 Performance overhead . . . 41

3 Overview of MAMBO **43**
3.1 Introduction . . . 43
3.2 Aims and current state . . . 44
3.3 The ARM architecture . . . 44
3.4 Scratch space . . . 46
3.5 Executable loader . . . 47
3.5.1 The userspace ELF Loader . . . 48
3.6 Code cache . . . 49
3.7 Code scanner . . . 50
3.8 System call interception . . . 51
3.9 Test and development methodology . . . 51
3.10 Plugins . . . 53
3.11 Transparency . . . 53
3.12 Summary . . . 54

4 Branch linking **56**
4.1 Introduction . . . 56
4.2 Indirect branches . . . 56
4.2.1 Function returns: low overhead return address prediction . . . 57
4.2.2 Table branches: space-efficient linking . . . 60
4.2.3 Inline hash lookup for indirect branches . . . 63
4.2.4 Fallthrough branch linking . . . 65
4.2.5 Indirect branch target prediction . . . 66
4.3 Direct branches . . . 67
4.3.1 Direct branch linking . . . 67
4.3.2 Eliding unconditional direct branches . . . 68
4.4 Evaluation . . . 70
4.4.1 Experimental setup . . . 70
4.4.2 Contribution of different optimisations . . . 73

4.4.3 Comparison of the space-efficient and fastBT table branch linking schemes . . . 74
4.4.4 Overall performance . . . 76
4.4.5 Code cache size . . . 79
4.5 Summary . . . 81

5 Microarchitectural optimisations 82
5.1 Introduction . . . 82
5.2 Traces . . . 83
5.2.1 Trace heads . . . 85
5.2.2 Trace building . . . 87
5.2.3 Trace size limits . . . 88
5.2.4 Summary . . . 88
5.3 Indirect branches . . . 89
5.3.1 Hardware-assisted return address prediction . . . 90
5.3.2 Low footprint inline hash table lookup dispatch . . . 93
5.3.3 Adaptive indirect branch inlining . . . 96
5.3.4 Huge pages . . . 100
5.4 Evaluation . . . 101
5.4.1 Experimental setup . . . 101
5.4.2 Trace creation threshold . . . 103
5.4.3 Overall performance . . . 104
5.4.4 Performance counter analysis . . . 105
5.5 Summary . . . 122

6 Conclusions and future work 124
6.1 Summary and conclusions . . . 124
6.2 Optimisation selection guidelines . . . 127
6.3 Portability to AArch64 . . . 127
6.4 Future work . . . 130
6.4.1 Asynchronous multithreaded trace generation . . . 130
6.4.2 Automatic optimisation of DBM plugins . . . 131
6.4.3 Dynamic microarchitectural optimisations . . . 132
6.4.4 Trace layout optimisations . . . 132

Bibliography 133

A Example plugin **140**

B The full evaluation results **142**

C Raw performance counter values **148**

Word Count: 36686

List of Tables

1.1 Structure of the thesis. 24

3.1 Linux system calls discarded, emulated or otherwise modified by MAMBO 52

4.1 Number of branch mispredictions on Jetson TK1 with different implementations of shadow branches tables. 76
4.2 Summary of geometric mean overheads for MAMBO, Valgrind and QEMU running SPEC CPU2006. 76
4.3 Code cache size for SPEC CPU2006, in KiB and number of basic blocks. 80

5.1 Overview of the NET algorithm. 84
5.2 Comparison of MAMBO traces and NET. 88
5.3 Overview of the systems used for evaluation. 101
5.4 The MAMBO baseline, *+traces* and the configuration with the lowest overhead for SPEC CPU2006 on each system. 104
5.5 Overhead under the baseline MAMBO configuration, on ODROID-XU3 (LITTLE cluster). 108
5.6 Overhead under the baseline MAMBO configuration, on ODROID-X2. 109
5.7 Overhead under the baseline MAMBO configuration, on Transmart R28. 109
5.8 Overhead under the baseline MAMBO configuration, on Jetson TK1.110
5.9 Overhead under the baseline MAMBO configuration, on APM X-C1.110
5.10 Overhead of the *traces* optimisation (relative to baseline), on ODROID-XU3. 112
5.11 Overhead of the *traces* optimisation (relative to baseline), on ODROID-X2. 112

5.12 Overhead of the *traces* optimisation (relative to baseline), on Tronsmart R28. 113
5.13 Overhead of the *traces* optimisation (relative to baseline), on Jetson TK1. 113
5.14 Overhead of the *traces* optimisation (relative to baseline), on APM X-C1. 114
5.15 Overhead of the *hw_ras* optimisation (relative to *+traces*), on ODROID-XU3. 115
5.16 Overhead of the *hw_ras* optimisation (relative to *+traces*), on ODROID-X2. 115
5.17 Overhead of the *hw_ras* optimisation (relative to *+traces*), on Tronsmart R28. 115
5.18 Overhead of the *hw_ras* optimisation (relative to *+traces*), on Jetson TK1. 116
5.19 Overhead of the *hw_ras* optimisation (relative to *+traces*), on APM X-C1. 116
5.20 Overhead of the *hugetlb* optimisation (relative to *+hw_ras +traces*), on Jetson TK1. 117
5.21 Overhead of the *hugetlb* optimisation (relative to *+hw_ras +traces*), on APM X-C1. 117
5.22 Overhead of the *ldm_pc_sr* optimisation (relative to *+traces*), on ODROID-XU3. 118
5.23 Overhead of the *ldm_pc_sr* optimisation (relative to *+traces*), on ODROID-X2. 118
5.24 Overhead of the *ldm_pc_sr* optimisation (relative to *+traces*), on Tronsmart R28. 119
5.25 Overhead of the *ldm_pc_sr* optimisation (relative to *+traces*), on Jetson TK1. 119
5.26 Overhead of the *ldm_pc_sr* optimisation (relative to *+traces*), on APM X-C1. 120
5.27 Overhead of the *aibi* optimisation (relative to *+traces*), on ODROID-XU3. 120
5.28 Overhead of the *aibi* optimisation (relative to *+traces*), on ODROID-X2. 120

5.29 Overhead of the *aibi* optimisation (relative to *+traces*), on Tronsmart R28. 121
5.30 Overhead of the *aibi* optimisation (relative to *+traces*), on Jetson TK1. 121
5.31 Overhead of the *aibi* optimisation (relative to *+traces*), on APM X-C1. 122

6.1 Optimisation selection guidelines 128
6.2 Portability of the optimisations to AArch64 129

C.1 The raw performance counter values for the native execution of SPEC CPU2006 on ODROID-XU3 (thousands). 149
C.2 The raw performance counter values for the native execution of SPEC CPU2006 on ODROID-X2 (thousands). 149
C.3 The raw performance counter values for the native execution of SPEC CPU2006 on Tronsmart R28 (thousands). 150
C.4 The raw performance counter values for the native execution of SPEC CPU2006 on Jetson TK1 (thousands). 150
C.5 The raw performance counter values for the native execution of SPEC CPU2006 on APM X-C1 (thousands). 151

List of Figures

2.1 The components of a DBM system and the control flow between them. 27

3.1 The ARM registers. 44
3.2 Translation using scratch registers for an instruction which accesses the stack. 48
3.3 Structure of an ELF file. 48
3.4 MAMBO data structures for an example basic block. 49

4.1 Example of a typical function call and the translation generated by MAMBO. 57
4.2 Space-efficient shadow branch table. 61
4.3 Comparison of shadow branch table size for SPEC CPU2006. . . . 61
4.4 Inline hash lookup routine. 64
4.5 Linking of fallthrough branches. 66
4.6 Comparison between translations with and without unconditional direct branch eliding. 68
4.7 Comparison of basic block sizes. 69
4.8 Relative execution time for SPEC CPU2006 with the *ref* dataset on ODROID-X2. 71
4.9 Relative execution time for SPEC CPU2006 with the *ref* dataset on Jetson TK1. 72
4.10 Relative slowdown for selected SPEC CPU2006 benchmarks with the fastBT table branch linking scheme. 75
4.11 Relative execution time for SPEC CPU2006 under MAMBO, Valgrind and QEMU (*ref* dataset). 77
4.12 Relative execution time for PARSEC 3.0 with the *native* dataset. 79

5.1 Example control flow graph. Each box represents a basic block. Block A, the entry point, contains a conditional direct branch, block C contains an unconditional indirect branch and all other blocks contain unconditional direct branches. 85
5.2 Inline hash table lookup. 89
5.3 Example of a typical function call. 91
5.4 Comparison of hit rates on a selection of SPEC CPU2006 benchmarks for indirect branch predictors. 96
5.5 Adaptive indirect branch inlining. 98
5.6 Trace creation threshold vs relative execution time and trace code size 104

B.1 Relative execution time for SPEC CPU2006 with the *ref* dataset on ODROID-XU3 - with microarchitectural optimisations. 143
B.2 Relative execution time for SPEC CPU2006 with the *ref* dataset on ODROID-X2 - with microarchitectural optimisations. 144
B.3 Relative execution time for SPEC CPU2006 with the *ref* dataset on Tronsmart R28 - with microarchitectural optimisations. 145
B.4 Relative execution time for SPEC CPU2006 with the *ref* dataset on Jetson TK1 - with microarchitectural optimisations. 146
B.5 Relative execution time for SPEC CPU2006 with the *ref* dataset on APM X-C1 - with microarchitectural optimisations. 147

Abstract

Optimising Dynamic Binary Modification Across ARM Microarchitectures

Cosmin Gorgovan

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2017

Dynamic Binary Modification (DBM) is a technique for modifying applications at runtime, working at the level of native code. It has numerous applications, including instrumentation, translation and optimisation. However, DBM introduces a performance overhead, which in some cases can dominate execution time, making many uses impractical.

While avenues for reducing this overhead have been widely explored on x86, ARM, an architecture gaining widespread adoption, has received little attention. Consequently, the overhead of DBM on ARM, as reported in the literature and measured using the available DBM systems, has fallen behind the state-of-the-art by one or two orders of magnitude. The research questions addressed in this thesis are: 1) how to develop low overhead DBM systems for the ARM architecture, and 2) whether new optimisations are plausible and needed.

Towards that end, a number of novel optimisations were developed and evaluated specifically to address the sources of overhead for DBM on various ARM microarchitectures. Furthermore, many of the optimisations in the literature were ported to ARM and evaluated. This work was enabled by a new DBM system, named *MAMBO*, created specifically for this purpose. MAMBO, using the optimisations presented in this thesis, is able to achieve an overhead an order of magnitude smaller than that of the most efficient DBM system for ARM available at the start of this PhD.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

First of all, I would like to thank my supervisor, Mikel Luján, for his support, guidance and contributions during these past four years. His broad experience has been most helpful in guiding my research. In particular, Mikel has always encouraged me to consider the broader context, rather than getting caught up in the details of dynamic binary modification systems, for which I am thankful.

I would also like to thank Alasdair Rawsthorne for encouraging me to start a PhD and also for introducing me to the fields of dynamic binary modification and virtualisation. Had I not done the undergraduate final year project on system virtualisation for ARM, proposed and supervised by Alasdair, chances are that I would have never started this PhD.

I would also like to thank Oscar Palomar Perez, who has reviewed this thesis and has provided very useful and thorough feedback despite the tight timeline caused by my abysmal time management skills. Thanos Stratikopoulos has also provided feedback on this thesis, for which he has my gratitude.

I would like to thank Amanieu d'Antras and Guillermo Callaghan, the two other PhD students with whom I have worked on a regular basis. Amanieu was working on dynamic binary translation during the time I was working on dynamic binary modification. While we often do not agree on the finer points of mangling binaries, we have debated these points and occasionally joined efforts throughout most of our PhDs; MAMBO is all the better for it. Guillermo has been working hard on porting MAMBO to AArch64. He has made great progress and I am looking forward to putting the finishing touches together. Guillermo has also provided feedback on this thesis.

John Mawer has my gratitude for being the first user of MAMBO and providing valuable feedback and bug reports.

Last but not least, I will thank Amanieu, Mireya Paredes and Yaman Cakmakci for their support while we were writing up our theses at the same time.

This brings us to the other residents of IT301, who deserve our gratitude for putting up with four of their colleagues being stressed and keeping strange hours over the past few months.

Chapter 1

Introduction

1.1 Dynamic binary modification

1.1.1 General principles

Dynamic Binary Modification (DBM) is a technique for modifying applications transparently while they are executed, working at the level of machine code. A system implementing DBM is called a *DBM tool* or simply a *DBM system*. To be able to modify the code at runtime, a DBM system must maintain control of execution by scanning and, where required, translating all code before execution. To perform a given task (e.g. to optimise or to instrument the code), additional (*optional*) modifications are performed. A *DBM framework* is a DBM system which exposes its functionality through an external API, enabling an external tool to apply optional code modifications. For example, DynamoRIO [Bru04], Pin [LCM+05] and Valgrind [NS07b] are DBM frameworks, while Dynamo [BDB00] and QEMU [Bel05] are DBM systems which are not DBM frameworks.

It shall be noted that since a DBM system deals with machine code, a large part of the implementation of such a system is architecture-specific. Therefore, an optimised DBM implementation is not easily portable to a different architecture.

DBM can be used both at the system level and at the userspace level. The main differences between the two consist of isolation and security challenges in the case of a system level DBM (while for userspace DBM systems these are mostly handled by the operating system) and handling of irregular changes to the execution flow: for userspace DBM these consist of signals (on Linux), while system level DBM has to handle hardware interrupts, exceptions and faults. This

thesis explicitly deals with a userspace level DBM system. However, the proposed optimisations for branch handling are generally suitable for use in a system level DBM as well.

1.1.2 Uses

DBM is a general purpose technique with numerous applications. Most uses of DBM can be classified in one of three categories: Dynamic Binary Instrumentation (DBI), Dynamic Binary Translation (DBT) and Dynamic Binary Optimisation (DBO).

DBI is a technique for dynamically adding instrumentation code. DBI is often used for fine-grained analysis of applications, since it allows data to be collected up to the level of individual instructions and data bits. One example of DBI is Memcheck [NS07a], a tool which allows sources of memory errors to be tracked at the level of individual bits and which was implemented using Valgrind [NS07b], a DBM framework.

DBT is a technique for efficiently translating software at runtime from one Instruction Set Architecture (ISA) to another one. At the system level, DBT has been used by Transmeta to run x86 software on their processors, which implemented a proprietary VLIW architecture [DGB+03]. DBT is also commonly used by Virtual Machine Monitors, for example QEMU [Bel05] and those developed by VMware [AA06].

DBO is a technique which uses runtime information to optimise machine code. For example, DynamoRIO has been used to show that even programs compiled with a high level of static optimisation can benefit from dynamic application of traditional optimisations [Bru04]. Additionally, microarchitectural optimisations have been demonstrated [Bru04]. Furthermore, the NVIDIA Denver processor is employing both DBT to execute ARM applications on a proprietary VLIW architecture and also DBO to optimise the translated code for the VLIW processor [BBTV15].

1.1.3 Overhead

A general limitation of DBM is that it introduces various overheads compared to native execution, even if no optional transformations are performed. Such overheads include increasing the memory usage due to the data used by the

DBM system and increasing the start-up time by having to inspect and patch the code before executing it. However, the most important overhead is the one affecting execution speed, even after the start-up phase. This overhead varies significantly between DBM systems, workloads and hardware architectures. The average overhead on the SPEC CPU benchmark suite can be as low as 10% for performance-optimised state-of-the-art DBM systems on the x86 architecture and as high as 1900% for less optimised systems.

Depending on the magnitude of the execution speed overhead, many of the uses described in Section 1.1.2 can become limited or even impractical. For example, if the performance of a DBT system is noticeably poor to its users, then the technology is unlikely to be deployed at all. Similarly, if a profiling or debugging tool using DBI has high overhead, it might become impossible to employ it in some cases (e.g. if the intention is to analyse a real-time video game which ends up running at a very low frame rate under the tool). Finally, it only makes sense to create a DBO tool when the overhead of the underlying DBM framework is low enough to be easily amortised.

The execution speed overhead is caused by multiple factors, however it is essential to note that the main source of overhead is the lower execution speed of the code after being patched by the DBM system and not the patching process itself. The reasons for the lower execution speed include: the additional instructions inserted for handling branches, in particular indirect branches, and those used to hide the DBM system from the application; also microarchitectural causes such as lower cache hit rates due to the increased code and data sizes and poor branch prediction rates in the modified code. Section 2.5 further discusses the causes of performance overhead identified in the literature.

1.1.4 DBM for ARM

The recent popularity of the ARM architecture, especially in the mobile and embedded markets, has created a demand for compatible DBM systems. However, both the literature and the existing DBM systems have almost exclusively focused on support for other architectures, especially x86. Consequently, ARM is supported by only two DBM / DBT systems (Valgrind [NS07b] and QEMU [Bel05]), neither of which is designed to achieve low overhead. Previously, an ARM port of Intel Pin has also been available [HK06], however 1) it has since been discontinued and 2) its overhead was around one order of magnitude higher compared

to state of the art DBM systems for x86. The lack of publications and expertise on optimising DBM for the ARM architecture, together with the demand for such tools, both from academia and from industry, presented an interesting opportunity.

1.2 The ARM architecture

1.2.1 Emergence

Development of fast, low power systems led to the wide adoption of powerful mobile devices in the form of smartphones starting from the late 2000s. This created the current situation of the consumer mobile device market dwarfing the traditional computer market and the tentative adoption of low power architectures for less traditional applications such as datacentres. These platforms have reached similar complexity with that of more performance-oriented architectures. Most recent developments aim to further improve power efficiency and performance using same-ISA heterogeneous multicore chips, such as ARM big.LITTLE, which combines a cluster of ISA-compatible low power processors and a cluster of high performance processors on the same chip [ARM13b].

1.2.2 The ARM hardware ecosystem

Multiple vendors are developing and commercialising ARM-based SoCs for different markets, such as smartphones and tablets, network appliances, embedded applications and network servers. Since these markets have different requirements, a wide range of ARM implementations are commercially available at the same time. There are two paths through which commercial ARM implementations are developed. First, an implementation can be licensed from ARM, which allows the vendor to tweak some microarchitectural parameters such as the size of caches and TLBs, but does not allow for fundamental changes to the microarchitecture. Alternatively, the vendor can develop their own implementation, which has to correctly support the ARM instruction set, however this allows complete freedom in designing the microarchitecture. Several vendors have chosen the second approach, further increasing hardware fragmentation.

Currently available implementations vary from single issue, in-order systems (e.g. ARM Cortex-A5) to high performance 6-issue out-of-order cores (e.g. Apple

Twister) or cores designed for large multicore systems (e.g. APM X-Gene2, a 4-issue out-of-order microarchitecture which can be used in clusters of 8 to 16 cores [SFY14]). Additionally, same-ISA heterogeneous multicores have been widely adopted.

The work presented in this thesis is applicable to the *Application profile* of the ARM architecture, i.e. ARMv7-A and ARMv8-A, which is used for general purpose computing. The ARM architecture also has a *Microcontroller profile* and a *Realtime profile*, which are used for microcontrollers and for hard real-time applications respectively. The latter two profiles are not within the scope of this document. Furthermore, ARMv8-A has introduced a 64-bit execution state called *AArch64*, while also providing a 32-bit state called *AArch32*, which is backwards-compatible with ARMv7-A. The techniques presented in the following chapters have been implemented for the 32-bit state only, due to the late availability of ARMv8 hardware and time limitations. Therefore, in the context of this document, the generic term *ARM* will be used to refer to the 32-bit execution state of ARM. Nevertheless, many of the optimisations presented in this thesis are expected to also apply to AArch64, with little or no modification. The compatibility of the optimisations with AArch64 is summarised in Section 6.3. The ARM ISA and the specific challenges it poses to DBM are discussed in Section 3.3.

1.3 Motivation

The ARM architecture, once almost exclusively found in embedded systems, is growing in adoption for general purpose computing, however most DBM systems and research have focused on the x86 architecture. This has resulted in the performance of DBM systems for ARM lagging behind; cf. Pin [LCM+05] or DynamoRIO [Bru04] on x86/x86-64. For example, Valgrind serialises multithreaded execution on ARM and x86/x86-64, while performance optimised DBM systems such as Pin and DynamoRIO do not. This raises two of the questions that this thesis addresses: (i) how to develop such DBM systems for the ARM architecture and (ii) whether new optimisations are plausible and needed.

Additionally, the ARM ecosystem poses unique challenges for high performance DBM systems because of the large number and wide range of capabilities of the commercially available implementations: from single issue, in order cores (Cortex-A5), up to 6-issue out-of-order cores (Apple Twister) and including less

traditional implementations such as NVIDIA Denver (a 7-way VLIW processor using a combination of hardware and software DBT to run ARM code). These challenges are exacerbated by the wide adoption of single-ISA heterogeneous multicores (such as *big.LITTLE* [ARM13b]), which use different microarchitectures on the same System on Chip (SoC) and allow the migration of active applications from one type of core to another. This raises the question of whether it is possible to develop DBM optimisations which either improve or, at the very least, do not affect performance on all available systems and microarchitectures. Furthermore, are the results obtained by evaluating a DBM system on one or a small selection of systems relevant across the available platforms?

1.4 Contributions

The contributions of this thesis are:

- developing a research DBM system, to enable optimising the key code manipulation algorithms for ARM 32-bit platforms; this system has been named MAMBO and is presented in Chapter 3;
- proposing the principle of *behavioural transparency* to relax the transparency requirements for DBM systems in order to minimise implementation complexity and improve performance (Section 3.11);
- comparing the performance of MAMBO against two other DBM systems, Valgrind and QEMU (Section 4.4.4);
- showing that due to the wide range of ARM microarchitectures commercially available, runtime selection of optimisations is desirable to achieve optimum performance (Section 5.4);
- microarchitectural optimisations for DBM, which take advantage of certain features of the hardware implementations:
  - introducing a hot code profiling technique for the creation of *traces* while avoiding branch target prediction for poorly predictable branches or expensive modification of existing translated code (Section 5.2);
  - introducing a novel technique to enable hardware return address prediction in a code cache without a software return address stack (Section 5.3.1);

  - introducing a software indirect branch prediction scheme which allows effective prediction for polymorphic indirect branches (Section 5.3.3);
  - introducing a technique aimed at reducing the data cache and TLB footprint of the inline hash table lookup routine used in the translation of indirect branches on ARM (Section 5.3.2);
  - introducing the use of huge pages to reduce the TLB pressure created by running modified code from a software code cache (Section 5.3.4);

- architectural optimisations for DBM, which mainly aim to reduce the dynamic instruction count:
  - introducing a novel return address prediction scheme which trades off guaranteed transparency for improved performance (Section 4.2.1);
  - introducing an improvement of the fastBT [PG10] table branch linking scheme, which is implemented more efficiently on ARM and reduces the memory space requirements on all architectures (Section 4.2.2);
  - describing how to implement and configure several other established DBM optimisations for ARM (Sections 4.2.3, 4.2.4 and 4.3); and
- evaluating and providing a detailed analysis of the effectiveness of these optimisations when running on a wide range of microarchitectures (Sections 4.4 and 5.4).

1.5 Publications

The material from Chapters 3 and 4 appears in the following journal publication:

- MAMBO: A Low-Overhead Dynamic Binary Modification Tool for ARM. **Cosmin Gorgovan**, Amanieu d'Antras and Mikel Luján. In *ACM Transactions on Architecture and Code Optimization (TACO)*, 13 (1), 14, April 2016.

The material from Chapter 5 is based on the following submission-pending paper:

- Optimising Dynamic Binary Modification Across ARM Microarchitectures. **Cosmin Gorgovan**, Amanieu d'Antras and Mikel Luján.

Other related publications:

- Optimizing Indirect Branches in Dynamic Binary Translators. Amanieu d'Antras, **Cosmin Gorgovan**, Jim Garside and Mikel Luján. In *ACM Transactions on Architecture and Code Optimization (TACO)*, 13 (1), 7, April 2016.

1.6 Thesis structure

Chapter 2 presents the organisation and components of a DBM system and the interactions between them. The chapter then continues with a literature review of the existing optimisations for these components, in particular for the *code cache* and for *branch linking*. The final section of the *Background* chapter discusses the performance overhead of state-of-the-art DBM systems. Chapter 3 then presents an overview of the MAMBO system. This chapter defines the aims of the system and then presents the design decisions which were taken towards those aims, in the context of the larger design space presented in Chapter 2.

Chapter 4 starts presenting the novel contributions of this thesis related to branch linking, which is one of the major sources of overhead for DBM systems. Existing linking techniques are also discussed in the context of efficiently implementing them for the ARM architecture. The chapter includes an evaluation, conducted on two commercial ARM systems, using two different microarchitectures. The evaluation is done both for each individual optimisation separately and also for the overall system performance, in comparison to the *Valgrind* and *QEMU* systems. At this point, the overhead of MAMBO is significantly lower than the overhead of the other two systems. This is considered the *baseline* version of MAMBO, which achieves lower overhead than the other DBM systems available for ARM, but slightly higher than the state-of-the-art DBM systems. Chapter 5 then continues presenting novel optimisations for reducing this overhead, however the focus is shifted from optimising at the architectural level (i.e. implementing operations using a low number of instructions) to optimising for the underlying microarchitectures. For example, the techniques presented in this chapter prioritise efficient use of the memory subsystem and good branch prediction over a low dynamic instruction count. Due to the nature of these optimisations, their evaluation is done on five different microarchitectures with a wide range of capabilities. In addition to the execution time, the hardware performance counters

are used to measure the effect of each optimisation on the microarchitecture, by counting events such as cache, branch prediction and TLB misses.

Chapter 6 summarises the contents of this thesis and the conclusions which have been drawn. Based on these conclusions and the various behaviours observed in the evaluation of the existing system, a number of avenues for future optimisation are suggested. Furthermore, a possible approach towards automatic optimisation of instrumentation code inserted via the plugin API of MAMBO is proposed.

Table 1.1 summarises the structure of this thesis.

Chapter 1	Introduction, motivation, contributions and publications.
Chapter 2	Background and related work.
Chapter 3	Overview of MAMBO, the DBM platform.
Chapter 4	Linking techniques for indirect and direct branches.
Chapter 5	Microarchitectural optimisations, including traces.
Chapter 6	Conclusions and future work.

Table 1.1: Structure of the thesis.

Chapter 2

Background and related work

2.1 Binary modification

Binary modification is a technique for modifying computer programs at the level of machine code. It is used either when 1) the source code or toolchain required to re-build the code are not available, or 2) when it is desired to preserve parts of the machine code intact. The first case is fairly clear: without the ability to access and modify the source code and then to rebuild the program from the updated source code, the only alternative is to modify the native code. However, the second case deserves further explanation. Let us take the example of instrumentation being used to obtain code execution traces, for the purpose of architectural simulation. This could be implemented by modifying the source code by adding an instrumentation procedure and a number of calls to it. However, the additional instrumentation code would change the original layout of the executable and therefore affect the addresses recorded in the execution trace. Furthermore, the compiler and linker will observe and optimise the instrumented code, possibly changing the generated machine code even more in terms of control flow, generated instructions and their scheduling. In practice, an application instrumented by modifying its source code will provide an insight into its own behaviour rather than into the behaviour of the original, uninstrumented application. This is often undesirable, including in this example. On the other hand, at the level of native code, it is practical to observe the original code, and then insert instrumentation with minimal disturbance.

2.1.1 Static and dynamic binary modification

There are two approaches to binary modification: static binary modification and dynamic binary modification. Static binary modification is altering the program ahead of its execution. Its main advantage is that it can have minimal or no overhead compared to the original executable. However, it is not universally applicable because of the *code discovery problem* [HM80]: for non-trivial programs, it is impossible to accurately determine which locations contain code and which locations contain data ahead of time, therefore it will suffer from incomplete coverage, when code is misidentified as data and incorrect modification, when data is misidentified as code. Additionally, static binary modification cannot be used for Just-In-Time-compiled code and self-modifying code.

Dynamic binary modification avoids the code discovery problem by identifying the code just before it executes: instead of statically scanning and modifying the whole executable, this is done in single-entry and single-exit units called *basic blocks*. The first basic block is scanned at the entry point of the program, stopping when the first control flow instruction is encountered. Once the basic block has finished executing, the target address of the control flow instruction is used to scan the second basic block, and so forth. This approach may not have complete coverage of all static code regions, however it will capture all code that is executed and will not misidentify code and data. Furthermore, it is compatible with Just-In-Time (JIT) compilation and self-modifying code. Therefore, DBM is a universal solution to the binary modification problem. However, this process introduces a number of overheads, which can have a high impact on performance. Avoiding and amortising these overheads is the aim of DBM performance research.

2.2 Transparency

A key concept related to DBM is *transparency*, which refers to preventing the code being modified from observing any changes compared to native execution. There are multiple *types* of transparency (i.e. what kind of changes can be observed) and also various *degrees* of transparency (i.e. how can the changes be observed and how likely are they to interfere with correct execution).

Bruening [Bru04] classifies the types of transparency in three categories, depending on how they are handled by DynamoRIO. The first category includes the

resource usage conflicts (e.g. library transparency, I/O transparency), which are handled by avoiding to share resources between the application and DynamoRIO. The second category includes the types of transparency which can be provided by *leaving the application unchanged* (e.g. thread transparency, data transparency). The third category includes the types of transparency which require specific handling by DynamoRIO (e.g. application address transparency, debugging transparency).

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

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

2.3 The implementation of dynamic binary modification

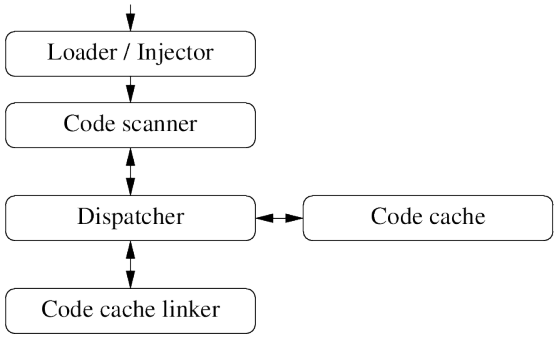


Figure 2.1: The components of a DBM system and the control flow between them.

Figure 2.1 shows the major components of a DBM system. The entry point is the mechanism which allows the DBM system to take control over the execution

of an application (the *Loader / Injector*). There are two major approaches to doing so: either launching the DBM system first and then *loading* the application or, alternatively, *injecting* a DBM system into an already running process. These are further explored in Section 2.3.1.

Once the DBM system has control of a process, it must inspect and modify the code of the application. This is done by the *code scanner*, which is further discussed in Section 2.3.2. The modified code produced by the code scanner is then typically stored in a software *code cache* (Section 2.3.3), from where it can execute multiple times to amortise the overhead of the code scanner. To efficiently transfer control between different basic blocks, they are linked together by the *code cache linker* (Section 2.4). Because code is dynamically discovered and then scanned at runtime, execution of the modified code, of the code scanner and of the linker is interleaved. Control transfers between the code cache and the other system components is managed by a *dispatcher*, which preserves the context of the application on exit from the code cache and then restores it back when returning.

2.3.1 Tool injection / application loading

DBM systems generally run in the same thread and memory space as the application to be modified. It follows that before the DBM system can start executing the application, both executables (the DBM system itself and the application) have to be loaded in the memory space of the process. This can be achieved using one of two approaches: **Injection**, when the application is launched in the regular way, using the system loader and dynamic linker. After loading has completed, the DBM system injects itself into the same memory space. Injection has been implemented mainly using two mechanisms: *LD_PRELOAD*, which is a feature of the dynamic loader on GNU/Linux systems which forces the loading of specific shared object files in a process. A DBM system can inject itself by using this feature, and then it can take control of execution by declaring a *constructor* function, which the dynamic linker calls before the *main* function of the application. This approach is simple to implement because it delegates loading to the system software. However, this method is fundamentally incompatible with statically compiled applications. It can also allow some application code to run unmanaged (e.g. constructors from other loaded libraries) and it is not portable to other platforms. This is the default approach taken by DynamoRIO [Bru04].

A second mechanism which can be used for injection is *ptrace*. Ptrace is a Unix API which allows one process to control a second process. A DBM tool can use ptrace to pause the execution of the application, inject code which loads the DBM system in the same process, and then resume execution. After a successful injection, the DBM tool has to restore the memory contents of the application and start managed execution. This approach is difficult to implement reliably because the tracing process has limited control over the context of the traced application, requiring the DBM tool to overwrite existing machine code in the application, only with limited information being available to it. Additionally, ptrace functionality is incomplete on many combinations of hardware platforms and operating systems, reducing portability. The advantage of this approach is that it can be used to inject a DBM system into a running process. This is an alternative injection implementation used by DynamoRIO [Bru04].

The alternative to injection is **Loading**, when the DBM system is started first, using the standard launching mechanism of the platform. The DBM system then loads the application itself. Because the executable loader used on GNU/Linux is implemented in the kernel, a different loader, running in the userspace, has to be included with the DBM system. This approach allows the DBM tool to maintain control of the execution at all times. In addition, it is more easily portable to different platforms. The current implementation of Valgrind uses this loading strategy, which has been found to be more reliable than injection [NS07b].

2.3.2 Code scanners

DBM systems work by scanning and, if required, modifying code before it executes. This is done by code scanners, which disassemble the native machine code, inspect it and then assemble any needed modifications. Multiple approaches have been used to build code scanners. The main differentiating criteria is instruction representation: *Copy-and-Annotate* (C&A) code scanners essentially copy most code unmodified, while only translating a relatively small number of instructions, such as sensitive and control flow instructions. This approach is well suited for DBI and DBO applications, which tend to only modify a small subset of the input code. C&A is generally used by performance-oriented DBM systems, such as DynamoRIO [Bru04], Pin [LCM+05] and FastBT [PG10].

Disassemble-and-Resynthesise (D&R) code scanners, on the other hand, translate all input code into an Intermediate Representation (IR), then perform any

required modifications at the IR level and finally translate the IR back to native code. This approach is well suited for portable DBM systems because it allows the modification logic and any architecture-independent optimisations to be implemented once and reused across multiple architectures. Similarly, this approach is also well suited for DBT applications, which have to translate all instructions to a different ISA. D&R can also be useful for performing complex analysis by using a simplified IR to reduce the challenge of writing instrumentation code. D&R enables the JIT compiler to optimise across the original and the modified code, which can also reduce the challenge of developing complex instrumentation [NS07b]. However, it is difficult to maintain the performance of the original input code across the native-to-IR and IR-to-native translations. In particular, it is difficult to represent all properties of the input native code in a simple IR, which typically results in the IR representation using significantly more instructions than the input. To control this overhead, an architecture-specific optimisation pass is usually required when the IR is translated back to native code. Furthermore, the scanning process itself generally has higher overhead compared to C&A because of the additional operations involved. D&R is used by DBM systems such as Valgrind [NS07b] and QEMU [Bel05].

Apart from instruction representation, another design decision is whether and how much of the code scanner to craft manually and how much to automate. FastBT, for example, uses a high level representation of the instruction set to automatically generate translation tables [PG10]. The logic for modifying specific instructions is then implemented using the C++ language. This allows FastBT to automate the error prone aspects of the code scanner, while also allowing low level control for modifying and translating instructions. On the other hand, DynamoRIO mostly uses hand-crafted C-language implementations for instruction decoding, instruction encoding and modification / translation logic. This allows experienced implementers the maximum degree of flexibility and opportunities for optimisation.

2.3.3 Code caches

Scanning and translating any given basic block takes significantly longer than executing the translated code in the basic block once, therefore it is essential for a DBM system to be able to reuse its translations, especially those of the *hot code* (the regions in which most of the execution time is spent). This can

be achieved in two ways: the first one is to modify the original code in-place. For example, to insert instrumentation code at a specific location, the original instruction at that location would be replaced by a trampoline. This trampoline would branch to a different memory region, where the instrumentation code, the original instruction and a branch back to the instruction following the trampoline have been injected. This approach is taken by Dyninst [BH00]. It has the advantage that a performance penalty is only paid when the original code is modified. However, it has a number of shortcomings related to code discovery and transparency [HMC94, BM11]. Additionally, the overhead of the modified code is relatively high since it cannot be inlined, or alternatively, it involves additional complexity in relocating large blocks of code to allow inlining [BM11].

The other approach, used by most DBM systems [Bru04, LCM+05, PG10, Bel05, SN05, BDB00], is to relocate all code to a *software code cache* and to link the code cache *fragments* directly, a technique introduced by Shade (under the name *chaining*), an instruction set simulator [CK94]. The mapping from the original application addresses (the Source Program Counters - SPCs) to their translations in the code cache (the Translated Program Counters - TPCs) is maintained using a hash table indexed by the SPCs. This approach has a number of advantages, namely: it avoids the code discovery problem by identifying the reachable code at runtime; it avoids transparency issues by preserving both the original code and data unmodified at their original locations; and it simplifies inline modification of code by combining the relocation and modification in one step [Bru04]. The disadvantage of this approach is that an overhead is introduced for all code due to its relocation to the code cache, even if it is not otherwise modified. However, various code cache linking optimisations have been used to minimise this overhead, while retaining the advantages of the code cache. The overhead of a number of DBM systems is further discussed in Section 2.5.

Basic blocks

A DBM software code cache is organised in *basic blocks*, which are single-entry and single-exit fragments, meaning they are created from linear code regions, with the first instruction being the only allowed entry point. Consequently, different entry points to the same linear code region in the application will create multiple basic blocks in the code cache, with tail duplication. The source of each block can only contain up to one control flow instruction, which must be the

last instruction. Other conditions such as reaching a size limit or encountering a sensitive instruction can be used to terminate a basic block, in which case they are equivalent to a basic block which unconditionally branches to the following instruction in memory.

Traces

Traces (also known as superblocks or extended basic blocks) are single-entry, multiple-exit fragments, usually a result of merging together multiple basic blocks which are reachable from a specific entry point. Traces eliminate branching on a preferred execution path and improve cache locality, thereby reducing the overhead of modified code.

Traces are usually built using hot path profiling, however they can also be created using static heuristics, in which case they are known as pseudo-traces. To be effective, online hot path profiling requires low overhead and must produce a result after a very limited number of executions, therefore offline profiling techniques are not generally considered suitable [DB00]. *Next Executing Tail* (NET) [DB00], a lightweight profiling scheme originally developed for the Dynamo dynamic optimiser [BDB00] has been widely adopted. NET works by inserting an execution counter into the basic blocks which are a target of backward branches, known as *path heads*, based on the insight that hot code regions consist of loops and the targets of backward branches are the start of loops. Once this counter has reached a certain threshold, a new trace is created with the path head as the entry point. The taken target at the end of each fragment in the trace is then appended to the trace, until another backward branch or the entry point to an existing trace is executed. When conditional branches are encountered, the selected target is added to the trace and the alternate target is conditionally linked, creating trace exits. NET is described in more detail in Section 5.2.

Last-Executed Iteration (LEI) has been proposed as an alternative to NET [HHS05]. It improves on NET by placing interprocedural cycles inside a single trace and by reducing duplication in the case of nested loops and unbiased conditional branches. However, its implementation introduced high profiling overhead, which overshadowed the expected performance benefits [DH11a].

NETPlus [DH11a] is another alternative to NET. It identifies the hot code regions in the same manner as NET, by inserting a dynamic execution counter in the targets of backward branches. Its improvement over NET is in the trace

termination condition: when NETPlus encounters a backward branch, it first performs a forward search for loops back to the trace head and then only terminates the trace if a loop is not found. This allows NETPlus to create fewer, longer traces compared to NET, further improving code locality. NETPlus was shown to reduce the number of transfers of control directly from one trace to another compared to NET on the SPEC CINT2006 benchmarks [DH11a].

2.4 Branch linking

Optimising the translation of branches in the code cache is essential to achieve low overhead. This is done by linking the code cache fragments directly, instead of going through the dispatcher to transfer control. Branches can be classified in *direct branches*, which have a static target, and *indirect branches*, which have a dynamic target. Due to this difference, the two types of branches are handled separately.

2.4.1 Direct branches

Because the target of direct branches is known at the time they are scanned, their handling is relatively straightforward: if a translation of the target is available in the code cache, then it can be *linked* directly by placing a branch at the end of the code cache fragment. The design space of direct branch linking is relatively small. One decision is whether to *proactively* link direct branches when the fragment is created or to *lazily* link them when and if they are taken. Bruening [Bru04] has found that for DynamoRIO on x86, proactive linking is generally more efficient and improves the overall performance for *mesa*, a SPEC CPU2000 benchmark, by 5% compared to lazy linking. The performance of some benchmarks is slightly reduced by proactive linking, but only by up to around 1.5% for *swim* and under 1% for other benchmarks. The overall effect is relatively small, with a harmonic mean improvement of under 1% across the SPEC CPU2000 benchmarks and a selection of desktop applications. Hazelwood and Klauser [HK06] also found proactive linking to be more efficient for the ARM implementation of Pin.

A second design choice is whether to terminate basic blocks on unconditional direct branches or to elide the branch and inline the target of the branch in the same basic block. Terminating minimises tail duplication, while inlining eliminates a control transfer and can increase code cache locality. Nethercote and

Seward [NS07b] argue that inlining, together with an efficient dispatcher implementation, enables Valgrind to achieve relatively low overhead despite not linking branches. DynamoRIO also elides unconditional direct branches, although the main benefit is in reducing the memory usage rather than directly reducing overhead [Bru04]. HDTrans elides unconditional branches, but only for targets which do not already have a translation in the code cache, to reduce code duplication [SSNB06].

2.4.2 Indirect branches

Because indirect branches have a dynamic target, they cannot be directly linked like direct branches. Instead, the original target address used by the application (the SPC) must be matched to the address of its translation in the code cache (the TPC) every time the indirect branch executes. The translation of indirect branches is the main source of overhead for DBM systems and an efficient implementation is critical for achieving good performance [KS03b]. Various solutions have been proposed, which can be broadly classified in two categories. The first category, hardware and hardware/software co-designed solutions, modify the architecture or microarchitecture of processors. The second category are the software-only solutions, which are compatible with unmodified general purpose architectures.

Before exploring the indirect branch linking techniques in the literature, it is necessary to further classify indirect branches into several types with distinct properties. The first category of indirect branches is *returns*, which execute at the end of a procedure (the callee) to return control back to the caller. More specifically, returns branch to the instruction immediately following the call instruction in the caller. This enables very accurate target address prediction, simply by recording the address of the instruction following each executed call in a Last In, First Out (LIFO) structure and then removing one entry, the predicted address, for each executed return. However, prediction is not always correct, for example due to exceptions and therefore must always be checked for correctness. A second category of indirect branches is *table branches*, which load their target address from an indexed table in memory. The third category of indirect branches is *generic indirect branches*, which simply branch to a dynamic target address from a register or from memory.

Returns

Returns make up the majority of executed indirect branches [SKC+04]. Most microarchitectures incorporate a Return Address Stack (RAS) which is automatically maintained by the hardware across calls and returns, and which is used for return address prediction when code is executing natively. However, the naive translation of calls and returns in the code cache uses generic branch instructions, which do not take advantage of this hardware branch prediction mechanism. Kim and Smith have found this to be a limiting factor in the performance of DBM systems [KS03a].

One optimisation which can be considered is to use code cache addresses (i.e. TPCs) instead of the original addresses (SPCs) as return addresses, by copying call and return instructions unmodified to the code cache. This would enable hardware return address prediction in the code cache, at the cost of application transparency. However, as reported by Bruening [Bru04], many applications are affected by breaking transparency in this way, therefore making this optimisation impractical. To avoid the transparency issues, another possible optimisation is to maintain a software RAS together with the hardware RAS. The software RAS consists of pairs of SPCs and their matching TPCs. An entry is pushed by the translation of call instructions, while translated return instructions pop the first entry from the software RAS, compare the actual target SPC of the application to the SPC of the RAS entry and if the two match, then execution can branch directly to the TPC in the RAS entry. This optimisation was used for return address prediction in FX!32 [HH97] and Pin for ARM [HK06]. However, an experimental implementation in DynamoRIO was found to be slower than the baseline indirect branch translation using an optimised hash table lookup and indirect branch [Bru04].

A different approach, in the form of *function cloning* [CHK93], is taken by Pin [LCM+05]. Pin creates multiple copies of procedures in the code cache, one for each call site. This allows each copy to statically predict a single return address, which is expected to be correct in the common case. However, this can result in large amounts of code duplication and increases the pressure on the instruction memory subsystem.

HDTrans introduced a technique called the *return cache* [SSB07], which is also used by FastBT [PG10]. Similarly to the software RAS technique, it is exploiting the relationship between calls and returns to predict the target address

of returns. The return cache is a thread-private hash table storing TPCs indexed by their SPC. The translated calls add an entry to the return cache for the instruction following the call. The translated returns then *blindly* load the hash table entry indexed by the SPC of the return target. Collisions are handled in the translated return fragment, which compares the return target SPC with its own source SPC and falls back to the generic SPC-to-TPC lookup if a mismatch is detected. This technique can reuse the return cache entries in case of deep recursion, as opposed to the software RAS predictor which stores a SPC-TPC pair for every call. It also requires use of fewer registers compared to the software RAS. However, the performance of the return cache is severely affected by collisions, which are reported to happen at a rate of 15% to 22% [SSNB06].

Table branches

FastBT introduced an optimisation for the translation of table branches called the *shadow jump table*. This optimisation detects table branches on the x86 architecture by pattern matching specific encodings of the *jmp* instruction, which load the target address from a table using an absolute or PC-relative base address and an offset. When a table branch is detected, FastBT creates a shadow jump table, which is populated on demand with the TPCs of the SPCs in the original table. The performance overhead of the DBM system is reduced by translating a table branch in the application to a table branch in the code cache, as opposed to one of the less efficient lookup implementations for generic indirect branches. However, the size of the jump table is not statically determined, therefore runtime bounds checking is used for the shadow branch table, introducing a performance overhead. Furthermore, a memory overhead is introduced by the shadow jump table.

Generic indirect branches

The most common way to translate generic indirect branches is by performing a hash table lookup in a hash table mapping SPCs to TPCs. Many DBM implementations use additional optimisations to avoid a hash table lookup in some cases, however it is used as the fallback technique by most DBM systems. The design space is fairly large and many different implementations have been developed. For example, one choice is whether to call a shared hash table lookup routine, minimising the code cache size, or to generate a separate, inlined, hash

table lookup routine in each translation of an indirect branch, reducing the length of the critical path [KS03b]. For example, DynamoRIO uses an inlined routine in the hot code (traces) and a shared routine in the cold code (basic blocks) [Bru04]; FastBT always uses an inlined routine [PG10], while QEMU uses a shared routine [Bel05]. Another decision is whether to perform the lookup in the global hash table or whether to use a small, local hash table for each translated indirect branch [LCM+05, HK06]. Other decisions include the size of the hash table, whether it has a static size or is dynamically resized and whether to use collision chains or open addressing with linear probing (the latter has been found to outperform the former in DynamoRIO due to better cache locality [Bru04]).

A common optimisation is Indirect Branch Inlining (IBI) (also called *indirect branch prediction* or *speculative chaining*), which caches one or multiple SPC-TPC pairs in the code cache fragment itself [WR96, BDB00, Bru04, LCM+05, HWH+07, PG10]. This optimisation relies on each indirect branch being heavily biased towards one or a very low number of targets. It is implemented by generating a chain of one or more compare-and-conditional-branch sequences which compare the target of the indirect branch against the cached SPC using only immediates. If the target matches the SPC, then execution branches directly to its TPC. Otherwise, execution falls through to the next compare-and-conditional-branch sequence or, if one is not available, to a fallback mechanism such as a hash table lookup. The cached targets are selected using different types of lightweight profiling: the optional DynamoRIO implementation records the targets of each execution up to a certain threshold count and then caches the hottest ones [Bru04], while the FastBT implementation replaces the cached targets on every prediction miss [PG10]. DBM systems which build traces across indirect branches, such as Dynamo [BDB00], implicitly use IBI with one cached target. In this case, the predicted target is simply selected as the target of the indirect branch at the time the trace path was recorded.

However, IBI is not effective if its prediction misses often, for example if the selected target changes during execution compared to the target which was profiled or if it regularly selects more unique targets than are cached [KS03b]. Furthermore, the cost of a prediction miss is high because time is spent comparing all cached SPCs and then also executing the fallback lookup procedure. The cost of mispredictions also limits the maximum number of targets which can be effectively cached, and it is mostly driven by the hardware branch mispredictions

generated in the compare-and-conditional-branch chain. Kim and Smith go as far as calling this technique a *performance limiter* [KS03b]. This limitation is also acknowledged by Payer and Gross [PG10], who have developed the *adaptive combined optimisation*, which adds a misprediction counter to IBI. When this counter reaches a certain threshold, IBI is replaced with an inline hash table lookup routine.

Dhanasekaran and Hazelwood [DH11b] have observed that indirect branches often have high locality (i.e. the selected target address for a given execution of the indirect branch is equal to the target address of the previous execution). To take advantage of this property, they have introduced a new entry at the beginning of the IBI chain, which caches the Most Recently Used (MRU) SPC and its TPC. Instead of implementing the target comparison on the code path, like standard IBI, this is implemented on the data path, which allows the prediction to be updated after every miss. All target fragments of the IBI chain are patched to update the predicted target when the MRU prediction misses. They have implemented this system in Pin, however no performance improvement was obtained. This is explained by the dynamic instruction count of MRU exceeding the dynamic instruction count of standard IBI, despite a higher prediction hit rate.

More recently, Jia et al. [JYHC14b] noted that the targets of indirect branches tend to exhibit phase behaviour and hypothesised that the accuracy of IBI predictions could be improved by detecting these phase changes and updating the predicted targets in response. They have proposed Software Prediction with Target Updating (SPTU), a collection of mechanisms which count the number of prediction misses at the level of predicted target (software-only), translated indirect branch (software-only), or application-wide (using the performance counters). Then, the related IBI prediction chain or chains can be updated when a certain miss count threshold is reached. An implementation for HDTrans of the *Global Miss count* version of SPTU, which maintains an application-wide miss count using performance counters, is shown to improve both the prediction rate of IBI and also the overall performance of HDTrans.

HDTrans introduced a hybrid technique which combines hash table lookups and IBI, called the *sieve* [SSNB06]. It works by calculating a hash value based on the target address of indirect branches, which is then used as the index for branching to the *sieve branch table*. Each entry in the sieve branch table contains

a branch either to a *sieve bucket* or to a fallback lookup routine. Sieve buckets compare the target against a cached SPC. If the target and the SPC match, then the sieve bucket restores the application context and branches to the cached TPC, otherwise control is transferred to the next sieve bucket, similarly to IBI. The sieve is essentially an implementation of a hash table lookup with open addressing which uses code instead of data. Therefore its performance relative to that of a regular data-based hash table lookup depends on the relative amounts of available code and data cache space [HWH+07].

SPc-Indexed REdirecting (SPIRE) does away with using a hash table lookup for SPC-to-TPC mapping and instead uses the whole memory space of the application as an indexed table [JYW+13]. SPIRE overwrites the targets of indirect branches with trampolines to their translation in the code cache. Therefore, indirect branches in the source application can be copied unmodified to the code cache. One challenge of this approach is in detecting the targets of indirect branches and inserting the trampolines before branching to these locations. This is solved by SPIRE by marking all memory mappings of the application as non-executable, therefore being able to trap execution at these locations. When the first trampoline is inserted in a page, SPIRE creates a shadow copy of the page for data transparency, then inserts the trampoline and fills the rest of the page with trap-generating instructions and marks the page as execute-only (i.e. no read and write access). This allows the trampoline to execute directly in the future, but traps branches to other locations in the page. Furthermore, attempts to access data on the page will trap, allowing SPIRE to substitute the shadow copy.

The base SPIRE design incurs high overhead when data in the modified pages is accessed. Additionally, it requires MMU support for specific combinations of read/write/execute permissions which are not supported either on x86, nor on ARM AArch32. Both of these issues are addressed by the authors by modifying SPIRE to maintain the trampolines in shadow pages at a fixed offset from the corresponding code, while leaving the pages of the application unmodified. However, this comes at a cost in terms of memory fragmentation and overhead. Furthermore, both approaches to SPIRE have low cache and TLB locality because of the additional level of indirection and fragmentation in the trampoline pages. Nevertheless, the evaluation shows that SPIRE can reduce the overhead of HDTrans on x86 and it can also outperform an implementation of IBI.

Jia et al. observed that the targets of many indirect branches are directly *selected* from memory rather than being *calculated* at runtime [JYHC14a]. Furthermore, the possible targets of these indirect branches are organised in tables, which are accessed using a index at runtime. *Direct-TPC-Table* (DTT) has been created to translate this type of branches. DTT works in two stages: first, the code of the application is inspected to detect these branches and their corresponding Target Address Tables (TATs). Then, a shadow table containing the TPC for each SPC entry in the TAT is created and a translation of the indirect branch which uses this table is generated. This translation of indirect branches can then execute efficiently by avoiding a runtime SPC-to-TPC lookup and instead executing an indirect branch using the TPC table. The main challenge of DTT is in identifying the base address and size of the TAT for each indirect branch. The authors propose a set of heuristics both for discovering the TATs at runtime and also for rolling back this prediction scheme when it is inefficient for a particular branch instruction (e.g. if the shadow TPC table grows too large or if the TAT is frequently modified, requiring multiple invalidations of the shadow TPC table).

DTT differs from the *shadow jump table* technique introduced by Payer and Gross [PG10] (discussed in the *Table branches* subsection above) in that it applies to a wider set of branches. Whereas the shadow jump table technique can only translate branches using an address table which can be identified at code scanning time (i.e. using an absolute or PC-relative base address), DTT is more widely applicable to branches which select their address from memory, even if the base address is dynamic, e.g. the value of a register. This allows DTT to handle additional indirect branch instructions, such as those used to implement virtual functions in higher level languages. However, *shadow jump tables* can be used to translate explicit table branches (as used for compiling switch statements) just as efficiently, using a simpler mechanism. Furthermore, the ARM architecture poses additional challenges to the implementation of DTT due to the limited range of immediate operands, requiring the insertion of additional instructions compared to the x86 implementation.

Hardware solutions

Hardware and hardware/software co-designed solutions can eliminate most of the performance overhead associated with indirect branch translation. Several techniques have been proposed, both for handling generic indirect branches and

also for returns. However, none of the proposed techniques have been implemented on general purpose architectures, therefore they are considered outside the scope of this thesis and will not be discussed in detail. The Jump Target-address Lookup Table (JTLT) [KS03b] is a hardware cache which maps SPCs to TPCs, allowing the translation of source indirect branches to a specialised type of indirect branches which take the SPC as an operand and automatically branch to the TPC. The main disadvantage of this approach is in the amount of associative memory required on-chip. The JTLT is suitable for the translation of table and generic indirect branches. For returns, the *dual-address return address stack* (dual-address RAS) [KS03b] modifies the hardware RAS predictor used by most processors to store SPC-TPC pairs, similarly to the software RAS prediction scheme discussed on page 35. This would allow the translation of source calls to a *push-dual-address-RAS* instruction taking as input the SPC and TPC of the expected return. The hardware return address predictor would then use the TPC of the RAS entry as the predicted target and would compare the SPC of the RAS entry against the SPC operand of the translated return instruction.

2.5 Performance overhead

The overhead of DBM systems has been an active area of research. Depending on the design aims and choices of such systems, the overhead varies significantly. Bruening et al. [BZA12] evaluated the performance of DynamoRIO [Bru04] and Pin [LCM+05] when running SPEC CPU2006 compiled for x86-64, with no instrumentation. They have reported an average overhead of 11% for DynamoRIO and 21% for Pin, with a maximum overhead under 60% for DynamoRIO and under 80% for Pin. In a recent DynamoRIO tutorial [BZ16], a harmonic mean overhead of 8% for x86 and 13% for x86-64 has been reported on SPEC CPU2006. The reported average overhead for StarDBT [WHK+07] running SPEC CPU2000 compiled for x86 is 12% and 27% on two Intel Xeon processors using different microarchitectures. Also on x86, the overhead of HDTrans has been reported to compare favourably to DynamoRIO, however the exact number is not specified [SSNB06]. Payer and Gross [PG10] have evaluated DynamoRIO, FastBT, HDTrans and Pin on x86. Using the reported execution times, the geometric mean overhead of each the four systems has been calculated to be 7%, 9%, 10% and 34%, respectively.

On the ARM architecture, the only DBM system with a focus on performance was the Pin port [HK06], which has since been discontinued. Its performance has been evaluated by Hazelwood and Klauser on a subset of the SPEC CPU2000 benchmarks, using a reduced input data set due to limitations of the test platform (using an ARMv4 processor clocked at 200 MHz, with 64 MiB of memory). Compared to native execution, its geometric mean overhead with no instrumentation was 187%, with a maximum of 656%. Copies of Pin for ARM are no longer distributed and therefore it could not be evaluated in this thesis. Furthermore, Pin for ARM appears to have only supported the ARMv5 ISA, while the evaluations in this thesis were done using benchmarks compiled for ARMv7.

DBM on the ARM architecture is currently also supported by Valgrind [NS07b] and QEMU [Bel05]. However, these systems do not prioritise performance and instead focus on providing good support for heavyweight instrumentation (in the case of Valgrind) or multi-architectural cross-ISA translation (QEMU). Their overhead is evaluated using the SPEC CPU2006 benchmarks in Section 4.4.4: Valgrind has a geometric mean overhead of 226% on a Cortex-A9 and 285% on a Cortex-A15, while QEMU has an even higher overhead of 1,907% on a Cortex-A15.

All DBM systems available on ARM introduce very high overheads, which dominate the execution time. This has motivated the development of DBM optimisations specifically for ARM in this thesis. At the same time, a number of low overhead DBM systems exist for the x86 architecture. These provide a reference point for the level of overhead which can be achieved in a highly optimised system. However, the two architectures and their implementations are significantly different, each raising distinct challenges and opportunities, therefore the results cannot be directly compared against each other. This point is also highlighted by the difference in overhead between x86-64 and x86: the overhead of DynamoRIO on x86-64 is almost 40% higher than its overhead on x86, even though the former is derived from, and similar to the latter. Likewise, the hardware platform and the microarchitecture of its processor can have a very strong influence on the results, as shown by the evaluation of StarDBT, whose overhead varies between 12% and 27% in otherwise identical tests.

Chapter 3

Overview of MAMBO

3.1 Introduction

A number of DBM systems were available at the time this PhD project was started, however only Valgrind [NS07b] and QEMU [Bel05] had support for the ARM architecture and were also actively maintained. Neither of them is designed to achieve near-native execution performance and their evaluation results, shown in Section 4.4.4, confirm that both have high overhead. At the same time, the internal architecture and the code base of these systems is fairly complex, making it impractical to redesign them for low overhead in the available time. Similarly, porting DynamoRIO [Bru04] to ARM has been considered and then decided against due to its large code base and different priorities in terms of transparency and complexity.

It became apparent that to investigate high performance DBM on ARM, a different kind of DBM system was required, one which was designed to have low overhead and which prioritised a small, simple code base over a rich feature set. This chapter introduces MAMBO, the DBM framework for the ARM architecture which we created, from scratch, to satisfy these two requirements. MAMBO was used to implement and evaluate all research put forward in this thesis.

The main purpose of this chapter is to provide the context of how the more advanced topics, such as those discussed in Chapters 4 and 5 fit in the context of a DBM / DBT implementation in general and in MAMBO in particular.

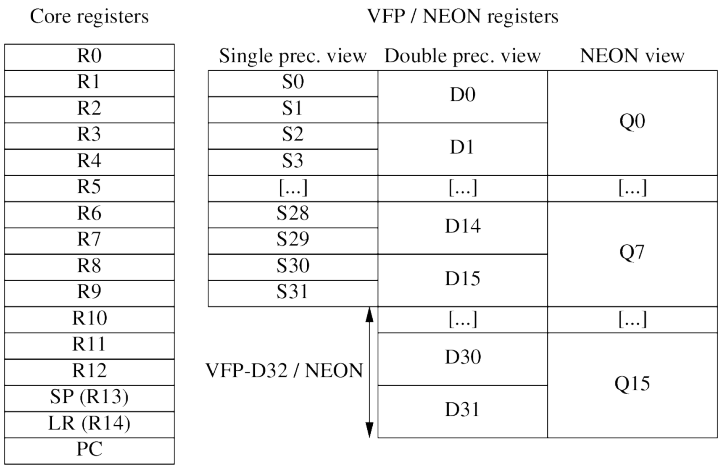


Figure 3.1: The ARM registers.

3.2 Aims and current state

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

The MAMBO implementation has been designed to be largely Operating System (OS) agnostic, however the pragmatic choice was to implement the small OS-specific components for a single OS, Linux.

3.3 The ARM architecture

ARM is a load/store architecture, meaning that data from memory is accessed using a small set of load and store instructions, while data processing instructions

work on registers. Figure 3.1 shows the organisation of ARM registers. There are 15 32-bit general purpose registers and a Program Counter (PC) register, which can be read and written by many of the general purpose instructions. By convention, register R14 is used to store the function return address and it is called the Link Register (LR) and register R13 is used as the Stack Pointer (SP). In addition, an optional floating point extension (called VFP) uses dedicated double precision 64-bit registers, which can also be accessed as 32-bit single precision registers. An optional SIMD extension (commonly called NEON) shares the 64-bit registers of the VFP, while also being able to access pairs of 64-bit registers as 128-bit registers.

One particularity of the ARM architecture is that it implements multiple instruction sets. The ARM instruction set (also known as A32 - not to be confused with AArch32, which is the 32-bit execution state) is the original one, and it uses a fixed instruction word length of 32 bits, with support for conditional execution of most instructions. Thumb was later developed to improve code density and uses an instruction word length of 16 bits. Most Thumb instructions only allow access to the lower 8 registers (r0-r7). Thumb-2 extends the Thumb instruction set and adds 32 bit instructions, which mirror almost all ARM-mode instructions. It was developed to increase the performance and improve the code density compared to Thumb by minimising the number of switches between ARM mode and Thumb mode. All ARMv7-A processors include support for Thumb-2. In this document, the term *Thumb* will be used to refer to the extended set of Thumb and Thumb-2 instructions.

MAMBO currently supports most of the instructions of the ARMv7-A architecture and of the 32-bit execution state in ARMv8 (AArch32), including the ARM and Thumb instruction sets and the optional VFP and NEON extensions. Support for various instructions is improved as they are encountered in the applications ran under MAMBO. At the time of writing this thesis, MAMBO supports 358 Thumb instructions and 137 ARM instructions. The ARMv8 architecture manual [ARM15] defines 293 AArch32 instructions. However, these numbers cannot be compared directly because instructions are defined differently in the two contexts. For example, MAMBO handles a 16-bit and a 32-bit version of an instruction separately, while the manual counts them as a single instruction. Similarly, MAMBO handles small groups of VFP or NEON instructions which do not access the general purpose registers and which have similar encodings as

single generalised instructions. The Jazelle extension and ThumbEE instruction set are not implemented and attempts to use them would trap to a debugging mode. Both have been deprecated and we have not encountered any GNU/Linux application using them.

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

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

3.4 Scratch space

The translated code produced by DBM systems for execution from the code cache often uses additional variables. Because ARM is a load/store architecture and it only supports small immediate operands, these additional values must be loaded in registers. For example, when translating an instruction taking the PC as an input, the value of the SPC must be loaded in a register on ARM, while on x86 a 32-bit immediate could be used directly. While in some cases it might be possible to use dead registers, that is not always the case. When

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

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

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

Unlike other platforms, the ARM ABI prohibits applications from storing valid data on the stack above the stack pointer. This allows safe use of the application stack for scratch space, for applications following the ABI. However, no persistent MAMBO data can be left on the stack, because it would add an offset to stack accesses from the application. Furthermore, care must be taken to fix up stack offsets when generating the translation for instructions which use the stack themselves. Figure 3.2 shows an example of such a translation, which must first reserve stack space for the value pushed by the application before spilling a register. This is the approach currently used by MAMBO.

3.5 Executable loader

MAMBO uses the loading approach over injection (discussed in Section 2.3.1), which ensures that no application code executes before it takes over and also

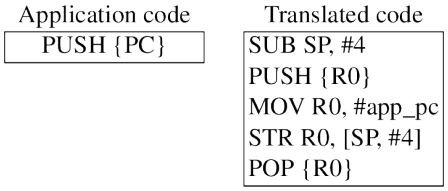


Figure 3.2: Translation using scratch registers for an instruction which accesses the stack.

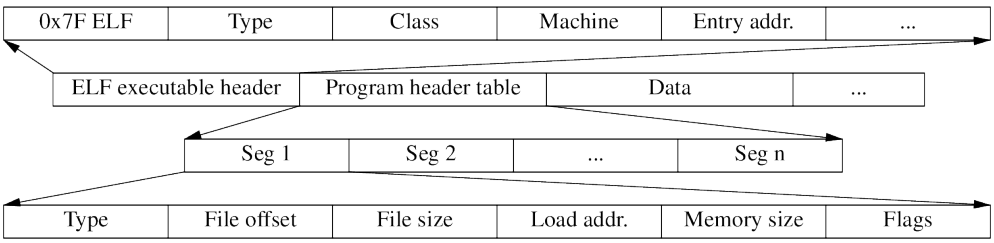


Figure 3.3: Structure of an ELF file.

supports both dynamically and statically linked executables. Because the executable loader used by GNU/Linux is implemented in the kernel rather than in userspace, it becomes necessary to include a userspace executable loader as part of MAMBO. Although GNU/Linux supports a number of different executable formats, e.g. ELF, a.out and ECOFF, in practice ELF binaries are used almost exclusively. Therefore, the lightweight userspace loader implemented in MAMBO only supports ELF files.

3.5.1 The userspace ELF Loader

The loader is responsible for allocating memory and then loading the application from storage. An ELF file contains various fields which need to be parsed for correct loading. Figure 3.3 shows a simplified view of this structure. The ELF executable header is always present and contains general information about the file, such as: type (either shared or executable), class (either 32-bit or 64-bit), the type of machine for which is has been compiled and the entry point (address) of the application. The ELF loader compares the values of these fields against the expected values (i.e. a 32-bit file compiled for ARM) and then passes the entry point to the code scanner. The program header table, another top level structure, describes the segments of an ELF file. The segments are continuous chunks of data which are included in the ELF file. Each segment is described by type (LOAD - application data or NOTE - metadata), offset and size of the

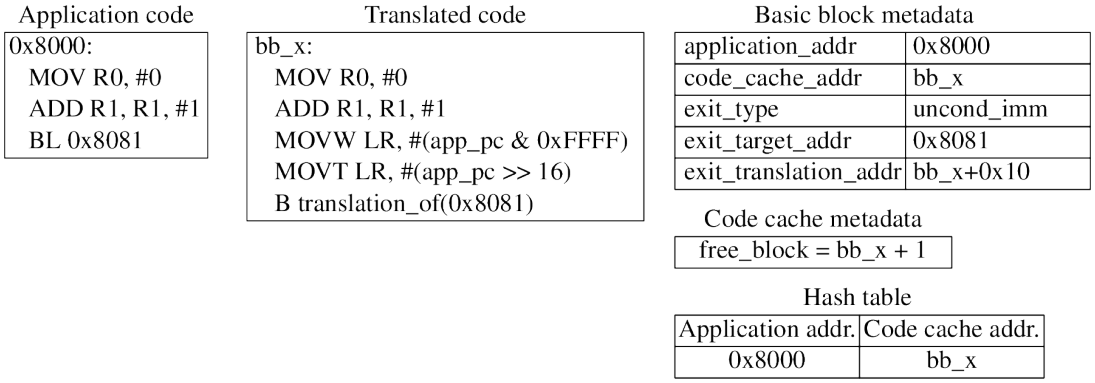


Figure 3.4: MAMBO data structures for an example basic block.

data in the file, the address where to load it in memory, the size of the memory allocation and permission flags (separate for read, write and execute). The ELF loader maps the segments of type LOAD in memory, following the requested addresses, sizes and permissions.

A statically linked ELF executable can be loaded using the information described above. Typically, there are two segments: a code segment with read and execute permissions, and a data segment with read and write permissions. A dynamically linked ELF executable contains additional information, including an entry pointing to an *interpreter* (actually the dynamic linker) and a list of shared library dependencies which need to be loaded by the dynamic linker. When a dynamically linked executable is detected, the ELF loader loads its *interpreter* following the procedure described above, and then prepends the path to the application itself to the launch arguments. The dynamic linker, running under the control of MAMBO, then loads and initialises the application itself.

3.6 Code cache

The code scanner works on short single-entry, single-exit units called *basic blocks*. To amortise the cost of scanning, the generated code is stored in a *code cache*.

MAMBO uses thread-private code caches, which allow scanning and execution from multiple threads with no synchronisation. This design scales fairly well on current systems with 4 to 8 cores, for workloads which use one thread per core to take advantage of parallel execution. However, some applications use a higher number of threads, e.g. as a way to handle blocking I/O or to logically separate different tasks in a process. For example, the Mozilla Firefox web browser uses

tens of threads during routine execution. The performance of MAMBO on such workloads has not been evaluated yet, however it is likely to be limited by the scalability of its shared-nothing architecture. This is a topic for future work, together with investigating a hybrid thread-private / thread-shared architecture to improve scalability.

Figure 3.4 shows the data structures created by MAMBO for an example basic block. Each thread-private code cache consists of a number of data structures:

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

Control is handed off between basic blocks using a *dispatcher*, which is called at the end of each basic block. More efficient mechanisms for handing off execution are the subject of Chapter 4.

3.7 Code scanner

The code scanner reads instructions from the source application, applies any requested modifications and outputs position-independent code which can be executed from the code cache. MAMBO has two code scanners, one for the ARM instruction set and one for the Thumb instruction set. Each of these scanners outputs code using the same instruction set as its input, which allows many types of instructions to be copied into the code cache unchanged by using the C&A approach presented in Section 2.3.2. Both code scanners work in a single pass and manipulate native code directly, without using an intermediate representation, which enables fast code scanning and translation, minimising application startup overhead.

A *code scanner* consists of a loop which reads, decodes and translates one instruction at a time. The type of the instruction is compared to a list of instruction types which need to be translated. The translated instructions include instructions which make use of the PC register (e.g. PC-relative memory operations, data processing operations which use the PC as an input or output) and explicit control flow instructions (e.g. branch, branch-and-link). The instructions whose

type is in this list are passed to translation routines, while other instructions are copied unmodified to the code cache. The scanner stops after translating the first control flow instruction in each block, which ensures that basic blocks have a single exit point.

To maintain correctness of the generated code, the following invariants are maintained:

- any register values which are overwritten by MAMBO-generated code are restored unless they can be proved to be dead;
- either no new instructions which modify the state of the Program Status Register (PSR) are inserted, or, alternatively, the state of the PSR is explicitly saved before executing such instructions and then restored after;
- no memory accesses are inserted between a LoaD Register EXclusive (LDREX) instruction and the matching STore Register EXclusive (STREX) instruction, which could otherwise fail;
- writing to the stack is only allowed above the stack pointer of the application;
- if the translation of an instruction temporarily pushes data on the stack, it must also restore the value of stack pointer, i.e. no MAMBO data is left on the stack between translated instructions; and
- no instructions are reordered with respect to barrier instructions.

3.8 System call interception

All system call instructions are translated into calls to an interception routine, which allows MAMBO to modify the arguments and return values of system calls or even to completely replace them with different operations. This mechanism is used to handle multithreading, signals and to detect events such as thread and application exit. Table 3.1 shows the Linux system calls handled by MAMBO.

3.9 Test and development methodology

MAMBO was developed using iterative testing and implementation on larger and more diverse applications. When new OS features, instructions and variants of

System call	Function	Action(s)
clone	creates a new thread	initialises a new MAMBO thread; scans the entry point in the new thread; emulated using *pthread_create()*
exit	terminates the thread	unmaps thread-private data structures
exit_group	terminates the process	terminates all threads
close	closes a file descriptor	discards attempts to close *stdout* and *stderr* (used by MAMBO)
rt_sigaction	registers a signal handler	replaces the pointer to the signal handler with a pointer to the translation of the signal handler
cacheflush	flushes the data cache and invalidates the instruction cache	flushes the software code cache to maintain consistency between the application code and the translated code
set_tls	sets the thread-local storage pointer	MAMBO stores the thread-local storage pointer of the application in a thread-private MAMBO data structure, this allows it to register its own thread-local storage pointer with the operating system; discards the system call
mmap2	allocates memory	ensures that executable allocations are readable (required for code scanning) and removes the executable permission (to prevent execution of untranslated code)
mprotect	changes the permissions of an existing memory allocation	same as mmap2
munmap	frees allocated memory	flushes the code cache when executable memory is freed to prevent the execution of stale translations
vfork	creates a new process, blocking the parent	ensures that the child uses a separate address space, to prevent execution in the child from overwriting the MAMBO data structures of the parent
exec	executes a program	optional: rewrite the file path to launch the new program under MAMBO

Table 3.1: Linux system calls discarded, emulated or otherwise modified by MAMBO

instructions were encountered in applications, support for them was added to MAMBO. This approach was enabled by maintaining a conservative whitelist of instruction variants known to be handled correctly and by trapping execution, to a development and testing mode, on instructions not on the whitelist.

Testing for correctness was done by comparing the output when executing under MAMBO against the output of native execution. The early testing and implementation was done by running hand-crafted programs and Valgrind [NS07b] tests, while later on the full benchmark suites used in the evaluation and standard GNU/Linux applications were used. By only implementing the functionality required to execute the relevant workloads, we have achieved the aim of maintaining the code base small and relatively simple. Furthermore, by mostly reusing

existing applications as test cases, we have minimised the time required for testing. This approach was practical because the translation or modification of most instructions can be treated statelessly, in isolation from the rest of the code. However, this is not always the case, e.g. for IT instructions in Thumb mode which affect the execution of the following instructions. In these cases, we have found that designing tests which exhaustively cover corner cases is more practical than testing against a multitude of existing applications.

The downside of this methodology is that only the instructions and behaviours observed during development and testing are supported and new executables or even different inputs to known applications might trap to the development and testing mode. Indeed, this has occasionally been observed when using different compilers or compiler options to build previously passing test applications. This is a minor issue considering that MAMBO is not intended to support running every possible application. Nevertheless, these occurrences have been largely eliminated over time by testing against the builds of a variety of applications provided by multiple GNU/Linux distributions.

3.10 Plugins

MAMBO is a DBM framework and therefore does not apply any behavioural changes to the application; it only applies the transformations required to efficiently run applications from the code cache. Additional modifications can be performed through the plugin API, which allows plugins to inspect and modify the instruction stream, and to observe the same events available to MAMBO, e.g. code scanning events, system calls and application exit. The plugin API is used to implement tools for dynamic instrumentation, program analysis, etc. on top of MAMBO. A full description of this API is outside the scope of this thesis. However, an example plugin is provided in Appendix A.

3.11 Transparency

This thesis introduces the principle of *behavioural transparency*, defined as iteratively identifying the *types* of transparency required for correct execution of the selected workloads and then implementing the minimum *degree* of transparency required to achieve correct execution. Behavioural transparency is an alternative

to *full transparency*, which is the approach of implementing a *best effort* degree of transparency for all transparency types. The aim is to reduce the complexity and development time and is particularly suitable for research DBM systems, which can adjust the tradeoff between the range of supported applications and the resources used to support transparency. In the case of MAMBO, we aim to support applications which follow the platform ABI, do not depend on undefined behaviour (as described by the ARM architectural manual [ARM15]) and use standard system libraries.

One design decision affecting transparency is to use the application stack for scratch space, as described in Section 3.4. A fully-transparent implementation would require the availability of a scratch register, either by stealing it from the application or by temporarily storing its value inside the code cache (due to the limited addressable range), likely incurring additional overhead. Because the ARM ABI prohibits applications from storing data on the stack at addresses lower than the stack pointer, a behaviourally transparent implementation can safely store temporary data (used in the translation of a single application instruction) on the stack. The only case when the scratch data on the stack requires special handling is for signal delivery; signals must be delivered when no temporary data is present on the stack.

3.12 Summary

This chapter introduces MAMBO, the DBM framework which was created to facilitate performance-related research of DBM systems for the ARM architecture. This was achieved by making its main design priorities low overhead and maintaining a small code base. Furthermore, MAMBO relaxes the transparency guarantees compared to other DBM systems when doing so can measurably improve performance or save development effort, without affecting typical workloads (*behavioural transparency*).

This chapter also provides an overview of the ARM architecture, in particular of its limitations relevant to a DBM system. The architectural challenges which have extensive effects and the ways in which they are mitigated in MAMBO are also discussed.

MAMBO is organised in multiple sub-systems, including a *code scanner* which generates the modified code, a *software code cache* which stores this code, and

a *dispatcher* which manages execution. These systems are used and further extended to implement the optimisations described in Chapters 4 and 5. The next chapter describes the techniques used to improve the performance of control transfers between basic blocks.

Chapter 4

Branch linking

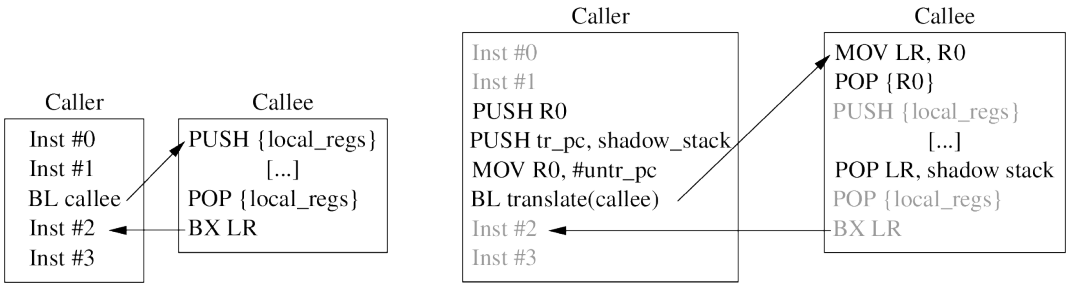
4.1 Introduction

With the basic code cache introduced in Section 3.6, control transfers between basic blocks are mediated by the runtime system, which involves two expensive context switches every time a translated branch instruction is executed. This is easily the dominant source of overhead in a DBM system and reducing it by directly linking together basic blocks is an active area of research [HWH+07, PG10, DH11b, JYW+13, dGGL16]. Branches can be broadly classified in direct (Section 4.3) and indirect (Section 4.2) branches, depending on whether their target address is static or dynamic. Both direct and indirect branches can be further classified in multiple sub-types which can be handled differently by a DBM system.

This chapter presents several novel linking schemes, in particular for dealing with indirect branches. In addition, other existing branch linking schemes have been ported to the ARM architecture and are also briefly discussed in this chapter.

4.2 Indirect branches

Indirect branches are control flow instruction with a target not known at translation time. Looking up the target of indirect branch instructions at runtime is the major source of overhead for DBM systems [KS03b]. Two types of indirect branch instructions can be handled specially:



(a) The original function call. (b) The translation. Unmodified code is greyed out.

Figure 4.1: Example of a typical function call and the translation generated by MAMBO.

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

4.2.1 Function returns: low overhead return address prediction

Figure 4.1(a) shows a typical function call in ARM code. A *caller* function contains a branch-and-link to the entry address of the *callee*. The callee preserves the return address, executes, and returns to the instruction following the branch-and-link in the caller using a return instruction which branches to the return address in the link register. Because it branches to an address in a register, this return instruction is an indirect branch. Fast return address prediction in DBM systems has been shown to be critical for achieving low overhead [KS03b].

While the return instruction is an indirect branch because its target address cannot be determined statically, it has the property that its translated address can be predicted with very high accuracy when its matching branch-and-link instruction executes. This property is sometimes exploited by a technique which stores pairs of untranslated and translated addresses on a software Return Address Stack (RAS) for every call instruction. Return instructions then load the untranslated address from the RAS, compare it with the return address in the

application and, if they match, directly return to the translated address from the RAS. We call this linking scheme a *fat entry* RAS return address predictor. This optimisation was used by several DBM systems, including Pin for ARM [HK06]. However, when implemented in MAMBO, this scheme causes a slowdown compared to our inline hash lookup (presented in Section 4.2.3), a result similar to that experienced by DynamoRIO [Bru04]. Additionally, the performance overhead of maintaining a software RAS has been shown to be around 10% compared to direct native execution of returns [DMW15].

We have developed an alternative scheme which trades off some transparency guarantees for increased performance, while still being able to execute typical applications correctly. Return type instructions are initially translated to exits to the dispatcher and their basic block is marked as exiting with a return instruction. When the dispatcher handles an exit from this type of basic block, it looks up the translated address and compares it with the entry at the top of the RAS. In case of a match, the dispatcher rewrites the exit code in the basic block to directly branch to the address at the top of the RAS. Translated returns are handled through the dispatcher only when they execute for the first time. Further executions of a translated return are handled through the fast RAS-based return operation inlined in the basic block. Our implementation of the call and return operations is shown in Figure 4.1(b).

Comparison with fat entry RAS return address prediction

Compared to the pair of 32-bit addresses used by a fat-entry RAS return predictor, the low overhead return predictor only pushes and pops a single 32-bit value (the predicted code cache return address) for each branch-and-link and return instruction. On ARM, this eliminates 4 instructions from the critical execution path. In addition, the correctness of predictions is not checked on the critical path for every return instruction. This eliminates another 4 instructions and also avoids placing a conditional branch in the translation of returns, which would increase the pressure on the branch predictor and affect branch prediction rates. On applications using deep nested function calls, the pressure on the data cache and on the data TLB is reduced due to the smaller size of the RAS.

Restrictions

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

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

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

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

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

The only situation where we have encountered this behaviour is in applications that throw exceptions. Because exception handling is done by unwinding the stack to search for exception handlers, it is possible for stale entries to remain on the RAS. However, the *libgcc* exception handling code we have examined is

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

4.2.2 Table branches: space-efficient linking

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

We propose a scheme which adapts the *shadow jump table* linking scheme introduced by fastBT [PG10] (previously described in Section 2.4.2 on page 36) for use on the ARM architecture and also improves the space efficiency of the shadow branch table.

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

The space-efficient shadow branch table we have implemented, shown in Figure 4.2, uses a two level table, where the *offset table* contains *max_index* bytes. This table encodes offsets into a *trampoline table* which contains direct branches (of size 4 bytes) to basic blocks. The size of this shadow table is:

$$mambo\_table\_size = max\_index + 1 + selected\_indexes \cdot 4$$

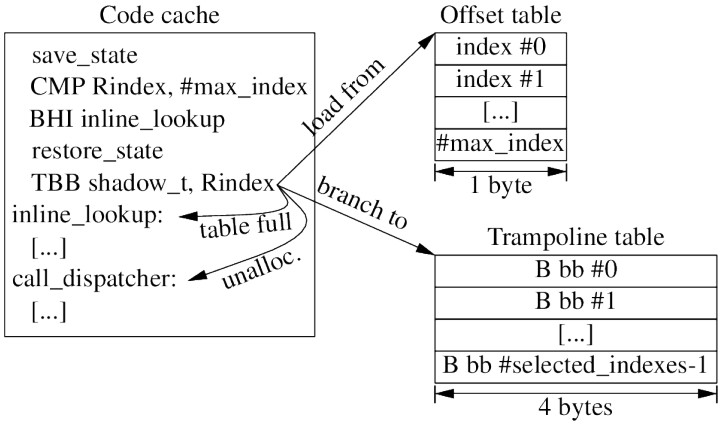


Figure 4.2: Space-efficient shadow branch table.

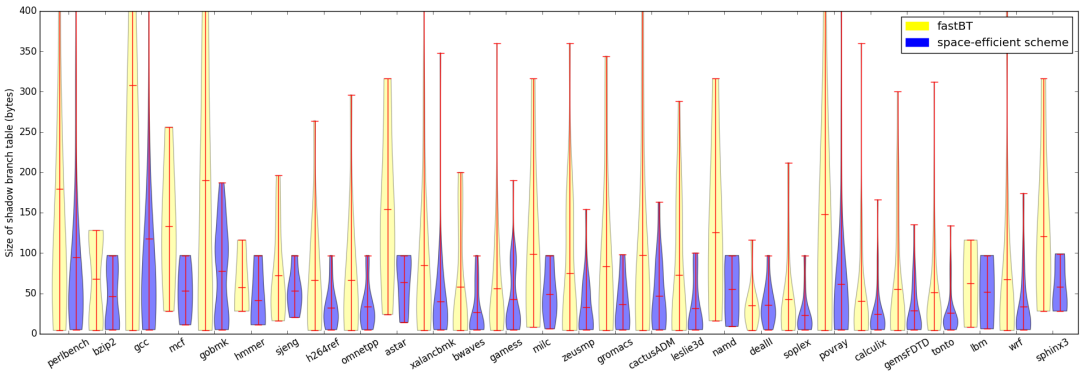


Figure 4.3: Comparison of shadow branch table size for SPEC CPU2006.

We have profiled the two parameters, *selected_indexes* and *max_index* for table branch instructions executed by SPEC CPU2006 benchmarks. The violin plot in Figure 4.3 shows the distribution of shadow branch table sizes for the two schemes. The width of the curve at various positions on the vertical axis shows the distribution of shadow branch tables of the corresponding size in each benchmark. The red horizontal markers show the minimum, mean and maximum (when under 400 bytes) sizes. Our scheme reduces both the median and maximum shadow branch table size for all SPEC CPU2006 benchmarks. Across all SPEC CPU2006 benchmarks, the mean shadow table size is reduced from 159 bytes to 72 bytes and the maximum size is reduced from 2,700 bytes to 1,388 bytes.

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

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

Implementation Figure 4.2 shows our implementation of the space-efficient shadow branch table. The size of the *offset table* determines the maximum index which can be handled; the size of the *trampoline table* determines how many unique indexes can be cached. Indexes higher than *max_index* are handed off to an inline hash lookup routine. Similarly, if the trampoline table gets filled, any additional indexes which execute are redirected to the inline hash lookup by setting the appropriate offset in the offset table. The inline hash lookup routine is discussed separately in Section 4.2.3. The current value of *max_index* is 152 and the size of the trampoline table is 32 entries. These values were chosen experimentally as a compromise between size requirements and performance for the SPEC CPU2006 benchmarks. These sizes allow over 99.9% of the table branches executed across SPEC CPU2006 benchmarks to be cached in the shadow branch tables.

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

Does the extra level of indirection affect performance? Even though the space-efficient shadow branch table appears to trade off performance for space compared to the single level fastBT shadow branch table, experimental results do not match this. When evaluated using a microbenchmark which applies no pressure on the branch predictors or caches, the fastBT scheme is around 5% faster than the space-efficient scheme. However, when used by MAMBO running a more

```
struct hash_table_entry {
  uint32_t application_address;
  uint32_t code_cache_address;
};

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

Listing 4.1: Structure of the hash table.

complex workload, such as the SPEC CPU benchmarks, the space-efficient scheme is consistently faster than the fastBT scheme. Benchmarking and profiling results are discussed in Section 4.4.3.

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

4.2.3 Inline hash lookup for indirect branches

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

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

The hash lookup routine is encoded inline in the basic blocks which contain the translation of an indirect branch. This allows adapting every instance to

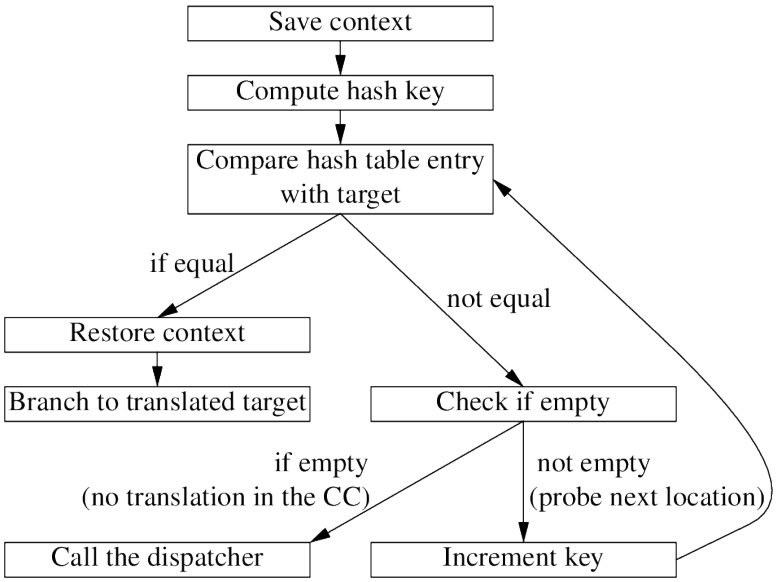


Figure 4.4: Inline hash lookup routine.

minimise context switch cost and also allows the processor to handle branch target prediction for every translated indirect branch individually.

The structure of the hash table is defined in Listing 4.1. Each entry consists of a pair of 32-bit addresses: the untranslated application address and the translated address in the code cache. Our implementation uses linear probing to solve collisions. To minimise the number of required registers, there is no wrap-around. Instead, a number of additional slots are used to handle possible collisions at the end of the table, followed by a guard entry, identical to empty slots, which marks the end.

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

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

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

Figure 4.4 shows the inline hash lookup routine. The first operation saves the context. This operation is specialised for each instance of the inline hash lookup and frees up a number of scratch registers. It only saves a reduced number of

registers or none at all if a stack pop in the application can be delayed until the hash lookup routine has completed, subject to data dependencies.

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

All inline hash lookup routines access a single thread-private hash table, therefore encoding a new routine only takes up code cache space and not any additional data space. The size of the inline hash lookup routine is between 94 and 118 bytes in Thumb mode and either 116 or 120 bytes in ARM mode, depending on the type of translated instruction and available registers. This includes fallback code for calling the dispatcher if the translation of the target address is not yet present in the code cache. Section 4.4.5 evaluates the overall code cache overhead when enabling inline hash lookups for SPEC CPU2006 benchmarks.

This inline hash lookup procedure is similar to others described in the literature (Section 2.4.2, page 36), in particular to that used by DynamoRIO [Bru04]. Apart from being implemented for ARM, the other differences from the DynamoRIO inline hash lookup is that 1) collisions are handled directly instead of falling back to a shared procedure and 2) the size of the hash table is fixed instead of being dynamically resized.

4.2.4 Fallthrough branch linking

Conditional branches have an implicit fallthrough branch to the following instruction. The fallthrough branch is only taken if the conditional branch is skipped. Even if the conditional branch is indirect, the fallthrough branch is always a direct branch, which can be directly linked in the translated code. Figure 4.5 shows how fallthrough branches are translated: MAMBO links the fallthrough branch by placing a conditional direct branch with the opposite condition compared to that of the source branch before the indirect branch lookup routine.

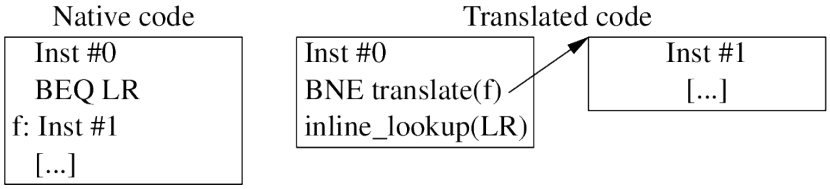


Figure 4.5: Linking of fallthrough branches.

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

4.2.5 Indirect branch target prediction

Other DBM systems, such as Pin [LCM+05], use a short series of inlined compare-and-branch sequences as their main method of resolving indirect branches. However, we have found that on ARM such an approach is generally outperformed by our fast inline hash lookup system. A fundamental limitation of a compare-and-branch predictor is that updating the predicted addresses is a relatively expensive operation (as opposed to a hardware indirect branch predictor, which can be updated every time the indirect branch executes) therefore the performance of the system relies on indirect branches being predictable and on using a good heuristic to decide which target to select. However, by instrumenting the indirect branches in SPEC CPU2006 benchmarks which make heavy use of indirect branches, we have determined that many indirect branches are not easily predictable. An oracle which always predicts the address taken most often for each indirect branch can get misprediction rates as high as 80% for *perlbench* with *splitmail.pl*, 52% for *sjeng*, or 47% for *gcc* with *scilab.in*. A practical branch predictor heuristic is likely to have even higher miss rates.

A second issue with compare-and-branch predictors is that only a few execution cycles can typically be saved compared to an inline hash lookup, only when both the software predictor and the hardware branch predictor hit. However, the compare-and-branch predictor uses additional conditional branches which can be

mispredicted by the hardware branch predictor. Hardware branch mispredictions have a latency proportional to the number of pipeline stages (e.g. around 20 cycles for Cortex-A15) and can easily start to dominate the lookup time. An inline hash lookup is generally expected to cause at most one hardware branch misprediction, only if the target is different compared to the last execution of the branch. On the other hand, a 2 entry compare-and-branch predictor with a fallback inline hash lookup can cause up to three branch mispredictions.

4.3 Direct branches

4.3.1 Direct branch linking

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

There are two types of direct branches:

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

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

Conditional branches which use the status register are linked as a conditional branch (implemented as an IT instruction followed by a branch) and one unconditional branch, to link both possible execution paths. Compare and branch

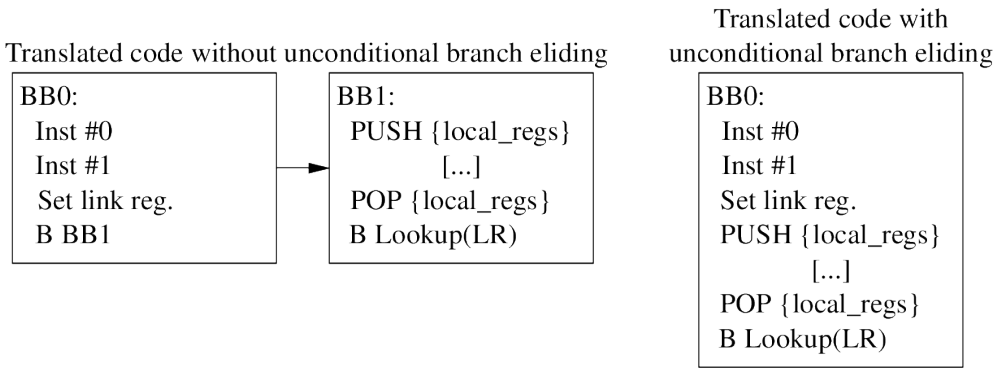


Figure 4.6: Comparison between translations with and without unconditional direct branch eliding.

on zero/nonzero (CBZ/CBNZ) instructions conditionally branch depending on the result of the comparison between a register and the value 0. These have a very limited range so are linked using a CB(N)Z instruction and two unconditional branches, with the CB(N)Z instruction conditionally skipping over the first unconditional branch.

4.3.2 Eliding unconditional direct branches

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

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

Figure 4.6 shows a comparison between the translations of the application code from Figure 4.1(a) with and without eliding the unconditional direct branch. In this example, by eliding the unconditional branch, the number of basic blocks is

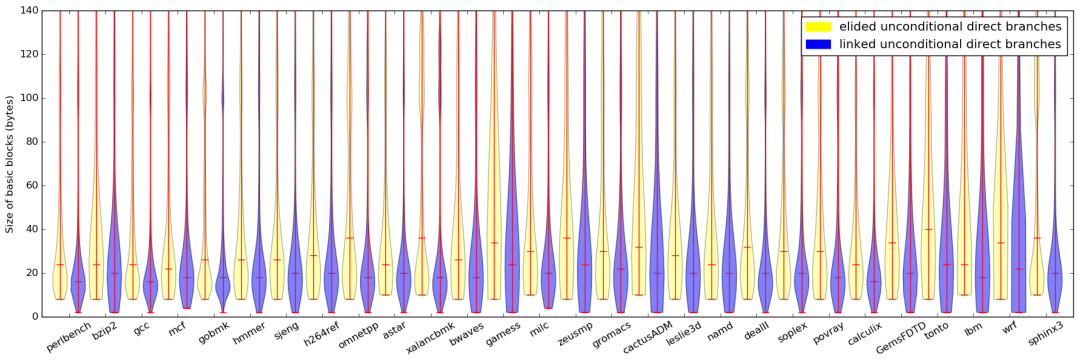


Figure 4.7: Comparison of basic block sizes.

lowered to one, the total number of instructions is reduced and a branch instruction is eliminated.

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

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

The violin plot in Figure 4.7 compares the distribution of basic block sizes depending on whether this linking scheme is enabled or not. The width of the curve at various positions on the vertical axis shows the distribution of basic blocks of the corresponding size in each benchmark. The red horizontal markers show the minimum and median sizes. It can be observed that eliding unconditional branches increases both the median and minimum size of basic blocks compared to linking unconditional branches. Four byte basic blocks (i.e. the minimum size of basic blocks for most benchmarks with linked unconditional direct branches), generated when the first instruction in a basic block is an unconditional direct branch, are completely eliminated.

4.4 Evaluation

4.4.1 Experimental setup

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

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

Power management features such as DVFS and core power-gating were disabled and a fan was added to the passive heatsink of the ODROID-X2 to minimise the risk of thermal throttling. SPEC CPU2006 was compiled with GCC 4.6.3 and PARSEC 3.0 was compiled with GCC 4.8.2, both configured to generate Thumb code (which is the default configuration) with NEON and VFP support, at the *-O2* optimisation level. Non-essential services were disabled and the systems were otherwise idle.

The *libquantum* benchmark from the SPEC CPU2006 suite has been disabled because it fails to complete, both when executed natively and under MAMBO. All other CPU2006 benchmarks are enabled when running natively or under MAMBO and produce the expected output. Valgrind fails to load the *zeusmp* benchmark because of its large BSS section (1.1 GiB) and throws an exception when running *povray* because it fails to decode a valid *ADD.W* instruction. The benchmarks *Canneal* and *Raytrace* from PARSEC 3.0 do not build on ARM. *fluidanimate* requires the number of threads to be a power of two, therefore it could only execute with 1, 2 and 4 threads (i.e. not with 3 threads). All SPEC CPU2006 results were obtained using the *ref* data set and all PARSEC 3.0 results were obtained using the *native* data set.

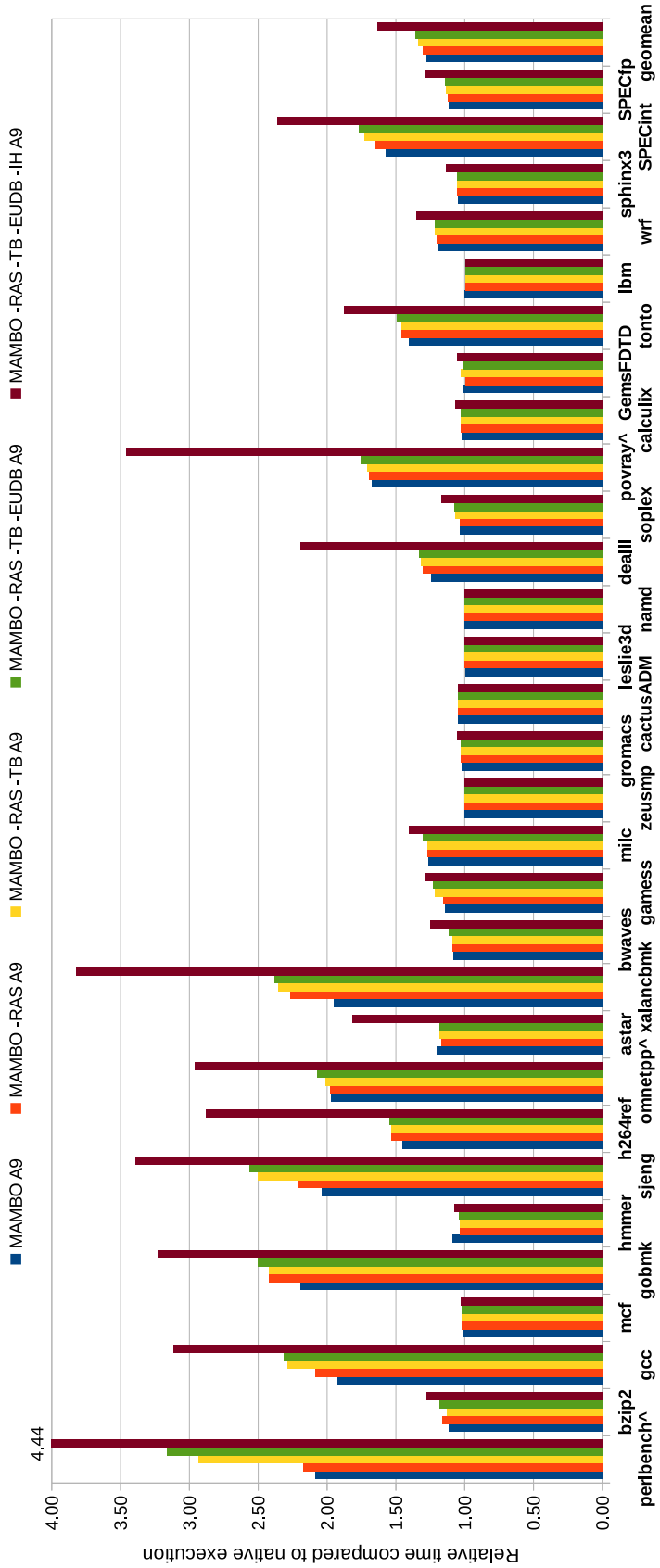


Figure 4.8: Relative execution time for SPEC CPU2006 with the *ref* dataset on ODROID-X2.

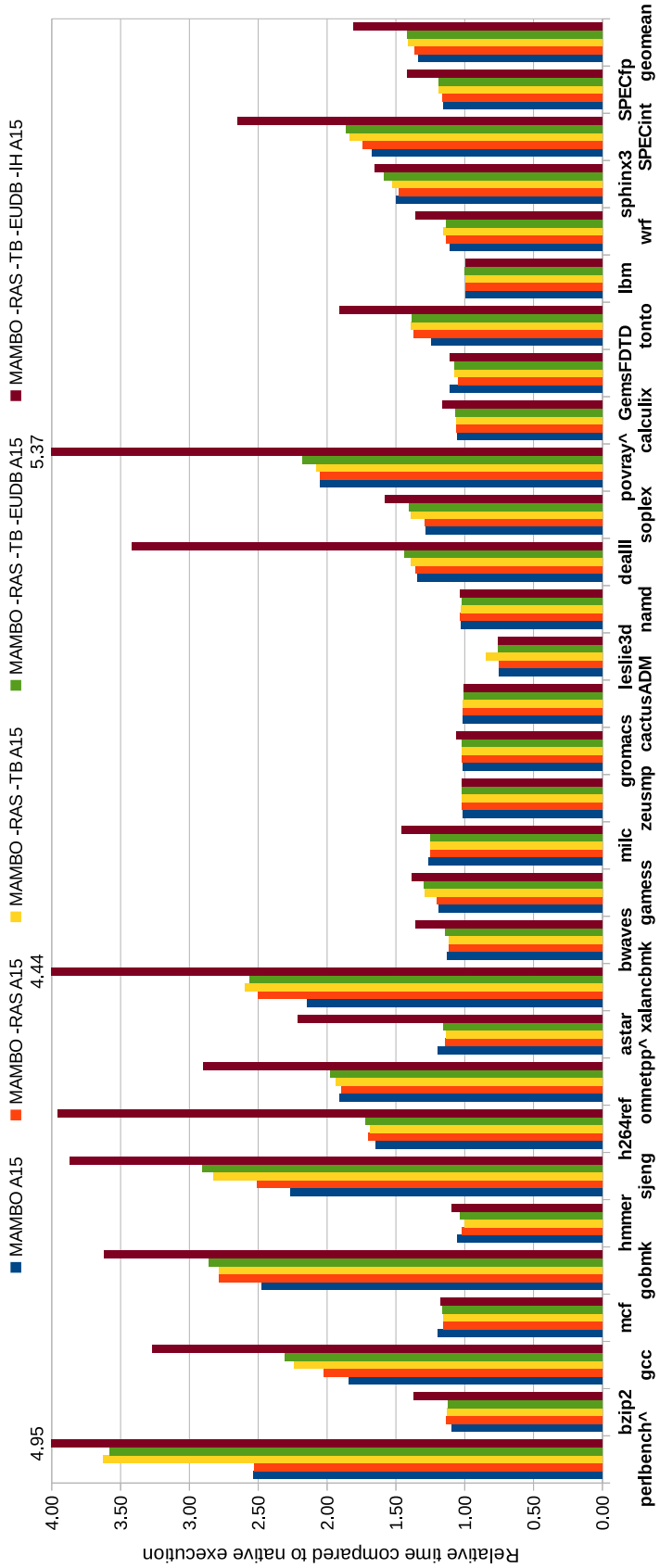


Figure 4.9: Relative execution time for SPEC CPU2006 with the *ref* dataset on Jetson TK1.

4.4.2 Contribution of different optimisations

Figures 4.8 and 4.9 show a comparison of overhead with different types of optimisations being enabled. The results using the *A9* and *A15* suffixes were obtained on the ODROID-X2 and Jetson TK1 systems respectively. The five configurations which were evaluated are:

- MAMBO - all linking schemes are enabled, it is the fastest version;
- MAMBO-RAS - return address prediction using the software RAS is disabled and return instructions are translated to inline hash lookups;
- MAMBO-RAS-TB - both return address prediction and table branch linking are disabled, table branches are handled using the dispatcher;
- MAMBO-RAS-TB-EUDB - the previous two linking schemes are disabled and unconditional direct branches are being directly linked to the basic block containing the translation of their target instead of being elided; and
- MAMBO-RAS-TB-EUDB-IH - the previous three linking schemes and the inline hash lookup are disabled. All types of indirect branches are handled using calls to the dispatcher.

The benchmarks marked with a caret (ˆ) cause return address mispredictions which require disabling of the RAS predictor at some point during execution. These are *perlbench*, *omnetpp* and *povray*.

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

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

Enabling the return address predictor reduces the overhead of benchmarks with many calls to functions which return relatively quickly. At the same time,

benchmarks with high data cache pressure can be negatively affected by the additional operations on the RAS and benchmarks with high instruction cache pressure can be negatively affected by the increased code cache size.

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

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

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

4.4.3 Comparison of the space-efficient and fastBT table branch linking schemes

To compare the two table branch linking schemes, we have limited the set of benchmarks to the six which make significant use of table branches: *perlbench*, *gcc*, *sjeng*, *gamess* and *soplex*, using the *ref* dataset. Figure 4.10 shows the slowdown of two fastBT configurations relative to the space-efficient scheme:

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

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

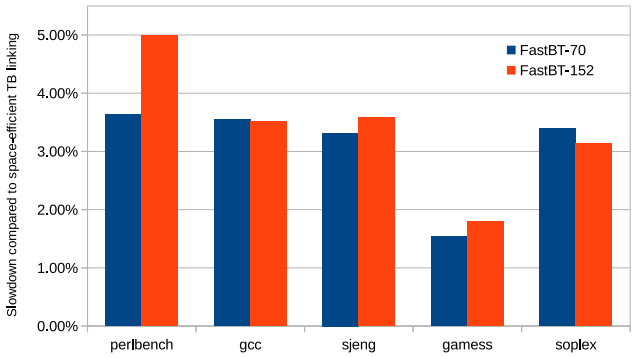


Figure 4.10: Relative slowdown for selected SPEC CPU2006 benchmarks with the fastBT table branch linking scheme.

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

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

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

Profiling using the performance counters on the Cortex-A15 shows that the space-efficient scheme runs with fewer branch mispredictions compared to the fastBT shadow branch tables (see data in Table 4.1). For example, perlbench runs with 40% fewer mispredictions on the Cortex-A15 when using the space-efficient shadow table compared to the fastBT-152 shadow table. Unfortunately,

the indirect branch predictor in these ARM cores is not documented, therefore the root cause of this behaviour remains unexplained.

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

Table 4.1: Number of branch mispredictions on Jetson TK1 with different implementations of shadow branches tables.

4.4.4 Overall performance

Singlethreaded performance: SPEC CPU2006

The results in Figure 4.11 were obtained by running SPEC CPU2006 with the *ref* dataset, using MAMBO with all linking schemes enabled, Valgrind 3.10 and QEMU 2.0.0 in user mode. QEMU results are only reported for the Cortex-A15 system, due to time constraints caused by the high overhead of QEMU. We estimate that SPEC CPU2006 would take more than 20 days to finish running under QEMU on the ODROID-X2. The geometric mean of overheads is summarised in Table 4.2.

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

Table 4.2: Summary of geometric mean overheads for MAMBO, Valgrind and QEMU running SPEC CPU2006.

MAMBO has higher overhead when running on Jetson TK1 compared to ODROID-X2 for most benchmarks. The likely causes of this pattern are described in Section 4.4.2. In some cases the difference can be very large (e.g. from 3% to 28% overhead for soplex) and it warrants further investigation. However, several benchmarks have lower overhead on Cortex-A15 compared to Cortex-A9. These benchmarks likely benefit from having a larger L2 cache, which can better accommodate the larger working set of translated applications.

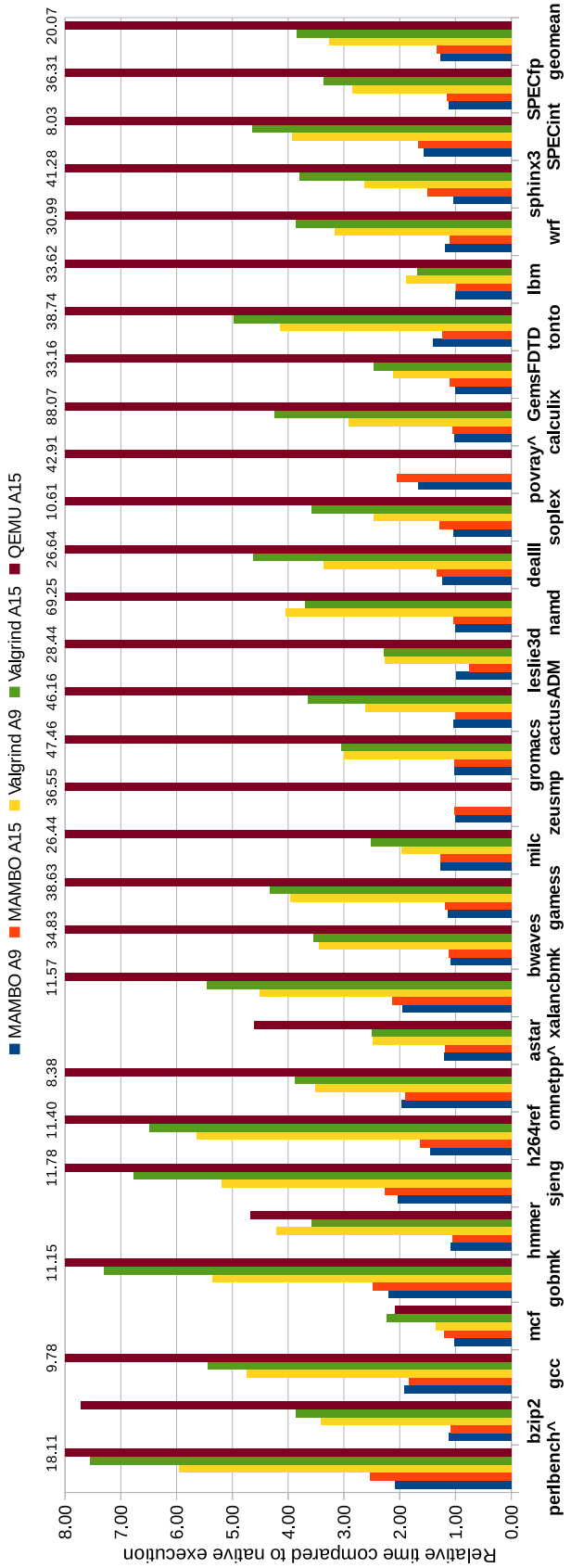


Figure 4.11: Relative execution time for SPEC CPU2006 under MAMBO, Valgrind and QEMU (*ref* dataset).

Some benchmarks execute with significant overhead under MAMBO. By profiling using the hardware performance counters, the main causes have been determined to be poor L1 instruction cache utilisation leading to a significant increase in cache misses compared to native execution and high instruction TLB miss rates. Both issues are caused by the fixed-size basic block layout used by MAMBO. This layout causes basic blocks to exclusively use at least one cache line (32 bytes on a Cortex-A9 or 64 bytes on a Cortex-A15) even when their length is significantly shorter. It also spreads out the translated code across more pages than the untranslated code. This issue could be addressed either by modifying the code cache to use variable size basic blocks or by implementing traces. In Chapter 5, we propose and evaluate the use of traces in MAMBO.

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

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

Multithreaded performance: PARSEC 3.0

The results in Figure 4.12 were obtained by running PARSEC 3.0 with the *native* dataset and MAMBO with all linking optimisations enabled, on the Jetson TK1

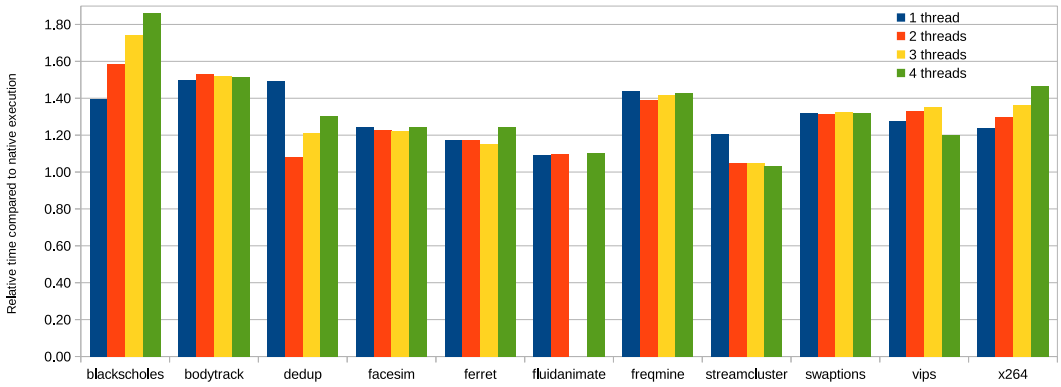


Figure 4.12: Relative execution time for PARSEC 3.0 with the *native* dataset.

board. The geometric mean is 1.30 when running with one thread, 1.27 for two threads and 1.32 for three and four threads.

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

4.4.5 Code cache size

The size of the code cache has been measured in the same five configurations described in Section 4.4.2. The results for all SPEC CPU2006 benchmarks are shown in Table 4.3. For benchmarks which are launched multiple times with different inputs, the arithmetic mean is shown. For benchmarks which cause return mispredictions and a subsequent flushing of the code cache in the *MAMBO* configuration - marked with a caret (^) - only the higher value between the size at the time of the misprediction and the size at exit is shown. Note that MAMBO uses lazy linking for conditional branches; if a translation of either of the two possible targets does not exist in the code cache yet, an exit stub which calls the dispatcher is generated. The size of these exit stubs is included in the

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

Table 4.3: Code cache size for SPEC CPU2006, in KiB and number of basic blocks.

reported code cache size, even though they will execute, at most, two times and are therefore expected to have a minimal impact on the hardware cache hit rate.

Enabling the inline hash lookup (column *MAMBO-RAS-TB-EUDB* vs *MAMBO-RAS-TB-EUDB-IH*) increases the size of the code cache across SPEC CPU2006 benchmarks by around 1.3% and makes no impact on the number of basic blocks. When unconditional direct branches are elided (column *MAMBO-RAS-TB* vs *MAMBO-RAS-TB-EUDB*), the size of the code cache is increased by around 1% due to code duplication, however the number of basic blocks is reduced by approximately 3.8%. When table branch linking is enabled (column *MAMBO-RAS* vs *MAMBO-RAS-TB*), the code cache size increases by another 1.9% and the numbers of basic blocks increases by around 0.3%, due to the space required for shadow branch tables. Enabling return address prediction (column

MAMBO vs *MAMBO-RAS*) increases the code cache size by 21.9% and reduces the number of basic blocks by 9.8%. Due to the fixed size basic block layout used by MAMBO, an increase in code cache size is not necessarily going to increase the pressure on the hardware instruction cache, as long as the number of basic blocks is not also increasing.

4.5 Summary

This chapter presented a number of techniques for efficient handling of control transfer instructions. These instructions can be classified in two types: direct branches, when their target is static and indirect branches, when their target is not known at translation time and which might be dynamic. Indirect branches can be further classified into *function returns* (used to exit a procedure and return to its caller), *table branches* (indirect branches which load their target from a static table) and *generic indirect branches*. Conditional execution is allowed both for direct and indirect branches.

Techniques for handling all of these cases are provided. Of particular importance are the novel contributions for handling function returns (in Section 4.2.1) and table branches (in Section 4.2.2). An efficient ARM implementation of *inline hash table lookups* for handling generic indirect branches is described in Section 4.2.3, while the handling of various types of direct branches available on the ARM architecture is presented in Section 4.3.

The *Evaluation* section (Section 4.4) is organised in two parts: first, the effect of individual optimisations is shown in Section 4.4.2, then the overall performance is evaluated and compared against other DBM systems (Valgrind [NS07b] and QEMU [Bel05]) in Section 4.4.4. The overhead of MAMBO is significantly lower than that of either of these systems: a geometric mean overhead of 28% on a Cortex-A9 and 34% on a Cortex-A15 for SPEC CPU2006, compared to 226% on a Cortex-A9 and 285% on Cortex-A15 for Valgrind and 1900% for QEMU on a Cortex-A15.

Chapter 5

Microarchitectural optimisations

5.1 Introduction

The optimisations presented and evaluated in Chapter 4 have focused on improving the translated code at the architectural level, i.e. by reducing the number of executed instructions or by reducing the size of the generated translation to allow more code to fit in the software code cache. However, in the evaluation it became apparent that some of the performance overhead is introduced by the interaction between the generated code and the microarchitecture. In particular, significant overhead seemed to be caused by the high number of instruction cache load misses and instruction TLB load misses.

This chapter introduces a number of optimisations for improving performance at the microarchitectural level. First, *traces* are introduced to improve the locality of the code cache and increase the instruction cache and TLB hit rates (Section 5.2). Then, a technique for enabling hardware return address prediction for translated returns in the code cache, without the overhead of maintaining a software return address stack is presented in Section 5.3.1. Next, the data cache and TLB footprint of the inline hash table lookup procedure is minimised in Section 5.3.2. Adaptive indirect branch inlining, a form of software indirect branch target prediction which avoids the hardware branch misprediction penalty of indirect branch inlining, is then introduced in Section 5.3.3. Finally, the use of huge pages is evaluated for minimising the number of instruction TLB load misses and for reducing the impact of SPC-to-TPC lookups on the data TLB (Section 5.3.4). The evaluation of these optimisations is done both by directly

measuring the overhead, but also by using hardware performance counters to directly observe their effect on a number of performance events.

The strong effect of the interaction between the code and the microarchitecture raises another question: given the wide range of currently available ARM microarchitectures, how much are the results of the evaluation likely to change depending on the selection of systems used in the evaluation? Furthermore, could an optimisation technique improve performance on one microarchitecture while harming performance on a different microarchitecture? To answer these questions, the evaluation of these optimisations is done on five different microarchitectures, ranging from a small in-order core (Cortex-A7) and up to a server-grade out-of-order APM X-Gene 1 core.

5.2 Traces

The baseline code cache, organised in basic blocks, creates and stores the basic blocks in the order they are first executed. However, the basic blocks in the software code cache have high fragmentation, making inefficient use of the hardware code cache. Furthermore, the two paths of conditional branches are translated in two separate basic blocks in the software code cache, increasing the number of executed branches (by executing a branch in the translated code even when the source conditional branch is not taken). A breakdown of how performance is affected by the code cache is shown in Sections 5.4.4 and 5.4.4 of the evaluation. To avoid these limitations, traces (also known as *superblocks*) were implemented for MAMBO. Traces are single-entry, multiple-exit units built by merging together multiple basic blocks on the hot code path. The single-entry, single-exit units which make up a trace are called *trace fragments*. The concept of code cache traces was introduced in Section 2.3.3.

Because creating a trace has a non-trivial cost (both in terms of code cache space, and execution time spent creating the trace instead of running the application), it is important to only create traces for hot code, which is expected to execute many times in the future and amortise its creation cost. On the other hand, to get the best performance, it is preferred to create traces for all of the hot code in an application and as early as possible. The challenge is in 1) quickly identifying the hot code in an application and 2) in profiling the hot execution

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

Table 5.1: Overview of the NET algorithm.

paths through this code with low overhead. MAMBO builds traces using an improvement of the Next Executing Tail (NET) online profiling scheme [DB00]. The NET algorithm is summarised in Table 5.1. It is designed to minimise the profiling overhead. Towards that end, NET initially maintains an execution counter only for the basic blocks which are the potential start of a hot path. These instrumented basic blocks are called *trace heads*. The insight is that the hot execution path must consist of cycles, therefore NET uses the targets of backwards branches (both direct and indirect) as trace heads. Once the execution counter for a particular trace head reaches a certain threshold, then the trace is considered hot and NET records the full execution path following the trace head, until a backwards branch is encountered (which *terminates* the trace). This record is then used as the predicted path, based on the rationale that the *trace tail* following a hot trace head is also likely to be part of the hot execution path. For example, let us consider the Control Flow Graph (CFG) depicted in Figure 5.1, where each box represents a basic block and block A ends with a conditional direct branch, blocks B, D, E, F, G, H and I end with unconditional direct branches, while block C ends with an unconditional indirect branch. The trace heads in this examples are the two blocks which are the target of backwards branches: A and C. If, for example, the execution count threshold is reached for the trace head A and then the blocks CEH execute, the trace will consist of the blocks $ACEH$, ending with a branch back to the beginning of the trace.

An important property of NET is that it builds traces across indirect branches, statically predicting their target address to be the same as observed in the path recording phase. In the previous example involving the trace $ACEH$, the target of the indirect branch from block C is block E in the path recording stage, therefore NET builds the trace predicting that the target of block C is always E. However, analysis of the SPEC CPU benchmarks showed that most indirect branches are polymorphic and poorly predicted by a static target predictor, as used by NET. Furthermore, a static indirect branch predictor adds overhead in the case when

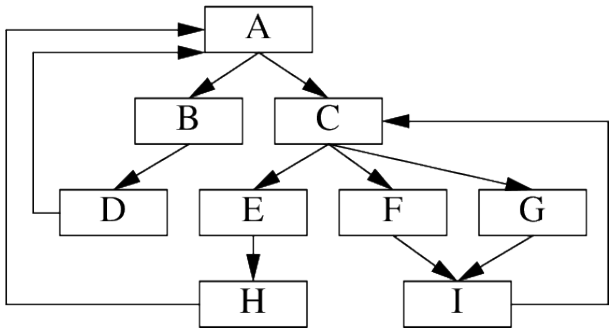


Figure 5.1: Example control flow graph. Each box represents a basic block. Block A, the entry point, contains a conditional direct branch, block C contains an unconditional indirect branch and all other blocks contain unconditional direct branches.

the prediction is incorrect. This analysis is available in Section 5.3.3, which also presents AIBI, a more accurate indirect branch prediction scheme, which has been implemented in MAMBO. To avoid this limitation, the MAMBO trace building scheme terminates on indirect branches, which avoids static target prediction and instead allows their SPC-to-TPC lookup to be implemented using an inline hash table lookup, optionally with AIBI.

5.2.1 Trace heads

The new trace termination condition described in Section 5.2 avoids adding the targets of an indirect branch to a trace tail, by terminating the trace. However, one or more of these targets are likely part of the hot execution path, therefore all targets of indirect branches should have execution counters (i.e. become trace heads) to allow creation of traces. Nevertheless, the NET trace head selection algorithm only instruments the targets of backwards branches and would generally fail to instrument many of these targets. If, for example, block C in the CFG shown in Figure 5.1 is on the hot code path and its indirect branch has a 70% bias toward block E, 30% toward block F and never branches to block G, then both the E and F blocks are also on the hot code path. If these blocks would be trace heads, then the traces *EH...* and *FI...* would be created. However, the trace head selection of NET does not allow this and instead the blocks E, H, F and I could not be trace heads, nor would they be included in trace tails because of the additional termination condition used by MAMBO.

NET also presents an implementation challenge for DBM systems: if a basic block is first reached using a forward branch, then it will be created without an execution counter. However, if it is later reached using a backward branch, then an execution counter has to be added to the existing block or, otherwise a second version of the basic block has to be created. Both options make inefficient use of the code cache space and increase fragmentation. For example, in the control flow graph depicted in Figure 5.1, the first execution of block C would necessarily be a result of the branch from block A, therefore not creating a trace. If block I would execute at a later time, then the backwards branch to C would be discovered and an execution counter would have to be added to the existing block C.

Both of these issues are addressed in MAMBO by a single change to the trace head selection algorithm: whether a basic block is a trace head or not is decided at the time it is created, depending on whether it ends with a direct branch (then it is a trace head) or an indirect branch (then it is a regular basic block). Basic blocks containing an indirect branch are not allowed as trace heads because they would be terminated immediately and would therefore create traces containing a single fragment. This algorithm also allows the targets of indirect branches to be trace heads and avoids ulterior transformation of existing basic blocks into trace heads, by removing the reachability of basic blocks as an input to the trace head selection algorithm. Instead, it relies exclusively on the contents of the basic block itself, which are known at the time it is created. In the example CFG in Figure 5.1, all basic blocks apart from block C (which contains an indirect branch) would be trace heads.

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

In MAMBO, a trace head is implemented as a basic block with a header (shown in Listing 5.2) which: 1) pushes to the stack the contents of 3 scratch registers and of the Link Register, 2) sets the id of the trace head in R0 and 3) calls a shared procedure which then decrements the execution counter of the trace head by one and returns, until it reaches zero. When zero is reached, trace

creation is started, using the id passed to the shared procedure to identify the trace head. The rest of the trace building process is described in Section 5.2.2.

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

Listing 5.2: The code added to trace heads.

5.2.2 Trace building

Trace building works similarly to NET: when a trace is first created, the SPC of the trace head is used to create the first fragment in the trace. Then the first fragment is executed and its selected target is appended to the trace. This iterative process continues until a termination condition is met. The first such condition is the execution of an indirect branch, as previously discussed. An additional condition is the execution of a direct branch to the entry point of an existing trace (including itself), which is intended to limit *tail duplication* between different traces. If a branch to the entry point of an existing trace is encountered, then a direct branch to that trace is inserted and the partial trace is terminated. For example if a trace was created from block A in Figure 5.1, then the trace would initially contain the fragment A. After the fragment A would execute, its target would be appended to the trace. If this target was B, then the partial trace would contain the fragments AB. Since B contains a branch to D, this fragment would also be added to the trace, which would then contain ABD. Finally, the target of the D fragment is A, for which a trace would already exist (the partial trace itself). The ABD trace would be terminated and linked directly to its own entry point.

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

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

Table 5.2: Comparison of MAMBO traces and NET.

5.2.3 Trace size limits

Some code duplication is allowed inside each trace, to encourage partial unrolling of short loops. However, excessive code duplication is undesirable, therefore the maximum number of fragments in each trace is limited. If this configurable limit is reached, the trace is terminated on its next backwards branch. For example in the CFG shown in Figure 5.1, the blocks *CFI* form a loop. If this loop would execute while the trace *ACFICFICFI...* was built, then this would result in an increasingly large trace, which would eventually fill the trace code cache. However, because the maximum number of fragments in a trace is limited, the trace would be terminated on the backward branch from I to C after a limited number of iterations.

5.2.4 Summary

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

indirect branches are shown to generally be polymorphic and poorly predicted by a static target predictor. In this context, several changes to NET are proposed, as shown in Table 5.2, which eliminate static indirect branch prediction while managing the undesired side-effects.

5.3 Indirect branches

Indirect branches are control flow instructions with a target not known at translation time. Looking up TPC for the SPC of indirect branches at runtime is the major source of overhead for DBM systems [KS03b]. We classify indirect branches in three types, as presented in Section 2.4.2:

- function returns, for which we introduce *hardware-assisted return address prediction* in Section 5.3.1; and
- generic indirect branches, handled in MAMBO using inline hash table lookups (Section 4.2.3), for which we introduce the optional *low footprint* optimisation - Section 5.3.2 and *adaptive inlining* - Section 5.3.3; and
- table branches, handled in MAMBO using the *space-efficient shadow branch table linking*, previously described in Section 4.2.2.

Figure 5.2 shows the steps involved in an inline hash table lookup, which is the mechanism used for handling indirect branches in the baseline MAMBO: 1) first, if required and depending on the type of indirect branch, the values of up to three registers are pushed onto the stack to enable their use as scratch registers; then, 2) the SPC is copied or generated in one of the scratch registers; 3) the

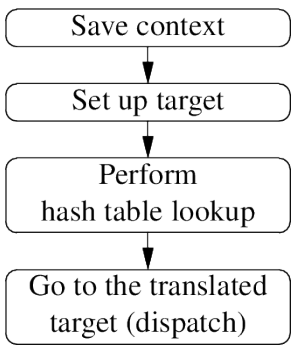


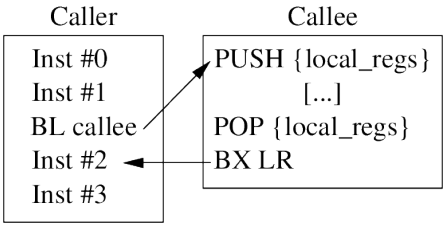
Figure 5.2: Inline hash table lookup.

hash table lookup is performed, with the TPC ending up in one of the scratch registers; 4) finally the values of the scratch registers are restored and a branch to the TPC is performed. The *hardware-assisted return address prediction*, *low footprint inline hash table lookup dispatch* and *adaptive indirect branch inlining* optimisations are all a variation or extension of inline hash table lookups.

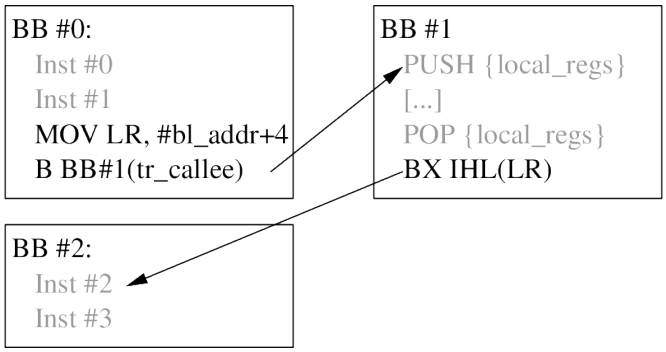
5.3.1 Hardware-assisted return address prediction

Return instructions are the instructions which execute at the end of a procedure (the callee) to return control back to the caller. More specifically, returns target the instruction immediately following the call instruction. Therefore, at the time a call is executed, the target of the first return to execute can be accurately predicted to be the address of the instruction following the call. If nested calls execute, then all predicted addresses can be recorded in a Last In, First Out (LIFO) structure for later use. These properties are used for return address prediction in virtually all modern microprocessors, including by most ARM implementations, which maintain a Return Address Stack (RAS) which is not exposed architecturally [ARM16f, ARM13c, ARM10, ARM16e, ARM13a, ARM14, ARM16a, ARM16b, ARM16c, ARM16d]. However, the translated code generated by a DBM system does not generally maintain these properties because call instructions are translated to regular branches while returns are translated to regular indirect branches. Consequently, hardware return address prediction is not used. Instead, return instructions are predicted by the hardware using the generic indirect branch prediction mechanisms, which are both less accurate and also limited in the number of indirect branches which can be tracked and predicted simultaneously. Since fast return handling is critical for achieving low overhead in DBM systems [KS03b], this limitation is an important contributor to the total overhead.

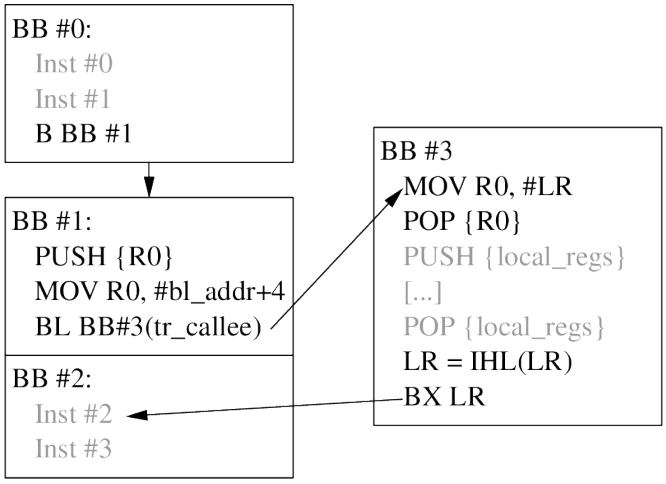
Some of the proposed solutions for optimising returns include: modifying the ISA to allow explicit manipulation of the hardware RAS [KS03b], however this change has not been implemented on general purpose architectures such as x86 or ARM; maintaining a software RAS [HK06], however this is only beneficial on modern microarchitectures if certain transparency guarantees are relaxed (Section 4.2.1), or in the case of DBT when the target architecture provides additional registers which can be easily used as a RAS pointer [dGGL16]. In this context, hardware-assisted return address prediction was developed to allow use of the



(a) The original function call.



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



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

Figure 5.3: Example of a typical function call.

hardware mechanisms for return address prediction, while forgoing the use of a software RAS. Hardware-assisted return address prediction also maintains transparency and is compatible with traces.

Figure 5.3(a) shows a typical function call in ARM code. A *caller* function contains a call (implemented using a Branch-and-Link - BL - instruction) to the entry address of the *callee*. The callee preserves the return address from the Link Register (LR), executes, and then returns to it using a return instruction (implemented using a Branch-and-eXchange - BX - instruction). Because the target address of the return is in a register, this return instruction is an indirect branch.

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

The naive translation of BL and BLX instructions (from the native code in Figure 5.3(a) to Figure 5.3(b)) emulates the call instruction by setting the value of the LR explicitly to the SPC of the instruction following the call and then branches to the translation of the target using a regular (i.e. without link) branch. Similarly, return instructions are translated to an inline hash table lookup (represented by the *IHL()* pseudocode) followed by a regular branch to the TPC of the return address. Therefore, the naive translation of calls and returns is not compatible with the hardware return address predictor, which increases branch mispredictions by 1) translating call-type instructions to regular branch instructions, which do not cause a push on the RAS and by 2) translating return-type instructions to generic indirect branches, which are predicted using the less accurate indirect branch predictor, while also increasing the pressure on the indirect branch predictor. Hardware-assisted return address prediction solves these issues, by modifying the translations as shown in Figure 5.3(c): first, it translates call-type instructions to a sequence which ends with a call-type instruction (*BB #1*), which allows the hardware predictor to push an entry to the RAS. Next, it

inserts the translation of the following instructions, i.e. the predicted return (*BB #2*) immediately after the call, as expected by the predictor. Finally, it modifies the translation of return-type instructions to use a return-type instruction, which will allow the hardware predictor to pop the predicted address from the RAS (*BX LR* in *BB #3*).

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

As an additional optimisation, when a call-type instruction is encountered in the middle of a basic block and a translation does not exist yet, the separate basic block generated for the call-type instruction will be stored in the code cache area immediately following the first basic block, allowing the eliding of the direct branch. For the example in Figure 5.3(c), *BB #1* would be stored immediately after *BB #0*, allowing the elimination of the *B BB#1* instruction from *BB #0*.

5.3.2 Low footprint inline hash table lookup dispatch

The baseline MAMBO implementation of inline hash table lookups has been observed to increase the overhead in some cases by generating an excessive number of data cache and TLB misses. The cause of this behaviour has been tracked down to the implementation of the *dispatch* stage of inline hash table lookups, as shown in Figure 5.2. This stage works by first storing the target address of the branch in a temporary buffer within the code cache fragment itself, then it restores the values of the scratch registers used by the previous stages of the routine and finally it branches to the target by using a PC-relative load instruction

```
// Original instruction: POP {PC}
// The three scratch registers are R4-R6
// and the result of the lookup is in R4
0x00:  .word // TPC reserved word
0x04:  SUBW R5, PC, #12
0x08:  STR  R4, [R5]
0x0C:  POP {R4-R6}
0x10:  ADD SP, SP, #4
0x12:  LDR, PC, [PC, #-24]
```

```
// sa is a pointer to a scratchpad
0x00:  MOVW R6, #(sa & 0xFFFF)
0x04:  MOVT R6, #(sa >> 16)
0x08:  STR R12, [R6]
0x0C:  STR R4, [R6, #4]
0x0E:  MOV R12, R6
0x10:  POP {R4-R6}
0x14:  ADD SP, SP, #4
0x16:  LDM R12, {R12, PC}
```

Listing 5.3: Default (left) and low footprint (right) inline hash table lookup dispatch (Thumb).

with PC as the destination register. This unusual design was imposed by various limitations of the ARM instruction set, which are described in the following paragraphs. The consequence of temporarily storing the 4-byte target address in the code cache fragment, which is not normally accessed via the data path, is that an additional cache line (which is either 32 or 64 bytes in length, depending on the microarchitecture) is fetched in the hardware data cache for each translated indirect branch. This increases the pressure on the hardware data cache, which in some cases results in a higher number of hardware data cache load misses. Similarly, a data TLB entry is used for each page of the software code cache containing the translation of indirect branches, increasing the data TLB pressure. The *low footprint inline hash table lookup dispatch* avoids these issues by modifying the dispatch stage to share a single temporary target buffer between all inline hash table lookups in a thread. This limits the number of hardware data cache lines and TLB entries required for inline hash table lookups to at most one.

On ARM, the target of an indirect branch can be either in a register or in memory. However, if the target is in memory, then the address of the memory location is limited to a +/- 4KiB offset from the base address in any register. The dispatch stage is challenging to implement efficiently because an indirect branch has to be executed while preserving the values of all registers, therefore the target must be loaded from memory, which has the limited offset range. This can be achieved in two ways. The first approach, used by default by MAMBO (shown on the left side of Listing 5.3), is to use a PC-relative load into the PC. However, because the immediate offsets which can be used with load instructions are limited to +/- 4KiB, the TPC has to first be saved (by the *STR* at address 0x08) in a reserved word within the basic block itself (shown at address 0x00). Then, the values of

the scratch registers are restored (by the *POP* instruction at address 0x0C), the stack pointer is incremented by 4 to simulate POP-ing the target address from the stack (at address 0x10) and finally the translated address is loaded in the PC from the temporary location inside the code fragment (by the *LDR* at address 0x12). The second approach is to push the translated return address to the stack and then to POP it directly in the PC. However, instructions of this type are considered function returns by all current ARM microarchitectures, which will attempt to predict their target using the hardware return address stack, which will obviously mispredict since this translation is generated for generic indirect branches. On the microarchitectures we have evaluated, the penalty for these branch mispredictions tends to be high enough to undo any performance benefits due to the lower number of instructions used by this implementation. This second approach is therefore only used for translated return instructions when the hardware-assisted return address prediction is enabled - see Section 5.3.1).

However, the PC-relative dispatch is still suboptimal on modern microarchitectures as previously discussed : because the target address is temporarily saved in the code cache fragment (which is not normally accessed through the data path), a whole 32 or 64 bytes (depending on implementation) data cache line is going to be mostly wasted since it will only contain 4 bytes of useful data, possibly causing the eviction of live cache lines; and similarly, a data TLB entry will be used by each page containing code with inline hash table lookups. This motivated the development of the *low footprint inline hash table lookup dispatch*.

Low footprint inline hash table lookup dispatch uses a thread private scratchpad for all inline hash table lookups, therefore reducing the total number of required data cache lines and data TLB entries to one. The implementation is shown in the right hand column of Listing 5.3. Initially, the address of the shared scratchpad is generated in one of the scratch registers (using the instructions MOVW and MOVT at address 0x00 and 0x04). Then, the value of another application register (R12 in this case) is spilled to offset 0 of the scratchpad (by the *STR* at address 0x08) and the TPC is saved at offset 4 of the scratchpad (by the *STR* at address 0x0C). Next, the pointer to the scratchpad is copied from the initial scratch register to the one whose value was spilled to the scratchpad (by the *MOV* at address 0x0E). Restoring the values of the first set of scratch registers proceeds as before. Finally, a load-multiple instruction (at address 0x16) is used to atomically load both the spilled value of the last scratch register and the

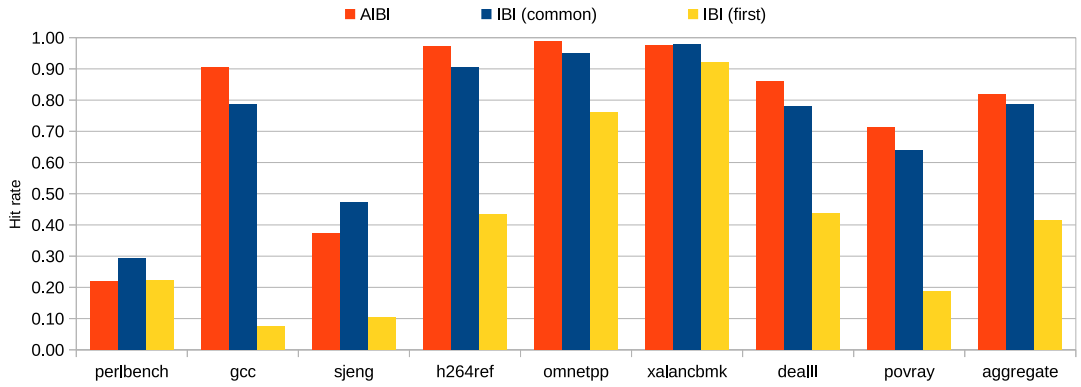


Figure 5.4: Comparison of hit rates on a selection of SPEC CPU2006 benchmarks for indirect branch predictors.

translated target into the PC. In effect, instead of loading the target into the PC while preserving the values in the general purpose registers, this works by using a scratch register and restoring its value at the same time as branching to the destination.

A limitation of this optimisation is its limited portability: in the AArch64 state, it is no longer possible to directly load the target address of an indirect branch from memory to the PC and therefore this technique cannot be implemented.

5.3.3 Adaptive indirect branch inlining

Indirect branches have dynamic targets, which are not known at translation time. Due to their nature, the translated indirect branches must perform a SPC to TPC lookup every time they execute. This lookup represents a major source of overhead for DBM systems [KS03b, HWH+07]. The baseline version of MAMBO and other DBM systems such as DynamoRIO attempt to reduce this overhead by generating a highly optimised inlined hash table lookup routine for each translated indirect branch. This approach allows the hardware branch predictors to handle separately each translated indirect branch (improving hardware prediction rates) and minimises the length of the critical path compared to a shared routine by taking advantage of the available dead registers, on a case-by-case basis. However, the hash table lookup operation inherently requires a number of additional instructions, including memory loads and conditional branches. Other DBM systems, such as Pin for ARM, use Indirect Branch Inlining (IBI), which consists of

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

Listing 5.4: The implementation of AIBI. Rs0 to Rsn are scratch registers, while Rtarget is the register which initially contains the target address (SPC).

a compare-and-branch chain which compares the current target address against a configurable number of previous targets, using only the code path (i.e. by using immediates). However, previous attempts to use this prediction scheme in MAMBO have failed to improve performance, due to the high overhead associated with updating the predicted target, the poor hit rate due to the polymorphic nature of indirect branches and the high penalty of hardware branch mispredictions triggered in the relatively common case when one or more predictions at the top of the chain miss. On the other hand, we designed the *Adaptive Indirect Branch Inlining* (AIBI) scheme to allow quick updating of the predicted address every time it misses, while still having a shorter critical path than the inline hash table lookup. This is similar to the way indirect branch target prediction works in most hardware implementations.

Figure 5.4 compares the hit rates for three indirect branch predictions schemes: *AIBI*, which always predicts the address of the most recent target; *IBI (common)*, which is a static predictor which predicts the most common target for each branch using post-mortem information; and *IBI (first)* which predicts the address of the first target seen for each branch. The selected benchmarks are those which execute a relatively high number of generic indirect branches. The *aggregate* bars show the hit rates when considering together all indirect branch executions from the selected benchmarks. IBI (common) shows the upper bound for a static predictor and since the information to choose the most common target is not available at

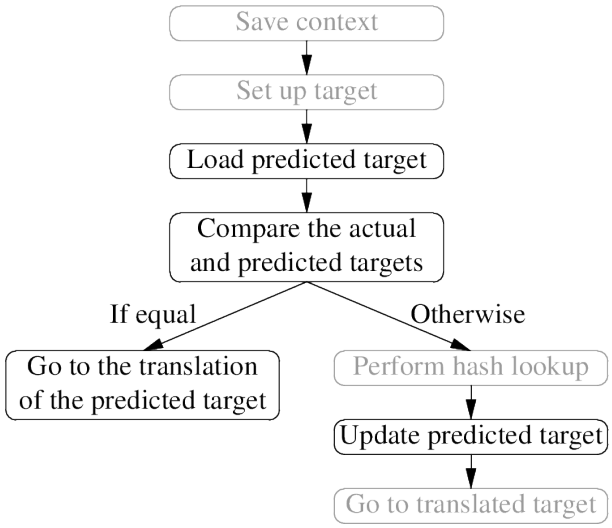


Figure 5.5: Adaptive indirect branch inlining.

runtime, practical IBI implementations will almost always have lower hit rates. The IBI (first) hit rate is more relevant for practical IBI implementations, which can either predict the target of the n-th execution of an indirect branch, or, alternatively, can profile the first few executions of the branch and predict the most common target among those samples. It can be observed that the hit rate for AIBI is generally similar to that of the IBI (common) predictor and for most benchmarks and overall, slightly better. On the other hand, the hit rate for IBI (first) is generally much lower, which indicates that IBI implementations will tend to have significantly lower hit rates than AIBI.

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

The diagram in Figure 5.5 shows how AIBI works, where the black boxes show the additional steps added specifically for AIBI, while the grey boxes show the unmodified steps which are part of the inline hash table lookup routine (which is shown separately in Figure 5.2). Listing 5.4 shows the implementation of AIBI. With AIBI, after the target address has been generated or loaded in a register, the predicted SPC is loaded using a single PC-relative load instruction and then two addresses are compared, as shown in the *check_pred* procedure. The comparison is implemented using a subtract instruction (SUB) and a Compare and Branch on NonZero (CBNZ) instruction to preserve the flags in the ARM Program Status Register (PSR). In case of a match, the context is restored and execution branches to the predicted TPC using a second PC-relative load, as shown in the *b_pred* procedure. Otherwise, in case of a miss, the regular inline hash table lookup proceeds, with the difference that after the hash table lookup has been performed, but before branching to the destination, the predicted SPC and TPC are updated, as shown in the *fallback* procedure. PC-relative stores are not allowed in the Thumb mode, therefore the address where the predicted SPC and TPC are stored is first generated using a subtract instruction.

AIBI is similar in predicting the address of the most recent target to the MRU IBI prediction scheme proposed by Dhanasekaran and Hazelwood [DH11b], which was presented on page 38, Section 2.4.2. However, while the MRU scheme is used in *addition* to IBI, AIBI is an alternative to IBI. When the MRU prediction misses, it falls back to IBI, while AIBI falls back to an inline hash table lookup. MRU updates the predicted address from the IBI target fragments, while AIBI updates the predicted address in the inline hash table lookup. Furthermore, AIBI as implemented in MAMBO is effective in reducing the overhead on all systems used in the evaluation (Section 5.4), while MRU as prototyped for Pin failed to improve performance on average. Unfortunately, insufficient information is available to determine why. The MRU publication explains that its dynamic instruction count was higher than that of standard IBI despite the increased prediction hit rate. However, on ARM platforms we have observed that the hardware branch prediction rate and other microarchitectural events often have a stronger effect than relatively small changes in the number of executed instructions. For example, when we have unsuccessfully tried to use IBI in MAMBO, the dynamic instruction count was significantly reduced, however the overhead was increased because of the hardware branch mispredictions introduced by the IBI chain. This

could indicate that 1) existing x86 implementations can predict IBI chains better than ARM implementations or, less likely, 2) that branch mispredictions are relatively cheaper on x86 implementations than on ARM implementations. Another possible explanation is that the performance of MRU was affected by the mechanism used to update the predicted address, which it duplicates across every target fragment linked by the IBI chain and whose details are not presented in the publication.

5.3.4 Huge pages

Translation Lookaside Buffers (TLBs) are small, fast buffers used by the processor to cache the virtual to physical address mapping for a limited number of active memory pages. For some combinations of workloads and TLB size, the miss rate for TLBs has been observed to increase excessively under MAMBO. ARM implementations use separate data and instruction TLBs and increased miss rates have been observed in both types. The data TLB is under additional pressure because of the hash table used to dynamically map the SPC to the TPC for indirect branches, which becomes part of the working set under MAMBO. The instruction TLB tends to be more heavily affected, because the size of the code in the code cache is affected by multiple factors: code expansion, caused by the additional instructions added to provide transparency; code duplication, caused by tail duplication both in traces and basic blocks; and fragmentation in the code cache, which spreads code fragments across a higher number of pages compared to the native code.

Most caches used by ARM implementations are Physically Indexed and Physically Tagged (PIPT), which require a virtual-to-physical address translation for every cache access. Therefore, the latency introduced by this translation has a direct impact on the latency of every single memory operation. The TLBs speed up the translation compared to performing a *page table walk* for every memory access. As a consequence, the maximum size of data which can be accessed with low latency is bound by the size of the TLB (the maximum number of entries it can store simultaneously) multiplied by the size of a page. However, if the data is fragmented across multiple pages, with poor temporal locality, the effective capacity will be proportionally reduced. A solution introduced by hardware designers was to allow the use of larger pages. For example on ARM, where

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

Table 5.3: Overview of the systems used for evaluation.

the standard page size is 4KiB, support for *huge pages* (2 MiB) has also been introduced.

To address the issue of increased TLB pressure resulting in high TLB miss rates under MAMBO, the option to request huge pages has been implemented as an optional feature. Huge pages are used both for the code cache and metadata (including the SPC-to-TPC hash table) and are requested using the standard interface provided by the *mmap* Linux system call. These structures were already stored in memory in large contiguous allocations, therefore no modifications to their internal organisation were required. This optional feature does not affect the memory allocations of the application itself, which can either use regular or huge pages. This optimisation is evaluated in Section 5.4.4.

5.4 Evaluation

5.4.1 Experimental setup

Three systems (ODROID-XU3, Tronsmart R28 and APM X-C1) have been added to the two used in the evaluation of Chapter 4. This was motivated by the dependence of the optimisations evaluated in this chapter on the microarchitectural features of the processor. The five systems were selected to cover a wide range of the commercially available ARM implementations. Table 5.3 describes their

[1]The specifications for ODROID-XU3 apply to the *LITTLE* cluster only. The *big* cluster is not used for this evaluation because it uses Cortex-A15 cores, the same as the Jetson TK1 system.

microarchitectures. All systems use a modified Harvard architecture, with separate 32 KiB L1 data caches and 32 KiB L1 instruction caches, and separate data and instruction L1 TLBs as described in Table 5.3. Higher level caches and TLBs are unified. The *IB predictor* row describes the hardware indirect branch prediction scheme: *previous* means that the address of the previous target of the instruction is predicted, while *adaptive* means that multiple target addresses are allowed per branch. Note that Cortex-A7 is documented not to predict the target for branches implemented as loads or data processing operations with PC as the destination, which are used to implement most indirect branches translated by MAMBO.

All systems are running Ubuntu 14.04 LTS with the Linux kernel version supported by the manufacturer: 3.8 for ODROID-X2, 3.10 for ODROID-XU3, Tronsmart R28 and Jetson TK1 and 4.2 for APM X-C1. SPEC CPU2006 has been compiled with GCC 4.6.3, configured to generate Thumb-2 code (the default configuration) for the *armhf* architecture using the *-O2* optimisation level and the executables were statically linked. Power management features such as DVFS and core offlining were disabled. The ODROID-XU3 system uses a heterogeneous *big.LITTLE* [ARM13b] configuration, with a *LITTLE* Cortex-A7 cluster, which was used for this evaluation and a *big* Cortex-A15 cluster which was not used in this evaluation because the same microarchitecture is used on the Jetson TK1 system.

The libquantum benchmark from the SPEC CPU2006 suite has been disabled because it fails to complete, both when executed natively and under MAMBO. All other CPU2006 benchmarks are enabled and produce the expected output. All SPEC CPU2006 results were obtained using the ref data set.

Multiple MAMBO *configurations* have been benchmarked. A configuration is a build of MAMBO with a specific set of enabled optimisations. The configuration with an empty set of optional optimisations enabled is called the *baseline* configuration. This is similar to the MAMBO configuration used in Section 4.4, with the exception that the *low overhead return address prediction*, which is a return address prediction scheme based on a software RAS, has been discontinued because it is incompatible with traces and therefore it is never used in this evaluation. *Hardware-assisted return address prediction*, introduced in this publication, serves a similar role while maintaining full transparency. All other configurations

are named *+<name of optimisation 0> ... +<name of optimisation n>*, for example the configuration with *hw_ras* and *traces* enabled is called *+hw_ras+traces*. The following optional optimisations have been evaluated:

- *traces* - code cache traces; and
- *hw_ras* - hardware return address prediction; and
- *hugetlb* - the code cache and metadata are allocated using huge pages; and
- *aibi* - adaptive indirect branch inlining; and
- *ldm_pc_sr* - low footprint inline hash table lookup dispatch.

5.4.2 Trace creation threshold

As described in Section 5.2, traces are created when a trace head reaches a predefined execution count threshold. For relatively long running tasks, such as SPEC CPU, minimising the trace creation latency by reducing this threshold is not critical. However, the value of the threshold affects which trace tail is recorded and then used as the path of the trace. Therefore, it is desirable to use a threshold which makes it more likely to record a typical execution of each loop rather than special cases such as the first or last iteration. This property is mostly specific to the application and input data, however some numbers are possibly a better choice on average. A second consideration is the size of the trace cache: for low thresholds, more traces are created, which can end up filling the limited size code cache and require flushing it, an expensive operation. To evaluate the impact of the trace creation threshold, *perlbench*, the SPEC CPU2006 benchmark which appears to be the most sensitive to the value of this parameter, was executed under MAMBO using the *+traces* configuration and a varying trace creation threshold, between 2 and 256.

The results of this evaluation are shown in Figure 5.6. This chart shows the size of the trace code size (lower is better) as the blue line (using the scale on the primary Y axis on the left) and the relative execution time (lower is better) as the red line (using the scale on the secondary Y axis on the right). As it can be observed, the relative execution time varies between around 1.7 and 1.9 without a clear pattern being noticeable. The thresholds which generate faster code are generally expected to be specific to *perlbench* and the *ref* input set, rather

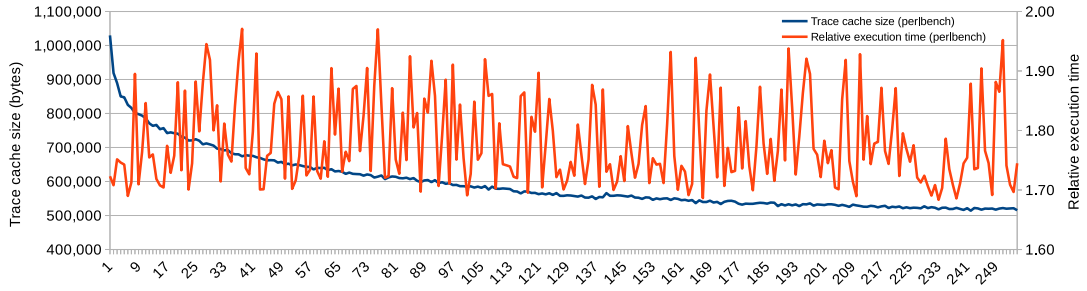


Figure 5.6: Trace creation threshold vs relative execution time and trace code size

System	MAMBO configuration	SPECint	SPECfp	SPEC CPU
ODROID-XU3 (LITTLE)	baseline	1.55	1.11	1.26
	+traces	1.41	1.10	1.21
	+hugetlb +hw_ras +traces	1.35	1.09	1.19
ODROID-X2	baseline	1.61	1.13	1.30
	+traces	1.33	1.07	1.17
	+aibi +traces	1.31	1.06	1.15
Tronsmart R28	baseline	1.60	1.12	1.29
	+traces	1.31	1.09	1.17
	+aibi +traces	1.29	1.08	1.16
Jetson TK1	baseline	1.71	1.16	1.35
	+traces	1.44	1.11	1.23
	+hw_ras +traces	1.38	1.11	1.21
APM X-C1	baseline	1.59	1.09	1.26
	+traces	1.34	1.07	1.17
	+hugetlb +hw_ras +traces	1.21	1.05	1.11

Table 5.4: The MAMBO baseline, *+traces* and the configuration with the lowest overhead for SPEC CPU2006 on each system.

than more generally applicable. The size of the trace cache, however, decreases following a logarithmic curve, tapering off around a threshold of 230. For the rest of the experiments in this evaluation, the trace creation threshold was set to 256 (the maximum allowed by the implementation because it maintains this counter in a byte), which appears to minimise the code size while not affecting the performance.

5.4.3 Overall performance

Table 5.4 summarises the overall performance of the baseline, *+traces* and of the optimal MAMBO configuration for each system (when running SPEC CPU2006). Appendix B includes the full results showing the performance of each MAMBO

configuration on each system. The reported value is the geometric mean of execution time relative to native execution for each benchmark. It can be observed that between the five test systems, three unique MAMBO configurations are needed to achieve the lowest possible overhead. This hints that, as expected, some of the optimisations have varying effectiveness depending on the microarchitecture. This behaviour is examined in detail in Section 5.4.4. Another related observation is that the spread of the average overhead between the microarchitectures is quite high: from only 11% on APM X-C1, up to 21% on Jetson TK1, which further underlines the impact of microarchitecture on the performance of DBM systems. The SPECint benchmarks run with higher overhead than the SPECfp benchmarks because they tend to be control (as opposed to data) bound.

The *traces* optimisation has by far the largest overall effect. This is the expected result, as improved software code cache locality and a reduced number of executed branches reduce the overhead 1) for most benchmarks and 2) on all microarchitectures. While the geometric mean overhead is generally reduced only by a few points for the other optimisations, this is in large part due to these optimisations targeting only specific types of workloads. For example, the *hw_ras* optimisation reduces the overhead of *gobmk* on Jetson TK1 from 94% to 67%, however, because only a few benchmarks gain a speed-up, the geometric mean overhead is only decreasing from 23% to 21%. By running the optimal configuration on each system, the geometric mean overhead is reduced compared to the *baseline* configuration by 29% on ODROID-XU3, 49% on ODROID-X2, 46% on Tronsmart R28, 41% on Jetson TK1 and 58% on APM X-C1.

5.4.4 Performance counter analysis

The benchmarks which have significant overhead under the baseline MAMBO configuration or whose overhead is changed by enabling various MAMBO optimisations have been executed under the *perf stat* tool, which uses hardware performance counters to report the total number of times various performance-related events have occurred. Note that some of the reported events occur as a result of *architectural execution* (i.e. code executed by the application), while others occur as a result of *speculative execution* (both architecturally executed code and instructions executed without being committed, as a result of mis-speculation). The following metrics are reported:

- Insts - instructions architecturally executed
- L1d-l - loads by the core from the L1 data cache
- L1d-m - misses in the L1 data cache
- L1i-l - loads by the core from the L1 instruction cache
- L1i-m - misses in the L1 instruction cache
- L2-l - loads from the unified L2 cache
- L2-m - misses in the unified L2 cache
- dtlb-m - L1 data TLB misses
- itlb-m - L1 instruction TLB misses
- br - architecturally executed branches
- br-m - mispredicted or not predicted branches speculatively executed

For the baseline configuration, the values are relative to native execution (i.e. a value of 2 means that a certain event has occurred twice as many times compared to native execution). All other results show the effect of enabling one additional optimisation compared to the baseline or to another MAMBO configuration. For example, Table 5.14 shows the effect of enabling the *traces* optimisation and shows results relative to the baseline configuration, while Table 5.19 shows the effect of enabling the *hw_ras* optimisation and shows results relative to the *+traces* configuration. Additionally, the raw performance counter values for the *native* execution are provided in Appendix C. These can be used to calculate the hit and miss rates for caches, TLBs and the branch predictor, for example.

Some events cannot be monitored on all systems: the *L1i-l*, *L2-l* and *L2-m* events are unavailable on the ODROID-X2, while the *br* event is not available on the APM X-C1. Furthermore, the *br-m* event on APM X-C1 does not appear to work as expected and has been discounted from this analysis. On APM X-C1, the counter for this event has values of similar magnitude and strongly correlated with the *Cycles* counter. Additionally, Cortex-A9 (used by ODROID-X2) does not implement the *INST_RETIRED* event, used on the other systems for the *Insts* counter and instead the *Instructions coming out of the core renaming*

stage event is used, which is described by ARM as providing *similar functionality*. This accounts for the slight difference between the values of the *Insts* counter on ODROID-X2 compared to the other systems. Cortex-A15 (used by the Jetson TK1 system) does not implement the architecturally executed software change of PC event (*PC_WRITE_RETIRED*), used on the other systems for the *branches* counter and instead the speculatively executed software change of PC event (*PC_WRITE_SPEC*) is used.

How to read and interpret the results

The tables provided in this section contain a large amount of data. This section will provide guidance on how to read it and interpret the results. Let us consider Table 5.9 as the first example, which shows the relative performance counter values for the *baseline* configuration of MAMBO compared to native execution, on APM X-C1.

The first column to inspect is *Cycles*, which shows the relative change in the number of execution cycles compared to native execution. For these compute-bound workloads, it is equivalent to the slowdown in terms of time for the userspace code (i.e. excluding time spent in system calls). In this example, the *perlbench* benchmark takes 1.98 times as long to execute the userspace code (i.e. it takes 98% longer), *tonto* takes 1.18 times as long and so on. The second column to inspect is *Insts*, which shows the relative number of executed instructions. *perlbench* executes 53% more instructions, while *tonto* executes 28% more instructions. Comparing the two benchmarks against each other, it can be observed that *perlbench* executes 53% more instructions while taking 98% longer and *tonto* executes 28% more instructions while only taking 18% longer. This indicates that the translation of *tonto* executes relatively efficiently on this microarchitecture and much of the introduced overhead can be attributed to the additional executed instructions. However, some of the other performance metrics for *tonto* are degraded and are likely to be responsible for some of this overhead. For example, the number of L1 instruction load misses (*L1i-m*) has increased 35 times and the number of instruction TLB misses (*itlb-m*) has increased 3 times, which is a result of code expansion, duplication and fragmentation in the code cache. This behaviour is likely to be improved by traces, which reduce fragmentation and improve code cache locality. Another degraded metric is the number of L1 data cache loads (*L1d-l*), which has increased by 14%, likely as a result of accesses to

the SPC-to-TPC hash table. However, it can be observed that the L1 data cache has been able to accommodate the increased working set size fairly well, as the number of L1 data cache load misses (*L1d-m*) has only increased by 3%. As a result of the increased number of L1 cache misses, the number of L2 cache loads (*L2-l*) has increased by 37%. This increased pressure on the L2 cache results in 20% more load misses (*L2-m*). Finally, the larger working set for data also slightly increases the number of data TLB misses (*dtlb-m*) by 18%.

Continuing to analyse the *tonto* benchmark, Table 5.14 shows the relative change in performance related events when the *traces* optimisation is enabled, compared to the baseline configuration of MAMBO. In this case, the *Cycles* column shows a 6% speed-up, with no change in the number of executed instructions. This performance improvement can be attributed to the 78% reduction in L1 instruction load misses, which also lowers the pressure on the L2 cache, reducing the number of L2 loads and L2 loads misses by 8% each. Enabling traces also reduces the number of executed branches by 42% (*br*). The number of executed branches is not reported by the APM X-C1 system, however, as it is an architectural event, its value is the same on all systems and can therefore be obtained from one of the ODROID-XU3, ODROID-X2 or Tronsmart R28 tables (Tables 5.10 to 5.12).

Baseline MAMBO

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	2.11	1.53	1.19	1.93	1.49	8.80	4.39	3.32	6.70	17.45	2.14	1.24
gcc	1.59	1.41	1.28	1.23	1.41	10.49	2.38	1.36	2.99	17.12	1.60	1.24
gobmk	2.35	1.38	1.13	1.84	0.84	12.71	6.14	4.62	10.98	45.77	1.85	0.87
sjeng	2.11	1.47	1.25	1.93	1.23	23.39	7.95	1.91	5.27	63.04	2.06	1.01
h264ref	1.55	1.40	1.28	1.52	1.47	11.59	2.42	1.30	2.54	12.26	1.49	1.21
omnetpp	1.69	1.90	1.73	1.39	1.25	45.81	3.29	1.13	2.24	7.50	2.06	1.21
astar	1.28	1.48	1.30	1.02	1.51	1.84	1.03	1.03	1.02	13.07	1.97	1.07
xalancbmk	1.95	1.80	1.78	1.89	1.52	27.54	3.56	1.29	2.36	10.11	2.53	1.28
dealII	1.36	1.46	1.46	1.04	1.57	9.43	1.27	1.03	3.37	7.47	1.48	1.22
povray	1.81	1.75	1.55	1.62	1.47	5.57	2.48	8.32	9.38	9.52	1.66	0.98
tonto	1.27	1.28	1.16	1.06	1.20	16.58	2.26	1.09	3.57	17.03	1.66	1.13

Table 5.5: Overhead under the baseline MAMBO configuration, on ODROID-XU3 (LITTLE cluster).

Tables 5.5 to 5.9 show the overhead of running SPEC CPU2006 under the baseline configuration of MAMBO compared to native execution, for benchmarks with an execution time overhead of at least 15%.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	2.13	1.62	1.20	1.70	N/A	17.68	N/A	N/A	2.76	8.18	2.12	3.57
gcc	1.95	1.51	1.39	1.15	N/A	25.50	N/A	N/A	2.09	8.99	1.64	3.50
gobmk	2.41	1.39	1.08	1.58	N/A	31.78	N/A	N/A	2.48	408.17	1.86	2.38
sjeng	2.17	1.42	1.19	1.81	N/A	97.81	N/A	N/A	2.51	1346.68	2.06	2.05
h264ref	1.52	1.47	1.31	1.31	N/A	42.07	N/A	N/A	2.60	65.52	1.49	3.99
omnetpp	1.94	2.16	1.71	1.28	N/A	322.95	N/A	N/A	1.97	41.19	2.06	4.37
astar	1.18	1.38	1.26	1.03	N/A	7.18	N/A	N/A	1.07	0.69	1.95	1.45
xalancbmk	2.19	2.00	1.81	1.57	N/A	44.00	N/A	N/A	2.09	5.74	2.53	3.90
milc	1.25	1.20	1.13	1.01	N/A	69.31	N/A	N/A	1.19	0.99	1.48	9.72
dealII	1.31	1.49	1.47	1.02	N/A	54.21	N/A	N/A	1.16	9.28	1.49	2.00
povray	1.65	1.78	1.56	1.50	N/A	13.65	N/A	N/A	2.10	3.10	1.66	2.22
tonto	1.44	1.35	1.19	1.02	N/A	47.53	N/A	N/A	1.65	12.22	1.69	4.05
wrf	1.21	1.18	1.14	1.02	N/A	94.11	N/A	N/A	1.81	5.61	1.50	3.36

Table 5.6: Overhead under the baseline MAMBO configuration, on ODROID-X2.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	1.95	1.53	1.19	1.69	1.58	20.43	7.14	2.32	2.61	20.99	2.12	1.93
gcc	1.66	1.42	1.40	1.20	1.62	32.62	3.94	1.19	2.30	18.83	1.63	2.58
gobmk	2.58	1.37	1.10	1.59	1.12	35.43	12.56	3.11	2.28	636.50	1.86	2.05
sjeng	2.36	1.47	1.23	1.89	1.52	127.62	19.39	1.56	1.93	117218.94	2.06	2.32
h264ref	1.62	1.40	1.28	1.44	1.88	47.39	2.77	1.49	2.13	164.69	1.49	3.23
omnetpp	1.77	1.89	1.65	1.36	1.83	427.80	5.35	1.15	1.71	174.81	2.06	4.44
xalancbmk	2.28	1.79	1.66	1.79	2.18	81.49	5.33	1.29	2.20	7.46	2.52	4.13
gamess	1.16	1.10	1.03	1.07	1.14	43.93	2.38	1.06	1.14	933.69	1.90	1.62
dealII	1.40	1.46	1.33	1.03	1.44	85.67	1.25	1.02	1.58	21.70	1.49	1.69
povray	1.79	1.73	1.55	1.55	1.56	13.66	3.53	2.97	1.96	4.43	1.66	2.42
tonto	1.31	1.28	1.16	1.03	1.49	57.96	2.11	1.11	1.47	20.98	1.68	2.84

Table 5.7: Overhead under the baseline MAMBO configuration, on Tronsmart R28.

Overall, the metrics measured by the performance counters for this subset of benchmarks are worse compared to native execution and generally multiple factors appear to affect the performance of translated code. The number of instruction cache loads is, however, reduced for multiple benchmarks and systems, despite an increased number of executed instructions: for *gobmk* on ODROID-XU3 it is reduced by 26%, for *leslie3d* on Jetson TK1 by 9%, and for *gobmk*, *h264ref*, *astar*, *dealII* and *tonto* on APM X-C1 by 15%, 6%, 21%, 19% and 9% respectively. This indicates that even a relatively simple dynamic code cache has the potential to improve (hardware) instruction cache spatial locality compared to static compilers and linkers. The number of branch mispredictions for *gobmk* on ODROID-XU3 is reduced by 13%, despite the number of branches increasing by 85%. This is likely enabled by the software code cache improving prediction for conditional direct branches.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	2.41	1.54	1.21	1.66	2.02	10.51	5.78	1.84	2.81	13.39	1.34	2.89
gcc	1.86	1.42	1.37	1.20	1.99	16.11	3.12	1.10	2.17	7.09	1.26	2.49
gobmk	2.75	1.37	1.05	1.54	1.43	16.26	7.06	3.13	2.52	86.95	1.29	1.44
sjeng	2.47	1.47	1.17	2.27	1.70	25.24	8.60	1.27	2.10	125.51	1.35	1.85
h264ref	1.67	1.40	1.23	1.37	1.83	17.23	2.30	2.39	2.35	19.61	1.35	3.21
omnetpp	1.89	1.90	1.62	1.52	2.68	50.02	4.75	1.32	1.78	55.37	1.39	3.60
xalancbmk	2.48	1.81	1.61	1.85	3.55	26.76	4.11	1.19	2.10	10.29	1.47	4.68
gamess	1.19	1.10	1.01	0.96	1.16	21.67	1.83	1.54	1.14	11.37	1.47	1.46
milc	1.25	1.19	1.08	1.00	1.17	7.78	1.00	0.94	1.04	0.86	1.33	1.70
leslie3d	0.76	1.07	0.97	0.54	0.91	1.37	0.71	1.00	0.99	0.86	1.11	0.98
dealII	1.38	1.46	1.29	1.03	1.51	19.64	1.24	1.01	1.12	5.42	1.20	1.32
soplex	1.31	1.25	1.07	1.01	1.24	11.20	1.00	1.00	1.10	2.22	1.29	1.18
povray	2.03	1.75	1.45	1.41	2.01	7.43	2.71	8.26	2.19	3.24	1.34	2.77
tonto	1.36	1.28	1.13	1.01	1.47	36.68	2.60	1.07	1.43	6.20	1.27	1.68
sphinx3	1.49	1.11	1.03	1.02	1.22	3.77	0.86	0.95	1.74	2.37	1.21	1.21

Table 5.8: Overhead under the baseline MAMBO configuration, on Jetson TK1.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	1.98	1.53	1.19	1.44	1.32	10.64	1.56	5.24	2.45	15.46	N/A	N/A
gcc	1.76	1.41	1.24	1.14	1.20	19.60	1.59	1.69	1.07	8.01	N/A	N/A
gobmk	2.65	1.37	1.04	1.87	0.85	22.98	2.18	9.75	2.16	16.90	N/A	N/A
sjeng	2.04	1.46	1.20	2.12	1.05	196.43	2.43	5.90	1.03	3.98	N/A	N/A
h264ref	1.36	1.40	1.28	1.36	0.94	43.37	1.55	1.48	1.26	4.70	N/A	N/A
omnetpp	1.77	1.90	1.64	1.14	1.18	429.36	2.84	1.22	1.04	2.26	N/A	N/A
astar	1.25	1.48	1.17	1.02	0.79	2.92	1.47	1.00	0.99	3.00	N/A	N/A
xalancbmk	2.57	1.80	1.78	1.71	1.42	74.64	3.94	1.84	1.18	4.70	N/A	N/A
dealII	1.30	1.46	1.50	1.02	0.81	61.49	2.23	1.05	1.02	2.51	N/A	N/A
povray	1.91	1.75	1.44	1.46	1.12	15.35	2.28	17.82	38.87	25.12	N/A	N/A
tonto	1.18	1.28	1.14	1.03	0.91	35.84	1.37	1.20	1.18	3.07	N/A	N/A

Table 5.9: Overhead under the baseline MAMBO configuration, on APM X-C1.

The benchmark with the highest execution time overhead on all systems is *gobmk*, with an overhead between 135% (on ODROID-XU3) and 175% (on Jetson TK1). The instruction count overhead under MAMBO is only around 37%, therefore there are additional causes for the high slowdown factor. Natively, *gobmk* is the benchmark with the highest rate of instruction cache misses (per cycle). When running under the baseline MAMBO, the number of instruction cache misses increases from 12 times on ODROID-XU3 up to 35 times on Tronsmart R28. The instruction cache misses lead to an increased number of L2 loads (from 2 times more on APM X-C1 up to 12 times on Tronsmart R28) and L2 load misses (from 3 times more on Tronsmart R28 up to 9 times more on APM X-C1). The number of instruction TLB misses also increases significantly, from 16.9 times on APM X-C1 and up to 636 times on Tronsmart R28. This behaviour appears to be primarily caused by the fragmented layout of basic blocks in the code cache, which decreases instruction cache locality, and spreads the

code across more pages. Additionally, the increased number of branches (1.86 times) and increased branch mispredictions (except on ODROID-XU3, from 1.44 more on Jetson TK1 up to 2.38 on ODROID-X2) likely affect hardware code prefetching.

leslie3d on Jetson TK1 gets a 24% speedup compared to native execution. This appears to be caused by improved cache hit rates. The 9% reduction in the instruction cache misses and the 14% reduction in instruction TLB misses are likely caused by the software code cache with unconditional branch eliding improving locality. However, the more significant 46% reduction in data cache misses appears to be a fortunate side effect of the memory layout of applications running under MAMBO being slightly altered because of the memory region reserved for the runtime. The layout of application data is not modified on purpose.

The overhead of *h264ref* on APM X-C1 is significantly lower compared to the other systems (36% on APM X-C1 and between 52% and 67% on the other systems). On APM X-C1, the number of L1 instruction cache loads under MAMBO is reduced by 6% from native execution, as opposed to the other systems on which it is increased from 43% (on ODROID-XU3) to 88% (Jetson TK1). The overhead on APM X-C1 in terms of L2 loads (55%) and L1 data TLB misses (26%) is also lower compared to the other systems (between 139% and 177% and between 113% and 160% respectively). Similarly, *tonto* running on APM X-C1 has a lower overhead (18%) compared to the other systems (27% - 44%), while having fewer L1 instruction cache loads and lower overhead in terms of L2 cache loads and L1 data TLB misses. Since the Jetson TK1 and Tronsmart R28 systems have the same instruction cache line length as the APM X-C1, it is not immediately clear why the benchmarks running on APM X-C1 have fewer L1 instruction loads. Unfortunately, the microarchitecture of APM883208 is not documented in detail, therefore further understanding of this behaviour is difficult. This difference could potentially be caused by a large loop buffer on the APM X-C1 system. The Cortex-A15 core used by the Jetson TK1 system is documented to have a 32 entry loop buffer [Lan11], while the other Cortex-A cores used in the other systems are not documented to use a loop buffer at all.

The effect of some of the microarchitectural parameters are clearly shown in the performance counter results. For example, the overhead in terms of L2 cache load misses is inversely correlated with the size of the L2 cache: taking perlbench

as an example, the L2 load miss overhead is 84% on Jetson TK1 (2MiB L2 cache), 132% on Tronsmart R28 (1MiB L2 cache), 232% on ODROID-XU3 (512 KiB L2 cache) and 424% on APM X-C1 (256KiB L2 cache).

Traces

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	0.82	0.99	1.00	1.04	0.97	0.33	0.53	0.51	1.01	1.13	0.49	1.26
bzip2	1.06	1.01	1.00	1.02	0.84	1.11	1.02	1.11	1.00	0.83	0.48	1.05
gcc	0.90	1.03	1.00	1.05	0.99	0.48	0.73	0.82	1.00	1.29	0.67	0.96
gobmk	0.74	0.99	1.00	0.99	1.15	0.60	0.67	0.46	1.46	1.60	0.57	1.18
sjeng	0.83	1.00	1.01	1.01	1.02	0.51	0.60	0.75	1.24	1.45	0.60	1.08
h264ref	0.97	1.01	1.01	1.02	1.11	0.56	0.82	0.95	0.96	1.29	0.76	0.97
omnetpp	0.84	0.97	1.00	1.04	1.36	0.31	0.60	0.98	0.94	1.01	0.57	0.98
xalancbmk	0.90	0.98	1.00	1.01	1.16	0.66	0.83	0.93	1.03	0.93	0.53	1.03
povray	0.94	0.97	1.00	1.02	1.13	0.69	0.86	0.50	1.38	1.40	0.56	1.13
tonto	0.94	1.00	1.00	1.02	0.99	0.31	0.62	1.00	1.33	1.17	0.58	0.98

Table 5.10: Overhead of the *traces* optimisation (relative to baseline), on ODROID-XU3.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	0.68	0.90	1.01	0.95	N/A	0.20	N/A	N/A	0.92	0.78	0.49	0.27
bzip2	0.80	1.01	1.00	0.95	N/A	0.60	N/A	N/A	0.97	1.09	0.48	0.73
gcc	0.85	0.96	1.00	1.01	N/A	0.24	N/A	N/A	0.96	1.04	0.67	0.37
gobmk	0.62	0.93	1.03	0.95	N/A	0.33	N/A	N/A	0.94	0.90	0.57	0.50
sjeng	0.69	0.98	1.03	0.92	N/A	0.27	N/A	N/A	0.86	0.55	0.60	0.60
h264ref	0.91	1.00	1.00	0.93	N/A	0.15	N/A	N/A	0.94	0.59	0.76	0.66
omnetpp	0.69	0.84	1.02	0.97	N/A	0.08	N/A	N/A	1.03	0.55	0.57	0.30
astar	0.96	0.99	1.01	0.99	N/A	0.51	N/A	N/A	1.04	1.13	0.63	0.83
xalancbmk	0.78	0.91	1.01	0.97	N/A	0.41	N/A	N/A	1.03	0.72	0.53	0.45
dealII	0.89	0.96	0.99	1.00	N/A	0.12	N/A	N/A	0.86	0.47	0.56	0.28
povray	0.83	0.93	1.00	1.02	N/A	0.25	N/A	N/A	1.01	0.90	0.56	0.44
tonto	0.86	0.95	1.00	1.02	N/A	0.19	N/A	N/A	0.84	0.64	0.58	0.27
wrf	0.96	1.01	1.00	0.97	N/A	0.28	N/A	N/A	0.79	0.26	0.69	0.37

Table 5.11: Overhead of the *traces* optimisation (relative to baseline), on ODROID-X2.

Tables 5.10 to 5.14 show the relative change in the number of performance events when running SPEC CPU2006 under the *+traces* configuration, compared to the *baseline* configuration of MAMBO, for benchmarks with a significant change in execution time. With traces, many benchmarks gain a significant speed up: up to 18% on ODROID-XU3 (perlbench), up to 38% on ODROID-X2 (gobmk), up to 41% on Tronsmart R28 (gobmk), up to 31% on Jetson TK1 and up to 36% on APM X-C1 (gobmk). Across the benchmarks getting a speed-up, the number of architecturally executed branches is substantially reduced (by 11%

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	0.72	0.99	1.00	0.96	0.80	0.20	0.32	0.58	0.98	0.51	0.49	0.51
gcc	0.80	1.03	0.99	1.00	0.79	0.22	0.44	0.90	1.01	0.60	0.66	0.46
mcf	1.06	1.00	1.02	1.00	0.82	0.44	1.00	1.00	1.00	0.94	0.47	0.82
gobmk	0.59	0.99	1.02	1.01	0.89	0.34	0.39	0.50	0.98	0.56	0.57	0.56
sjeng	0.62	1.00	1.01	0.99	0.83	0.25	0.31	0.79	1.03	0.37	0.60	0.52
h264ref	0.90	1.01	0.99	0.95	0.98	0.16	0.55	0.84	0.95	0.42	0.76	0.72
omnetpp	0.74	0.97	0.99	0.96	0.80	0.08	0.30	0.93	0.98	0.49	0.57	0.38
xalancbmk	0.77	0.98	0.98	1.00	0.84	0.40	0.58	0.88	0.96	0.69	0.53	0.47
gamess	0.91	0.99	1.00	0.98	1.01	0.24	0.56	1.12	1.00	0.55	0.55	0.60
leslie3d	1.04	1.05	1.01	1.03	1.07	1.35	1.01	1.00	1.21	0.98	0.89	1.09
dealII	0.88	1.00	0.99	0.99	0.81	0.12	0.83	0.99	0.95	0.33	0.56	0.46
povray	0.79	0.97	0.97	0.93	0.78	0.29	0.53	0.39	1.00	0.70	0.56	0.46
GemsFDTD	1.06	1.01	1.01	1.01	1.04	0.75	0.99	0.99	1.07	0.24	0.57	0.88
tonto	0.87	1.01	0.99	0.99	0.86	0.17	0.56	0.97	0.98	0.47	0.57	0.36

Table 5.12: Overhead of the *traces* optimisation (relative to baseline), on Tronsmart R28.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	0.71	0.99	0.98	0.94	0.80	0.33	0.51	0.83	0.98	1.16	0.90	0.73
gcc	0.83	1.03	1.00	1.05	0.89	0.39	0.66	1.00	1.04	1.50	1.12	0.93
gobmk	0.70	0.99	1.02	1.00	0.90	0.57	0.65	0.64	0.98	1.38	0.92	1.18
sjeng	0.69	1.00	1.02	1.07	0.89	0.48	0.58	0.86	1.06	0.72	0.93	0.89
h264ref	0.90	1.01	1.00	0.90	0.95	0.35	0.66	0.69	0.94	0.92	1.11	0.89
omnetpp	0.76	0.97	0.99	0.97	0.79	0.30	0.52	0.83	1.03	0.88	0.81	0.83
xalancbmk	0.81	0.98	0.99	0.99	0.72	0.60	0.78	1.00	0.99	0.89	0.87	0.86
gamess	0.92	0.99	1.00	0.93	0.95	0.43	0.67	0.86	1.01	1.36	0.85	0.90
soplex	0.80	1.03	0.99	1.00	0.96	0.22	1.03	1.02	0.99	0.64	1.14	0.86
povray	0.82	0.97	0.98	0.98	0.80	0.53	0.75	0.21	1.03	1.73	0.77	0.75
calculix	1.03	1.00	1.01	1.00	1.07	0.41	0.98	0.97	1.00	1.11	1.02	1.95
tonto	0.85	1.00	1.01	1.19	0.90	0.20	0.56	1.01	1.01	1.29	1.02	0.86
sphinx3	0.94	1.03	1.00	1.00	0.85	0.62	1.00	1.04	0.89	1.03	1.35	0.42

Table 5.13: Overhead of the *traces* optimisation (relative to baseline), on Jetson TK1.

up to 53%), which is an intended consequence of traces merging code fragments commonly executed one after each other. The improved layout of traces and reduced number of executed branches also reduces the number of branch mispredictions, however this appears to be heavily dependent on microarchitecture: on ODROID-XU3 the average is a slight increase in the number of mispredictions, on ODROID-X2 all benchmarks trigger fewer mispredictions (by 17% up to 73%), on Tronsmart R28 all benchmarks apart from *leslie3d* trigger fewer mispredictions (from 12% up to 62%) and on Jetson TK1 *gobmk* and *calculix* have 18% and 95% more mispredictions respectively, while most benchmarks have fewer (by 7% up to 58%).

The number of L1 instruction loads is decreased with traces (due to increased locality and higher utilisation of cache lines) for some combinations of benchmarks

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	0.77	0.99	1.03	1.00	0.94	0.34	0.86	0.43	0.95	0.92	N/A	N/A
gcc	0.80	1.03	1.01	1.00	1.09	0.33	0.88	0.71	1.01	1.34	N/A	N/A
gobmk	0.64	0.99	1.02	0.88	1.15	0.57	0.79	0.34	0.99	0.95	N/A	N/A
sjeng	0.73	1.00	1.04	0.91	1.15	0.45	0.79	0.34	1.00	0.95	N/A	N/A
h264ref	0.94	1.01	1.00	0.98	1.19	0.25	0.96	0.87	0.97	0.88	N/A	N/A
omnetpp	0.80	0.97	1.01	1.00	1.04	0.13	0.67	0.94	1.00	1.17	N/A	N/A
xalancbmk	0.76	0.98	1.01	0.99	1.12	0.54	0.83	0.69	0.93	0.70	N/A	N/A
dealII	0.92	1.00	1.00	1.00	1.17	0.18	0.95	0.98	1.00	1.23	N/A	N/A
povray	0.91	0.97	1.01	0.98	1.22	0.53	0.90	0.22	2.30	1.46	N/A	N/A
tonto	0.94	1.00	1.00	1.00	1.17	0.22	0.92	0.92	1.03	1.02	N/A	N/A

Table 5.14: Overhead of the *traces* optimisation (relative to baseline), on APM X-C1.

and systems and increased (due to the code duplication being introduced by traces) for others. On Tronsmart R28 and Jetson TK1 most benchmarks have fewer L1 instruction loads, on ODROID-XU3 the number of benchmarks with fewer loads is equal to the number of benchmarks with more loads, while on APM X-C1 most benchmarks have a slight increase in loads. The difference between the different systems is mainly caused by the cache line length (ODROID-XU3 has 32 byte lines, the other three have 64 byte lines), speculative execution capabilities and loop buffer size (if available). The *L1i-l* event is not available on ODROID-X2. The number of L1 instruction load misses is decreased for most benchmarks on all systems, by as much as 69% on ODROID-XU3 (for *omnetpp* and *tonto*), 92% on ODROID-X2 and Tronsmart R28 (for *omnetpp*), 80% on Jetson TK1 (for *tonto*) and 87% on APM X-C1 (for *omnetpp*). The reduced number of L1 instruction misses is likely to be one of the main contributors to the improved performance with traces for benchmarks such as *perlbench*, *gcc*, *gobmk*, *sjeng*, *omnetpp*, *xalancbmk*, *povray* and *tonto*, which have high L1 instruction cache miss rates (from 3% up to 27%, depending on benchmark and system) with the *baseline* MAMBO configuration.

Interestingly, the number of speculatively executed branches, measured by the Jetson TK1 system, is higher compared to the *baseline* configuration for some benchmarks.

However, there are also several minor slow downs: *bzip2* on ODROID-XU3 (6% slowdown), *mcf*, *leslie3d* and *GemsFDTD* on Tronsmart R28 (slowdowns of 6%, 4% and 6% respectively) and *calculix* on Jetson TK1 (3% slowdown). Of these benchmarks, *calculix* appears to be affected by an increased number of L1 instruction loads, L1 instruction TLB misses and branch misses, which seem to

indicate excessive code duplication in traces; similarly *bzip2* seems to be affected by increased L1 instruction cache and L2 cache misses.

Hardware return address prediction

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
bzip2	0.93	1.00	1.00	0.98	0.98	0.81	0.98	0.90	1.00	0.77	1.05	0.93
gobmk	0.92	1.02	1.03	0.99	1.07	0.77	0.83	0.84	0.73	0.65	1.10	1.00
sjeng	0.94	1.01	1.01	1.01	1.02	0.75	0.83	0.95	0.71	0.50	1.08	1.00
h264ref	0.96	1.00	0.98	0.90	1.03	0.57	0.80	0.96	0.92	0.73	1.14	1.08

Table 5.15: Overhead of the *hw_ras* optimisation (relative to *+traces*), on ODROID-XU3.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
gcc	0.96	1.03	1.07	0.98	N/A	0.82	N/A	N/A	0.94	0.73	1.07	1.07
h264ref	0.96	0.97	0.99	1.03	N/A	0.76	N/A	N/A	0.92	0.42	1.14	0.65
xalancbmk	1.03	1.03	1.04	1.02	N/A	0.81	N/A	N/A	1.00	0.86	1.18	1.26
dealII	1.03	1.00	1.06	1.00	N/A	0.80	N/A	N/A	0.96	0.66	1.24	0.88
povray	1.05	1.05	1.06	1.03	N/A	0.77	N/A	N/A	0.96	0.81	1.29	1.35

Table 5.16: Overhead of the *hw_ras* optimisation (relative to *+traces*), on ODROID-X2.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
gobmk	0.94	1.02	1.02	0.94	1.11	0.74	0.78	0.86	0.97	0.54	1.10	0.91
h264ref	0.94	1.00	0.99	0.93	0.91	0.80	0.89	0.91	0.90	0.47	1.14	0.50
gromacs	0.96	0.99	0.99	0.95	0.99	0.24	0.97	1.00	0.95	0.68	0.96	0.86
cactusADM	0.96	0.99	0.99	0.97	0.99	0.23	0.97	0.99	0.99	1.04	0.53	0.31
leslie3d	0.96	0.99	0.99	1.00	1.01	0.67	1.01	1.00	0.83	0.98	0.98	0.90
dealII	1.12	1.01	1.07	1.00	1.06	2.14	1.03	0.99	0.99	0.71	1.24	0.99
GemsFDTD	0.95	0.99	0.99	0.98	0.95	0.32	1.01	1.00	0.94	0.67	0.96	0.64

Table 5.17: Overhead of the *hw_ras* optimisation (relative to *+traces*), on Tronsmart R28.

Tables 5.15 to 5.19 show the relative change in the number of performance events when running SPEC CPU2006 under the *+hw_ras +traces* configuration compared to the *+traces* configuration of MAMBO, for benchmarks with a change in execution time greater than 3%. With hardware return address prediction, many benchmarks gain a speed up (up to 8% on ODROID-XU3, up to 4% on ODROID-X2, up to 6% on Tronsmart R28, up to 15% on Jetson TK1 and up to 21% on APM X-C1).

The main purpose of this optimisation is to improve branch prediction for translated function returns. All of the cores used on the test systems use a hardware return address stack for return address prediction, however, without the

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
gobmk	0.85	1.02	1.03	0.91	0.97	0.75	0.76	0.81	0.95	0.46	1.05	0.79
sjeng	0.95	1.01	1.03	0.93	1.05	0.78	0.83	0.98	0.98	0.61	1.04	0.97
h264ref	0.92	1.00	0.99	1.19	0.90	0.69	1.10	0.94	0.90	0.41	1.09	0.55
omnetpp	0.95	1.04	1.05	1.02	0.92	0.54	0.79	0.97	0.98	0.54	1.12	0.57
xalancbmk	0.89	1.03	1.00	0.98	0.91	0.76	0.84	0.97	0.99	0.59	1.07	0.52
sphinx3	1.04	1.00	1.00	1.00	0.99	0.86	0.96	0.98	0.92	0.79	1.01	0.92

Table 5.18: Overhead of the *hw_ras* optimisation (relative to *+traces*), on Jetson TK1.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	0.85	1.03	0.99	0.98	0.85	0.85	0.97	0.86	0.91	0.73	N/A	N/A
gcc	0.91	1.03	1.02	0.99	0.96	0.68	0.93	0.94	0.98	0.74	N/A	N/A
gobmk	0.83	1.02	1.03	0.92	1.00	0.72	0.89	0.75	0.91	0.74	N/A	N/A
sjeng	0.91	1.01	0.99	0.94	0.97	0.67	0.86	0.85	1.00	0.91	N/A	N/A
omnetpp	0.88	1.04	1.02	0.99	0.98	0.42	0.80	0.93	0.99	0.88	N/A	N/A
astar	0.93	1.03	1.02	1.00	1.13	0.95	0.89	1.00	1.00	0.53	N/A	N/A
xalancbmk	0.85	1.03	1.00	0.99	0.94	0.78	0.86	0.95	0.99	0.85	N/A	N/A
povray	0.79	1.05	1.03	0.99	0.87	0.62	0.85	0.68	0.12	0.25	N/A	N/A
tonto	0.95	1.01	1.01	1.00	0.94	0.68	0.92	0.96	0.93	0.73	N/A	N/A

Table 5.19: Overhead of the *hw_ras* optimisation (relative to *+traces*), on APM X-C1.

hw_ras optimisation, all return instructions in the application are translated to indirect branches, which are handled by the generic indirect branch predictor instead. Whether a benchmark is affected by this optimisation depends both on its execution profile (e.g. how many return instructions it executes and how often their target changes) and on the microarchitecture (e.g. the penalty for branch mispredictions, the size and sophistication of the indirect predictor). The more complex microarchitectures, with longer pipelines (Cortex-A15 and X-Gene), appear to benefit more from this optimisation compared to the microarchitectures with shorter pipelines (Cortex-A9 and Cortex-A17). This is expected behaviour, since implementing this optimisation adds some additional instructions and (easily predictable) unconditional direct branches, whose overhead has to be amortised by the improved return address prediction. In addition, the benchmarks running on Cortex-A7 (on the ODROID-XU3) benefit from this optimisation, despite the simple architecture with a short pipeline, because the generic indirect branch translations generated by MAMBO are not predicted at all.

The performance of several benchmarks is negatively affected: *xalancbmk*, *dealII* and *povray* on ODROID-X2 (3%, 3% and 5% respectively), *dealII* on Tronsmart R28 (12%) and *sphinx3* on Jetson TK1 (4%). None of the monitored events on Jetson TK1 show a significant overhead, therefore it is not clear what is

causing the slowdown. The three benchmarks suffering a slowdown on ODROID-X2 appear to be simply failing to amortise the cost of additional branches and instructions. *dealII* on Tronsmart R28 appears to be affected by an increase in L1 instruction cache loads and branches caused by the changes to the code layout introduced by this optimisation. While the number of L1 instruction load misses is increased by a factor of 2, the overall miss rate is still relatively low, around 0.3%, therefore it is unlikely to be a major contributor to the higher overhead.

Some benchmarks in the result tables show improved performance with an equal or higher number of mispredicted branches. This appears to be caused by the *br-m* event counting misses for all speculatively executed branches and not only for architecturally executed branches. The reported number of L1 instruction load misses and L1 instruction TLB misses is improved for most benchmarks on all of the five test machines, as expected and consistent with improved branch prediction and hardware prefetching. The number of L1 instruction cache misses is reduced by up to 63% on ODROID-XU3, 24% on ODROID-X2, 77% on Tronsmart R28, 46% on Jetson TK1 and 58% on APM X-C1, while the number of L1 instruction TLB misses is reduced by up to 50% on ODROID-XU3, 58% on ODROID-X2, 53% on Tronsmart R28, 59% on Jetson TK1 and 75% on APM X-C1. The improved hit rate in the first level instruction cache also reduces the number of loads and misses for the higher level caches.

Huge pages

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	0.97	1.00	1.00	0.96	1.01	1.05	0.83	0.99	1.02	1.01	1.00	1.06
xalancbmk	0.97	1.00	1.00	0.97	0.99	1.01	0.90	1.00	1.00	1.01	1.00	1.01
leslie3d	1.35	1.00	1.03	1.85	1.14	1.17	1.42	1.00	1.02	1.20	1.01	1.02

Table 5.20: Overhead of the *hugetlb* optimisation (relative to *+hw_ras +traces*), on Jetson TK1.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	0.96	1.00	1.00	1.00	0.99	1.08	1.00	1.00	0.46	0.03	N/A	N/A
gobmk	0.97	1.00	1.00	1.01	1.04	1.07	1.02	1.02	0.52	0.08	N/A	N/A
xalancbmk	0.96	1.00	1.00	1.01	1.03	1.05	1.00	1.00	0.90	0.05	N/A	N/A

Table 5.21: Overhead of the *hugetlb* optimisation (relative to *+hw_ras +traces*), on APM X-C1.

Support for huge pages is not implemented in the version of the Linux kernel used on ODROID-X2, therefore this optimisation has not been evaluated on that

machine. Additionally, this optimisation fails to produce a speed up of 3% or more on any SPEC CPU2006 benchmark on the ODROID-XU3 and Tronsmart R28 systems, despite reducing the number of L1 instruction TLB misses to a negligible value on both systems and also measurably reducing the number of L1 data TLB misses on Tronsmart R28. Tables 5.20 and 5.21 show the relative change in the number of performance events between running SPEC CPU2006 under the *+hugetlb +hw_ras +traces* configuration compared to the *+hw_ras +traces* configuration of MAMBO, for benchmarks with a change in execution time greater than or equal to 3%.

Jetson TK1, whose Cortex-A15 core only supports 4KiB granularity in the L1 TLB entries, gets a 3% speedup on the *perlbench* and *xalancbmk* benchmarks. On the same machine, *leslie3d*, a benchmark which runs 21% - 25% faster than native under most MAMBO configurations, has a significant slowdown due to the *hugetlb* optimisation and ends up running 3% slower than native. This appears to be caused by the data cache hit rate reverting back to the native value[1] rather than by additional overhead being introduced by this optimisation.

On APM X-C1, the benchmarks *perlbench*, *gobmk* and *xalancbmk* gain 4%, 3% and 4% speed-ups, respectively.

Low footprint inline hash table lookup dispatch

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	1.03	1.04	1.06	0.97	1.04	1.07	1.01	1.03	0.92	0.92	1.00	1.00
gobmk	1.04	1.03	1.06	1.04	1.02	1.06	1.05	1.04	0.89	0.98	1.00	1.00
omnetpp	1.03	1.06	1.08	1.01	1.02	1.15	1.05	1.01	1.14	1.03	1.00	1.00
xalancbmk	1.03	1.04	1.07	0.99	1.00	1.09	1.03	1.02	0.95	0.87	1.00	1.00
povray	1.06	1.08	1.09	1.03	1.05	1.02	1.03	1.36	1.03	1.21	1.00	1.00

Table 5.22: Overhead of the *ldm_pc_sr* optimisation (relative to *+traces*), on ODROID-XU3.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	1.03	1.04	1.05	1.02	N/A	1.09	N/A	N/A	1.01	1.02	1.00	1.04
xalancbmk	1.03	1.04	1.04	1.01	N/A	1.04	N/A	N/A	0.99	1.00	1.00	1.02
povray	1.04	1.07	1.06	1.01	N/A	1.07	N/A	N/A	1.02	1.04	1.00	0.97

Table 5.23: Overhead of the *ldm_pc_sr* optimisation (relative to *+traces*), on ODROID-X2.

[1] See the discussion about *leslie3d* on Jetson TK1 on page 111

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
gromacs	0.96	0.99	0.99	0.95	0.99	0.24	0.97	1.01	0.96	0.74	0.93	0.93
GemsFDTD	0.96	1.00	1.00	0.99	0.96	0.60	1.00	1.00	0.97	1.12	0.96	0.81

Table 5.24: Overhead of the *ldm_pc_sr* optimisation (relative to *+traces*), on Tronsmart R28.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
calculix	0.96	1.00	1.00	1.00	0.94	0.94	0.99	1.02	1.02	0.86	1.00	0.62
sphinx3	1.03	1.01	1.00	0.99	1.00	1.17	0.96	0.99	0.97	0.90	1.00	0.93

Table 5.25: Overhead of the *ldm_pc_sr* optimisation (relative to *+traces*), on Jetson TK1.

Tables 5.22 to 5.26 show the relative change in the number of performance events when running SPEC CPU2006 under the *+ldm_pc_sr +traces* configuration compared to the *+traces* configuration of MAMBO, for benchmarks with a change in execution time greater than or equal to 3%. The effect of this linking scheme varies depending on the microarchitecture: on ODROID-XU3 and ODROID-X2 all benchmarks runs with increased or equal overhead (by up to 6% on ODROID-XU3 and 4% on ODROID-X2), on Jetson TK1 one benchmark gains a speed-up (of 4%) while a second benchmarks suffers a slow-down (3%), while on Tronsmart R28 and APM X-C1 all benchmarks run with the same or lower overhead (up to 4% lower on Tronsmart R28 and up to 14% lower on APM X-C1). This linking scheme is intended to reduce the pressure on the data TLB and on the data cache, while also avoiding an expensive PC-relative load in the PC, at the cost of additional instructions being added to the inline hash table lookups. To get a speed-up, the improved performance due to higher TLB and cache hit rates has to overcome the penalty of executing these additional instructions, therefore it is expected to get the best performance on wider OOO microarchitectures, which is indeed the pattern observed here.

The main penalties associated with using this linking scheme as observed using performance counters are: an increased number of architecturally executed instructions for some benchmarks (up to 8% for *povray*); and an increased number of L1 instruction cache loads, L1 instruction cache load misses and L1 instruction TLB misses as a result of the increased code size, mostly on the ODROID-X2 and ODROID-XU3 systems.

On Tronsmart R28, the main benefits appear to be the slight reduction in L1 data cache misses and branch misses. Despite the large relative improvement, the reduced number of L1 instruction cache misses and L1 instruction TLB misses

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	0.89	1.04	0.99	1.00	0.84	1.00	1.08	1.00	1.00	1.03	N/A	N/A
gcc	0.94	1.03	1.02	1.00	0.91	1.03	1.06	1.01	1.00	1.03	N/A	N/A
omnetpp	0.92	1.06	1.05	0.99	0.99	0.65	1.02	0.98	1.00	1.03	N/A	N/A
astar	0.92	1.04	1.03	1.00	1.09	1.19	1.02	1.00	1.00	1.14	N/A	N/A
xalancbmk	0.95	1.04	1.04	1.01	0.92	1.07	1.08	1.02	1.00	1.01	N/A	N/A
povray	0.86	1.08	1.05	1.02	0.91	1.02	1.00	1.29	1.09	1.00	N/A	N/A

Table 5.26: Overhead of the *ldm_pc_sr* optimisation (relative to *+traces*), on APM X-C1.

are unlikely to have a significant contribution to the improved performance of *gromacs* and *GemsFDTD*: both benchmarks in the *+traces* configuration have an L1 instruction cache miss rate lower than 0.06% and an L1 instruction TLB miss rate under 0.002% of L1 instruction cache loads.

On APM X-C1, it is not clear why performance is improved. Overall, this optimisation is effective in some cases, however the expected improvements in data cache and data TLB hit rates were not observed.

Adaptive indirect branch inlining

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
h264ref	0.94	0.93	0.96	0.99	0.89	0.78	0.92	0.97	0.80	0.72	0.97	1.18
omnetpp	0.97	0.90	0.91	1.05	0.84	0.89	1.00	0.99	0.88	0.82	0.93	1.04
astar	0.94	0.93	0.94	0.99	0.90	0.93	0.99	0.99	1.01	0.81	0.96	1.02
xalancbmk	0.94	0.91	0.90	1.05	0.85	0.90	0.99	0.98	0.88	0.93	0.93	1.15

Table 5.27: Overhead of the *aibi* optimisation (relative to *+traces*), on ODROID-XU3.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
h264ref	0.96	0.94	0.96	1.03	N/A	1.03	N/A	N/A	1.13	1.42	0.97	0.97
omnetpp	0.97	0.90	0.94	1.04	N/A	0.86	N/A	N/A	1.01	1.15	0.93	0.87
xalancbmk	0.97	0.90	0.93	1.09	N/A	1.01	N/A	N/A	1.07	1.14	0.93	0.92
dealII	0.92	0.89	0.90	1.01	N/A	0.98	N/A	N/A	1.03	1.23	0.90	0.82

Table 5.28: Overhead of the *aibi* optimisation (relative to *+traces*), on ODROID-X2.

Tables 5.27 to 5.31 show the relative change in the number of performance events when running SPEC CPU2006 under the *+aibi +traces* configuration compared to the *+traces* configuration of MAMBO, for benchmarks with a change in execution time greater than or equal to 3%. This optimisation relies on the previous target of an indirect branch being a good prediction for the following

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
h264ref	0.96	0.93	0.98	1.01	0.90	0.90	0.97	0.99	1.22	1.05	0.97	1.04
omnetpp	0.96	0.90	0.98	1.04	0.87	0.74	0.98	0.99	1.09	0.89	0.93	0.87
xalancbmk	0.94	0.91	0.98	1.10	0.82	0.87	0.98	0.99	1.12	1.14	0.93	0.87
milc	0.97	0.94	0.96	1.00	0.94	0.94	0.99	1.00	0.86	0.24	0.93	1.08
gromacs	0.96	0.99	0.98	0.95	0.98	0.33	0.98	1.00	0.97	0.83	0.93	0.99
dealII	0.96	0.89	0.97	1.01	0.85	0.88	1.00	0.99	1.09	1.26	0.90	1.08
lbm	0.97	1.00	0.98	0.97	1.00	1.02	0.99	1.00	1.00	1.00	1.00	1.04

Table 5.29: Overhead of the *aibi* optimisation (relative to *+traces*), on Tronsmart R28.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	1.06	1.01	1.01	1.09	1.12	0.98	1.04	1.01	1.07	0.91	1.02	1.34
mcf	0.94	0.99	1.00	1.00	1.12	1.01	1.05	1.01	1.00	0.99	1.03	1.27
hmmer	0.95	1.00	1.00	0.64	1.06	0.95	0.77	0.87	0.99	0.96	1.00	1.01
h264ref	0.94	0.93	0.99	1.22	0.92	0.88	1.09	0.94	1.12	1.07	1.06	0.97
xalancbmk	0.96	0.91	0.98	1.11	0.87	0.94	1.03	1.05	1.20	1.12	1.01	0.91
dealII	0.91	0.89	0.96	1.02	0.85	0.85	1.00	1.00	1.06	0.93	1.02	0.87
calculix	0.96	1.00	1.00	1.00	0.94	0.97	0.98	1.00	1.03	0.92	1.00	0.64

Table 5.30: Overhead of the *aibi* optimisation (relative to *+traces*), on Jetson TK1.

execution of a branch. The performance benefit gained by executing fewer instructions and fewer branches in the case of a correctly predicted branch must amortise the additional cost of updating the predicted address in the case of a miss.

The performance improvement of this linking scheme can be attributed to multiple factors: when predictions tend to be correct, the number of architecturally executed instructions can be reduced by as much as 11% (dealII); additionally, by reducing the size of the code typically executed for inline hash table lookups, the number of instruction cache loads and instruction load misses is generally reduced; and by avoiding the hash table lookup in case of a correct prediction, the number of architecturally executed branches is reduced. Since the number of branch misses is reported as either increased and decreased, depending on benchmark and microarchitecture, and since the *br-m* event applies both to architecturally and to speculatively executed branches, it is not clear what is the effect of this linking scheme on branch prediction.

perlbench running on Jetson TK1 suffers a slowdown. This appears to be related to the poor predictability of the indirect branches executed by this benchmark, as shown in Section 5.3.3. These cause AIBI to mispredict relatively often and fall back to the inline hash table lookup routine, which leads to additional

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
sjeng	1.03	0.98	0.99	1.19	1.01	1.08	0.99	1.07	1.00	1.36	N/A	N/A
h264ref	0.96	0.93	0.97	1.13	1.01	0.97	0.95	1.01	1.03	1.16	N/A	N/A
dealII	0.92	0.89	0.91	1.01	0.92	1.23	0.71	1.00	1.00	1.26	N/A	N/A

Table 5.31: Overhead of the *aibi* optimisation (relative to *+traces*), on APM X-C1.

hardware branch mispredictions (which are particularly expensive on this system) and increases the number of L1 instruction loads.

This optimisation improves the average performance on all test systems, however on some systems it is not as effective as the *+hw_ras* linking scheme, as shown in Table 5.4.

On ODROID-XU3, ODROID-X2 and Tronsmart R28 all benchmarks are either not affected or achieve lower overhead. The highest speed-ups on each system are: 6% on ODROID-XU3 and Tronsmart R28, 8% on ODROID-X2 and APM X-C1 and 9% on Jetson TK1.

5.5 Summary

This chapter presented a number of optimisations which address the overhead introduced by the interaction between the hardware microarchitectures and the code generated by DBM systems: *traces* reduce the code cache fragmentation by grouping together basic blocks which are likely to execute sequentially; *hardware-assisted return address prediction* is a technique which enables use of the hardware return address prediction without maintaining a software return address stack; *low footprint inline hash table lookup dispatch* is intended to reduce the impact of inline hash table lookups on the data caches and TLBs; *adaptive indirect branch inlining* is a software indirect branch prediction scheme which allows quick and frequent updates of the predicted address; and use of *huge pages* for the code cache and metadata minimises the number of TLB misses caused by the larger working sets of data and instructions in translated applications. By using the optimal combination of these optimisations on each system, the geometric mean overhead of MAMBO was reduced by at least 29% (on ODROID-XU3) and by as much as 58% (on APM X-C1) compared to the *baseline* configuration.

The *low footprint inline hash table lookup dispatch* has improved the performance on two of the benchmarks running on Tronsmart R28 and six of the benchmarks running on APM X-C1. However, the expected improvement in data

cache and data TLB hit rates has been minimal on the former system and absent on the latter. It is not clear how the significant performance improvement on APM X-C1 was achieved.

The use of *huge pages* has had a low effect on performance, only on Jetson TK1 and on APM X-C1, despite significantly reducing the number of TLB misses. It appears that TLB misses have a minimal impact on the overhead of MAMBO.

Chapter 6

Conclusions and future work

6.1 Summary and conclusions

Dynamic Binary Modification (DBM) is a technique for modifying applications transparently while they are executed, working at the level of native machine code. DBM has numerous applications, including instrumentation and program analysis, virtualisation and dynamic translation. However, many uses are hindered by the introduced overheads, particularly in terms of execution speed. As a result, the topic of DBM performance has been an active area of research. Nonetheless, this thesis identified a number of limitations of the existing literature and research platforms: firstly, DBM performance research has mostly focused on the x86 architecture, while the ARM architecture, which is dominating the mobile computing market and is transitioning to general use, has generally been overlooked. This introduced the second limitation: existing DBM systems are large, complex systems, and therefore are difficult to understand and modify by researchers. In particular for the ARM architecture, there were no performance-oriented DBM systems at the time this investigation was done. This was addressed by creating MAMBO, a low overhead DBM system for ARM, which is capable or running industry standard benchmarks and many other applications. With MAMBO as a base experimental platform, DBM optimisations were then quickly prototyped and developed, particularly in the area of branch handling, hot trace generation and microarchitectural optimisations. However, a new concern then emerged: publications in the area of DBM performance tend to use one or two systems for evaluation. Is this an appropriate methodology and can it provide relevant results? To find out, a number of performance optimisations were evaluated on

five different ARM systems, each using a different microarchitecture. The results show that 1) the performance of a DBM system can change significantly between microarchitectures (the overhead varied between 11% and 21% on the five systems) and that 2) some optimisations which generally improve performance on one system can actually harm performance on another system (e.g. the *low footprint inline hash table lookup dispatch* improves performance on *APM X-C1*, while reducing it on *ODROID-XU3* and *ODROID-X2*).

An important contribution is the development of MAMBO, a DBM system for the ARM architecture, which achieves similar overhead to state-of-the-art DBM tools for the x86 architecture by using the optimisations put forward in this thesis. MAMBO has an overhead between 11% and 21% on the five ARM systems on which it was evaluated, while other DBM systems with support for ARM have much higher overheads, between 187% for the now discontinued Pin for ARM [HK06] and 1,907% for QEMU [Bel05]. Furthermore, the codebase of MAMBO is very small (fewer than 10,000 lines of code) compared to other DBM systems, which facilitates further research. MAMBO was publicly released as Free software under the Apache 2.0 license [Gor16]. The design of the base system is presented in Chapter 3.

MAMBO implements a number of novel optimisations presented in this thesis. These are: the generation of hot code *traces* using a new algorithm (Section 5.2), *low overhead return address prediction* (Section 4.2.1), *hardware-assisted return address prediction* (Section 5.3.1), *space-efficient table branch linking* (Section 4.2.2), *low footprint inline hash table lookup dispatch* (Section 5.3.2) and *adaptive indirect branch inlining* (Section 5.3.3). These optimisations were evaluated using the SPEC CPU2006 benchmark suite. The first optimisation is an improvement over the NET [DB00] **trace** building algorithm, which was modified to avoid recording traces across difficult to predict indirect branches. This optimisation was shown to reduce the geometric mean overhead of MAMBO across all test systems, from 20% on ODROID-XU3 to 43% on ODROID-X2, by reducing the number of executed branches, mispredicted branches and instruction cache misses. The next two optimisations are both used for translating *returns* and can not be used at the same time. **Low overhead return address prediction** is a variation on the well known software return address prediction with a *fat entry* software return address stack. Because of the properties of the ABI and the implementation of exceptions on ARM, *low overhead return address prediction* can

trade off full transparency for lower overhead, with the ability to detect incorrect predictions and fall back to a fully transparent but less efficient translation. This technique reduces the overhead by 6% on the Jetson TK1 and by 9% on the ODROID-X2. **Hardware-assisted return address prediction** is a technique which enables hardware return address prediction in the code cache, while continuing to use the generic translation for indirect branches at the software level. As opposed to *low overhead return address prediction*, this technique is fully transparent and is also compatible with traces. The effect of this optimisation varies widely depending on the microarchitecture: on APM X-C1, ODROID-XU3 and Jetson TK1 it reduces the overhead by 30%, 12% and 10% respectively, while on ODROID-X2 and Tronsmart R28 it has no or minimal effect. **Space-efficient table branch linking** is an optimisation for translated table branches which reduces the size of the translation compared to the *shadow jump table* used by FastBT [PG10]. Furthermore, on ARM it can be implemented more efficiently. On a subset of benchmarks which use a significant number of table branches, this optimisation improved performance by 1.5% to 3.6% compared to the shadow jump table scheme when both were configured to use the same amount of code cache space. **Low footprint inline hash table lookup dispatch** is an optimisation specific to ARM and to the MAMBO implementation of inline hash table lookups, used to perform the SPC-to-TPC lookup. This optimisation was intended to reduce the pressure on the data cache and the data TLB. It reduces the overhead by 17% on APM X-C1, however it actively harms performance on ODROID-XU3 and ODROID-X2. Furthermore, the performance counter analysis has failed to explain how the performance was improved on APM X-C1. However, this optimisation does not appear to significantly reduce the number of data cache or data TLB misses as expected. **Adaptive indirect branch inlining** is a software indirect branch target prediction scheme designed to have a high hit rate to avoid the hardware branch misprediction penalties incurred by *indirect branch inlining*. This optimisation reduces the geometric mean overhead of MAMBO by 3%, 7%, 7%, 9% and 10% on APM X-C1, ODROID-X2, Jetson TK1, Tronsmart R28 and ODROID-XU3, respectively. The main cause of this improvement seems to be the lower dynamic instruction count, obtained by avoiding the inline hash table lookups.

Of these optimisations, *low overhead return address prediction* and *hardware-assisted return address prediction* apply to the same type of instructions and

therefore can not be used at the same time. Furthermore, there are various interactions between some of the other optimisations, meaning that if combined, the actual performance improvement might be lower than expected. The *low footprint inline hash table lookup dispatch* and *adaptive indirect branch inlining* apply to branches translated using the inline hash table lookup. When optimisations specialised for specific types of indirect branches are enabled, fewer branches are translated using the inline hash table lookup, and therefore fewer translated branches can benefit from these other optimisations. For example, in most cases there is little to no speed-up when enabling *adaptive indirect branch inlining* if *hardware-assisted return address prediction* is already used.

For the microarchitectural optimisations in Chapter 5, the performance was analysed on five different ARM platforms, which has shown that 1) whether an optimisation for a DBM system is effective or not depends on multiple factors, include the microarchitecture of the processor it is running on and the type of workload, and that 2) the optimal combination of linking schemes can be different between multiple systems. Considering that some optimisations increased performance on one system and decreased it on other systems, while running the same workload, it becomes clear that performance evaluations of DBM systems should aim to report results using a similar wide range of hardware platforms. While this is the case for some of the cited publications [SSNB06], many only use one or two machines.

6.2 Optimisation selection guidelines

Table 6.1 summarises the dependence of each optimisation on microarchitectural features and provides guidelines on when it is effective to enable each optimisation.

6.3 Portability to AArch64

While the techniques introduced in this thesis have only been implemented for 32-bit ARM, their portability to AArch64 has also been considered. This is summarised in Table 6.2. It must be noted that this analysis is strictly showing whether the instruction set allows the implementation of the optimisations and it does not make any claims on their performance portability, i.e. whether their AArch64 implementation would be effective in improving performance.

Optimisation	Relevant microarchitectural features	Guidelines
Low overhead return address prediction	return address prediction (optional)	trade-off between improved return address prediction and the cost of maintaining a shadow return address stack; limited transparency; should be disabled if mispredicting; hardware return address prediction is desirable
Space-efficient table branch linking	-	always use
Inline hash table lookup	-	always use
Fallthrough branch linking	-	always use
Direct branch linking	-	always use
Eliding unconditional direct branches	-	trade-off between code cache size and code cache fragmentation
Traces	instruction cache; direct branch prediction; instruction prefetching	should be used in most cases, however application startup performance could be affected on very large code bases by the additional profiling
Hardware-assisted return address prediction	return address prediction	return address prediction is required; trade-off between a slight increase in dynamic instruction count and more accurate target prediction for translated returns; most effective on systems with no generic indirect branch target prediction and / or higher branch misprediction penalty
Low footprint inline hash table lookup dispatch	data cache; data TLB	must be able to directly load a target address from memory in the PC, not available on AArch64
Adaptive indirect branch inlining	indirect branch prediction	use for translated generic indirect branches; also use for translated returns on systems with hardware indirect branch prediction and low branch misprediction penalty; prediction accuracy heavily dependent on the workload
Huge pages	support for huge pages in data and/or instruction TLBs	slight performance improvement, most noticeable on high performance cores with small TLBs

Table 6.1: Optimisation selection guidelines

Optimisation	AArch64 comp.	Notes
Low overhead return address prediction	Y	-
Space-efficient table branch linking	Y	the explicit table branch instructions have been removed in AArch64; would require more complex pattern matching in the code scanner; the *LDRB* and *ADD* instructions can be used instead of *TBB* to implement the offset table
Inline hash table lookup	Y	-
Fallthrough branch linking	Y	-
Direct branch linking	Y	additional challenge because of the limited range of conditional branches and removal of the IT instruction
Eliding unconditional direct branches	Y	-
Traces	Y	-
Hardware-assisted return address prediction	Y	-
Low footprint inline hash table lookup dispatch	N	requires loads into PC, which are no longer allowed
Adaptive indirect branch inlining	Y	-
Huge pages	Y	-

Table 6.2: Portability of the optimisations to AArch64

6.4 Future work

6.4.1 Asynchronous multithreaded trace generation

DBM systems interleave scanning the application code and executing the modified code. Thus, the performance of the code scanner must be carefully managed to avoid interrupting the execution of the application for significant lengths of time. This is the desired behaviour when cold code runs for the first time (i.e. when MAMBO creates basic blocks), to minimise the startup latency. However, when traces of the hot code are created, more processing time could be used to optimise their translation, with the expectation that improved performance will amortise the longer scanning time. Nevertheless, the creation of traces is also interleaved with executing the modified application and therefore similar latency restrictions must apply. This restriction on the latency of the traces code scanner limits the number and types of optimisations which can be applied.

It would be worthwhile to decouple the generation of traces from the thread running the modified code. Because a version of the modified code would already exist in basic blocks at the time traces are created, its execution could continue uninterrupted. This would be implemented by starting additional threads exclusively for asynchronous generation of optimised code traces. A similar approach is used by ArcSim [ABVK+11], an instruction set simulator which uses interpretation for cold code and DBT generated by a pool of *JIT compilation workers* for hot code. However, because MAMBO uses basic blocks instead of interpretation for the cold code, the trade-offs involved change significantly. For example, because the overhead of basic blocks (2-3x slowdown in the worst cases) is much lower than the overhead of interpretation (on the order of a 100x slowdown), a longer trace creation latency can be tolerated. In this case, the factors limiting the trace creation time are expected to become the energy overhead and for some workloads, the duration of execution of the workload. On same-ISA heterogeneous systems, the selection of core on which to create traces presents an opportunity for adjusting the delay - energy overhead trade-off. Multithreaded workloads scalable to the number of cores available in the system pose a challenge to this approach because the workload and the trace creation threads would compete directly for processing time. This case is likely to require special handling to prevent performance degradation.

6.4.2 Automatic optimisation of DBM plugins

The optimisations presented in this thesis are aimed at improving the performance of the DBM systems themselves. The low overhead DBM frameworks which can be built using these techniques enable users to efficiently run their modification, instrumentation and optimisation DBM plugins. However, implementing the plugins efficiently requires experienced developers with a good understanding of the target architecture, microarchitecture and of the DBM system itself. Furthermore, implementing plugins at this low abstraction level is relatively verbose and slow compared to using a higher level programming language like *C* or *C++*. In practice, many DBM plugins are implemented using a low effort, inefficient approach. In particular, many DBM plugins implement most of their logic using functions in higher level programming languages (called *instrumentation functions* for the remaining of this section), which are then called by the modified application. However, this approach is inefficient because the application context has to be preserved before calling the instrumentation functions and then restored when returning to the application. This technique can be particularly inefficient when short instrumentation functions are called often, in which case the cost of these context switches can dominate the execution time.

Given this usage pattern, it becomes desirable to support automatic optimisation across the instrumentation functions implemented in high level languages and the code of the application. Furthermore, while the source code of the applications being modified is not available, the source code of plugins is available to their developers. The availability of a higher level representation for the instrumentation functions makes it practical to employ more advanced code transformations compared to modifying the code of the application. A number of such transformations are of potential interest. For example, automatic inlining of short instrumentation functions in the code cache. This would improve the code cache locality and also eliminate the function call and return. Inlining longer instrumentation functions, however, is likely to result in excessive code duplication. Furthermore, instruction scheduling has a significant effect on the performance of in-order microarchitectures and therefore re-scheduling code fragments which contain inlined instrumentation code could reduce overhead. Pin [LCM+05] performs some optimisations on the machine code of the instrumentation functions, however the opportunities brought by access to the source code or an IR representation of the instrumentation functions have not been explored.

6.4.3 Dynamic microarchitectural optimisations

The evaluation of the optimisations in Chapter 5 has shown that different microarchitectures can benefit from different optimisations. Furthermore, an experiment in optimising the *memcpy* function has indicated that in some cases it is possible to significantly exceed the native performance if generating an implementation tuned for a specific microarchitecture. However, in the context of same-ISA heterogeneous multicores which migrate threads between different types of cores, such as ARM big.LITTLE, it is not possible to optimise for a single microarchitecture. This can be addressed by dynamically detecting the type of core on which each thread is running. However, two challenges remain: how to efficiently detect the migration from one type of core to another? And once the thread has been migrated, how can this be handled efficiently? For example, it would be possible to maintain multiple separate software code caches, one for each microarchitecture. However, safely switching from one to another and reducing the memory overhead are likely to present significant difficulties.

6.4.4 Trace layout optimisations

The evaluation of performance for different trace creation thresholds in Section 5.4.2 has shown that this parameter can have a significant effect. However, no clear pattern has emerged and therefore further analysis of this behaviour is required. The aim is either to further refine the trace selection algorithm and minimise this variance or, alternatively, to develop heuristics which can be used at runtime to select *good* trace creation thresholds for each application. The first step in analysing this effect will be to use hardware performance counters to determine how performance is affected, similarly to the evaluation in Section 5.4.4. Then, the differences between the generated traces will be manually analysed, both in terms of selected paths and generated code.

Bibliography

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices*, 41(11):2–13, 2006.

[ABVK+11] Oscar Almer, Igor Böhm, Tobias Edler Von Koch, Björn Franke, Stephen Kyle, Volker Seeker, Christopher Thompson, and Nigel Topham. Scalable multi-core simulation using parallel dynamic binary translation. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 190–199. IEEE, 2011.

[ARM10] ARM. *Cortex-A8: Technical Reference Manual, Revision r3p2*, 2010.

[ARM13a] ARM. *ARM Cortex-A15 MPCore Processor Technical Reference Manual, Revision r4p0*, 2013.

[ARM13b] ARM. big.LITTLE technology: The future of mobile, 2013. (Visited on 13/07/2016).

[ARM13c] ARM. *Cortex-A7 MPCore Technical Reference Manual, Revision r0p5*, 2013.

[ARM14] ARM. *ARM Cortex-A17 MPCore Processor Technical Reference Manual, Revision r1p1*, 2014.

[ARM15] ARM. *ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, 2015.

[ARM16a] ARM. *ARM Cortex-A53 MPCore Processor Technical Reference Manual, Revision r0p4*, 2016.

[ARM16b] ARM. *ARM Cortex-A57 MPCore Processor Technical Reference Manual, Revision r1p3*, 2016.

[ARM16c] ARM. *ARM Cortex-A72 MPCore Processor Technical Reference Manual, Revision r0p3*, 2016.

[ARM16d] ARM. *ARM Cortex-A73 MPCore Processor Technical Reference Manual, Revision r0p2*, 2016.

[ARM16e] ARM. *ARM Cortex-A9 Technical Reference Manual, Revision r4p1*, 2016.

[ARM16f] ARM. *Cortex-A5: Technical Reference Manual, Revision r0p1*, 2016.

[BBTV15] Darrell Boggs, Gary Brown, Nathan Tuck, and K Venkatraman. Denver: NVIDIA's first 64-bit ARM processor. 2015.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, November 2000.

[BM11] Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM.

[Bru04] Derek Lane Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[BZ16] Derek Bruening and Qin Zhao. Building dynamic tools with DynamoRIO on x86 and ARM, 2016. `https://github.com/DynamoRIO/dynamorio/releases/download/release_6_1_0/DynamoRIO-tutorial-mar2016.pdf` (Visited on 17/08/2016).

[BZA12] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. *ACM SIGPLAN Notices*, 47(7):133–144, 2012.

[CHK93] Keith D Cooper, Mary W Hall, and Ken Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, 1993.

[CK94] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '94, pages 128–137, New York, NY, USA, 1994. ACM.

[DB00] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. *SIGPLAN Not.*, 35(11):202–211, November 2000.

[DGB+03] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 15–24. IEEE Computer Society, 2003.

[dGGL16] Amanieu d'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. Optimizing indirect branches in dynamic binary translators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):7, 2016.

[DH11a] Derek Davis and Kim Hazelwood. Improving region selection through loop completion. In *ASPLOS Workshop on Runtime Environments/Systems, Layering, and Virtualized Environments (RESoLVE)*, 2011.

[DH11b] Balaji Dhanasekaran and Kim Hazelwood. Improving indirect branch translation in dynamic binary translators. In *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering, and Virtualized Environments*, pages 11–18, 2011.

[DMW15] Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security, ASIACCS*, volume 15, 2015.

[Gor16] Cosmin Gorgovan. MAMBO: A low-overhead dynamic binary modification tool for ARM, 2016. `https://github.com/beehive-lab/mambo`.

[HH97] Raymond J Hookway and Mark A Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9:3–12, 1997.

[HHS05] David Hiniker, Kim Hazelwood, and Michael D Smith. Improving region selection in dynamic optimization systems. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, pages 11–pp. IEEE, 2005.

[HK06] Kim Hazelwood and Artur 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*, pages 261–270. ACM, 2006.

[HM80] R. Nigel Horspool and Nenad Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[HMC94] Jeffrey K Hollingsworth, Barton Paul Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 841–850. IEEE, 1994.

[HWH+07] Jason D Hiser, Daniel Williams, Wei Hu, Jack W Davidson, Jason Mars, and Bruce R Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 61–73. IEEE Computer Society, 2007.

[JYHC14a] Ning Jia, Chun Yang, Yu He, and Xu Cheng. Dtt: Program structure-aware indirect branch optimization via direct-tpc-table in dbt system. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 12:1–12:10, New York, NY, USA, 2014. ACM.

[JYHC14b] Ning Jia, Chun Yang, Yu He, and Xu Cheng. SPTU: Improving dynamic binary translation through software prediction with target updating. In *Proceedings of International Conference on Systems and Storage*, SYSTOR 2014, pages 2:1–2:12, New York, NY, USA, 2014. ACM.

[JYW+13] Ning Jia, Chun Yang, Jing Wang, Dong Tong, and Keyi Wang. SPIRE: improving dynamic binary translation through SPC-indexed indirect branch redirecting. In *ACM SIGPLAN Notices*, volume 48, pages 1–12. ACM, 2013.

[KS03a] Ho-Seop Kim and James E. Smith. Dynamic binary translation for accumulator-oriented architectures. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 25–35, Washington, DC, USA, 2003. IEEE Computer Society.

[KS03b] Ho-Seop Kim and James E Smith. Hardware support for control transfers in code caches. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 253. IEEE Computer Society, 2003.

[Lan11] Travis Lanier. Exploring the design of the Cortex-A15 processor, 2011. `http://www.arm.com/files/pdf/AT-Exploring_the_Design_of_the_Cortex-A15.pdf` (Visited on 13/07/2016).

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[NS07a] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74. ACM, 2007.

[NS07b] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

[PG10] Mathias Payer and Thomas R Gross. Generating low-overhead dynamic binary translators. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, page 22. ACM, 2010.

[SFY14] Gaurav Singh, Greg Favor, and Alfred Yeung. AppliedMicro X-Gene2. In *HotChips*, 2014.

[SKC+04] Kevin Scott, Naveen Kumar, Bruce R Childers, Jack W Davidson, and Mary Lou Soffa. Overhead reduction techniques for software dynamic translation. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 200. IEEE, 2004.

[SN05] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[SSB07] Swaroop Sridhar, Jonathan S Shapiro, and Prashanth P Bungale. HDTrans: a low-overhead dynamic translator. *ACM SIGARCH Computer Architecture News*, 35(1):135–140, 2007.

[SSNB06] Swaroop Sridhar, Jonathan S Shapiro, Eric Northup, and Prashanth P Bungale. HDTrans: an open source, low-level dynamic instrumentation system. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 175–185. ACM, 2006.

[WHK+07] Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar R Nair, Mauricio Breternitz Jr, Zhiwei Ying, and Youfeng Wu. StarDBT: an efficient multi-platform dynamic binary translation system. In *Asia-Pacific Conference on Advances in Computer Systems Architecture*, pages 4–15. Springer, 2007.

[WR96] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '96, pages 68–79, New York, NY, USA, 1996. ACM.

Appendix A

Example plugin

The plugin shown in Listing A.5 uses the API of MAMBO to modify the instruction stream: for each table branch instruction (Table Branch Byte - TBB or Table Branch Halfword - TBH) in the scanned code, it inserts an inlined code snippet which temporarily stores the required context on the stack; it loads, increments and stores back a 64-bit counter; and finally it restores the context of the application. This is implemented using a callback registered for the *pre_inst* event. This callback is called for each instruction that MAMBO scans, but before the translation has been inserted in the code cache. This allows the plugin to insert its own code before the translated instruction.

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

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

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

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

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

Listing A.5: Example plugin: dynamic execution counter for TBB and TBH instructions

Appendix B

The full evaluation results

These are the full performance results obtained in the evaluation of the microarchitectural optimisations in Section 5.4.

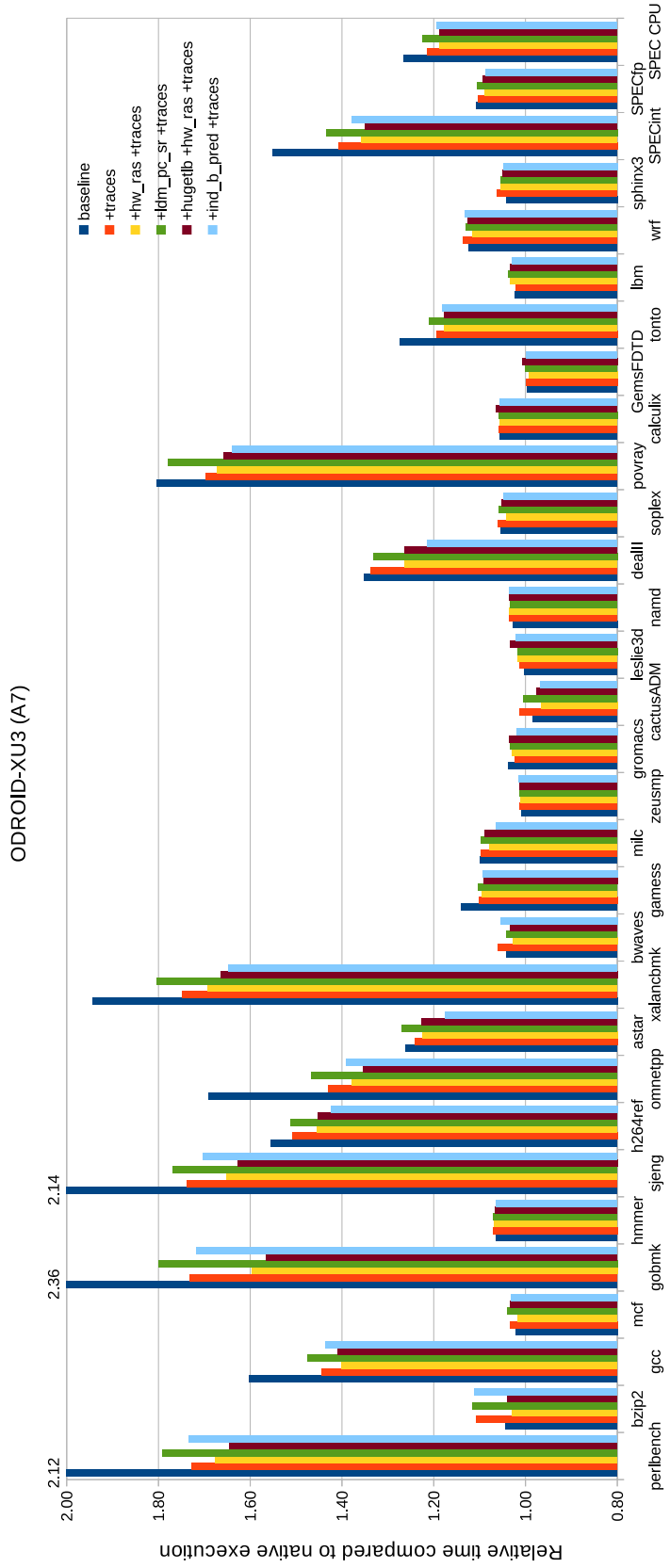


Figure B.1: Relative execution time for SPEC CPU2006 with the *ref* dataset on ODROID-XU3 - with microarchitectural optimisations.

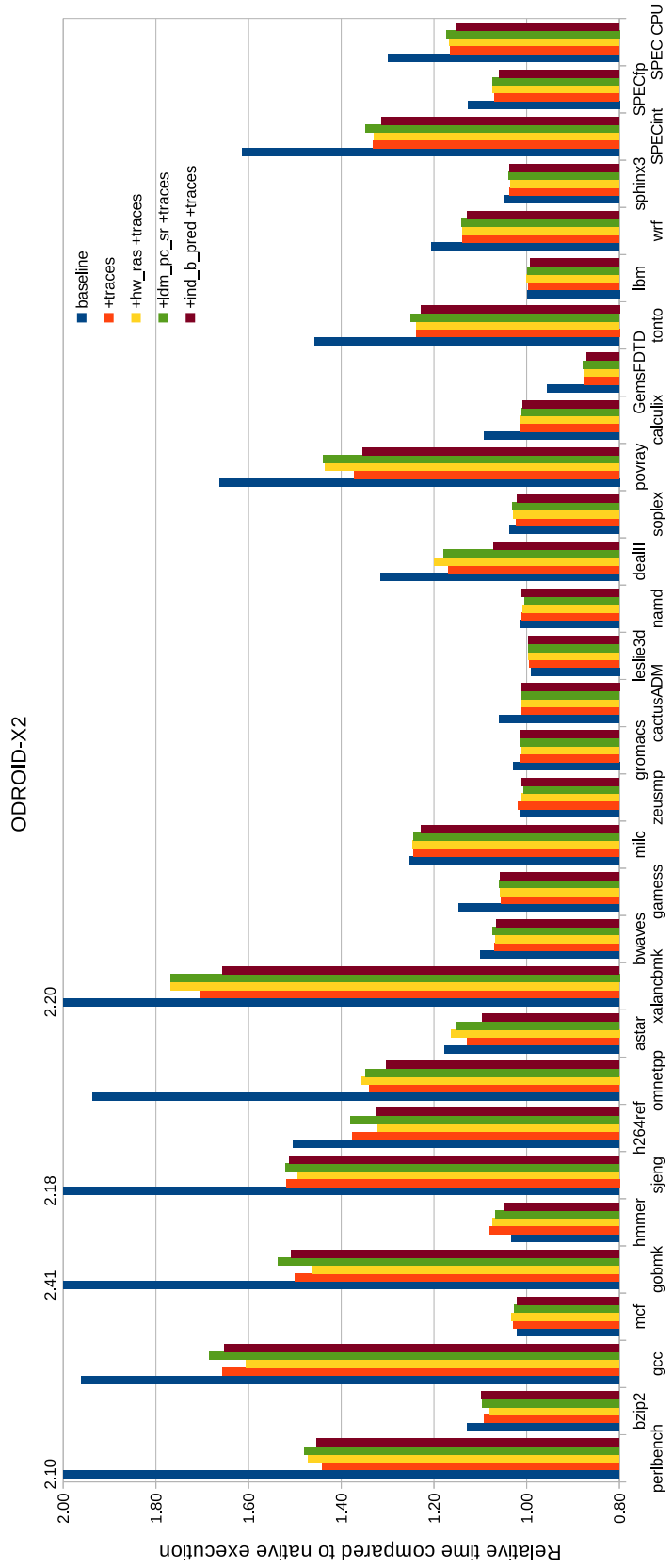


Figure B.2: Relative execution time for SPEC CPU2006 with the *ref* dataset on ODROID-X2 - with microarchitectural optimisations.

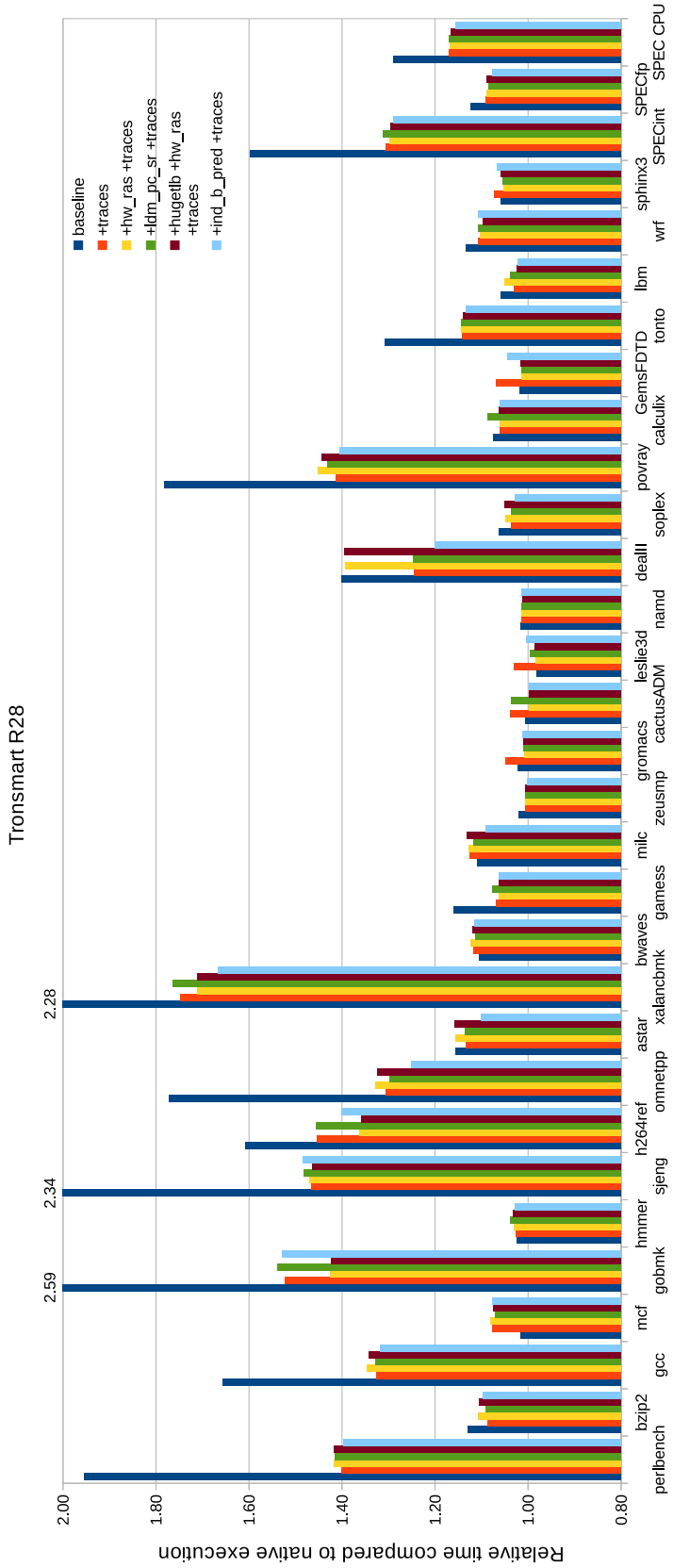


Figure B.3: Relative execution time for SPEC CPU2006 with the *ref* dataset on Tronsmart R28 - with microarchitectural optimisations.

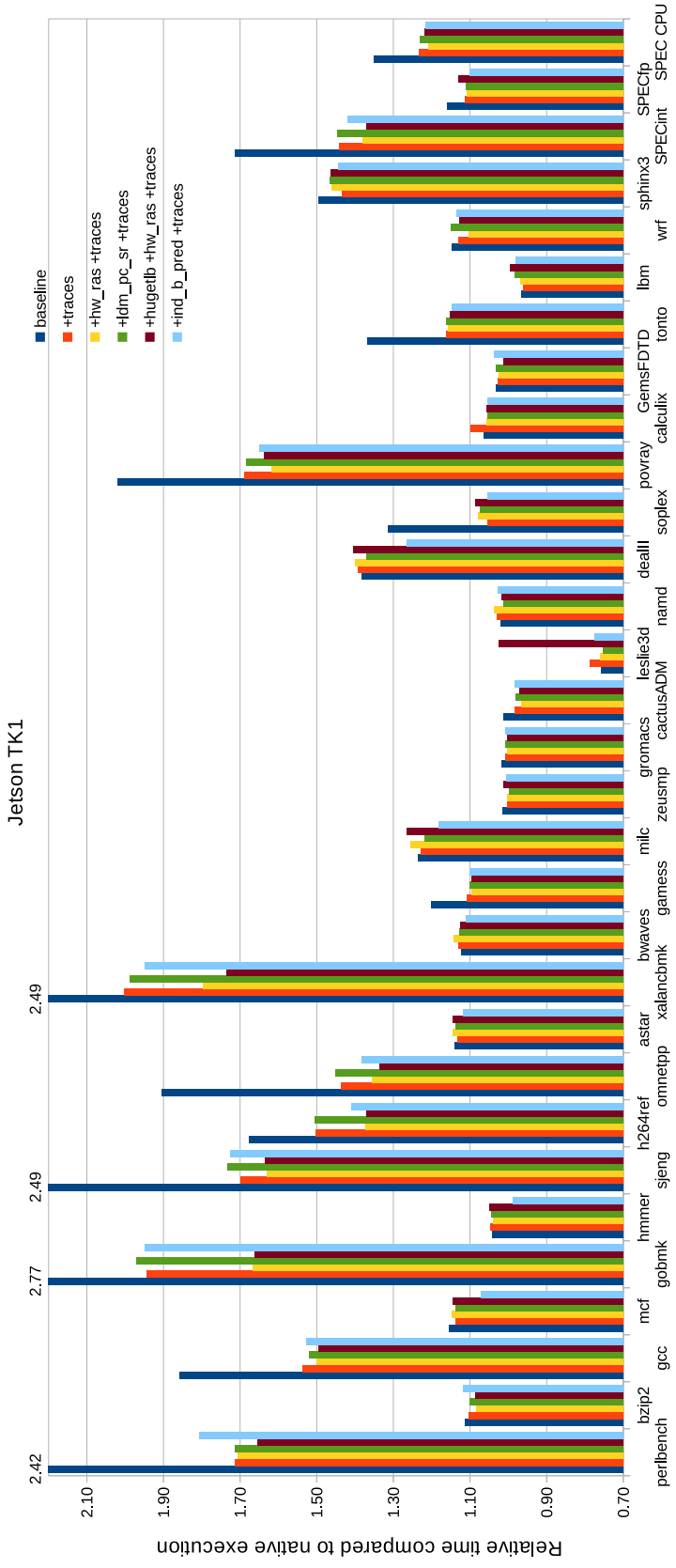


Figure B.4: Relative execution time for SPEC CPU2006 with the *ref* dataset on Jetson TK1 - with microarchitectural optimisations.

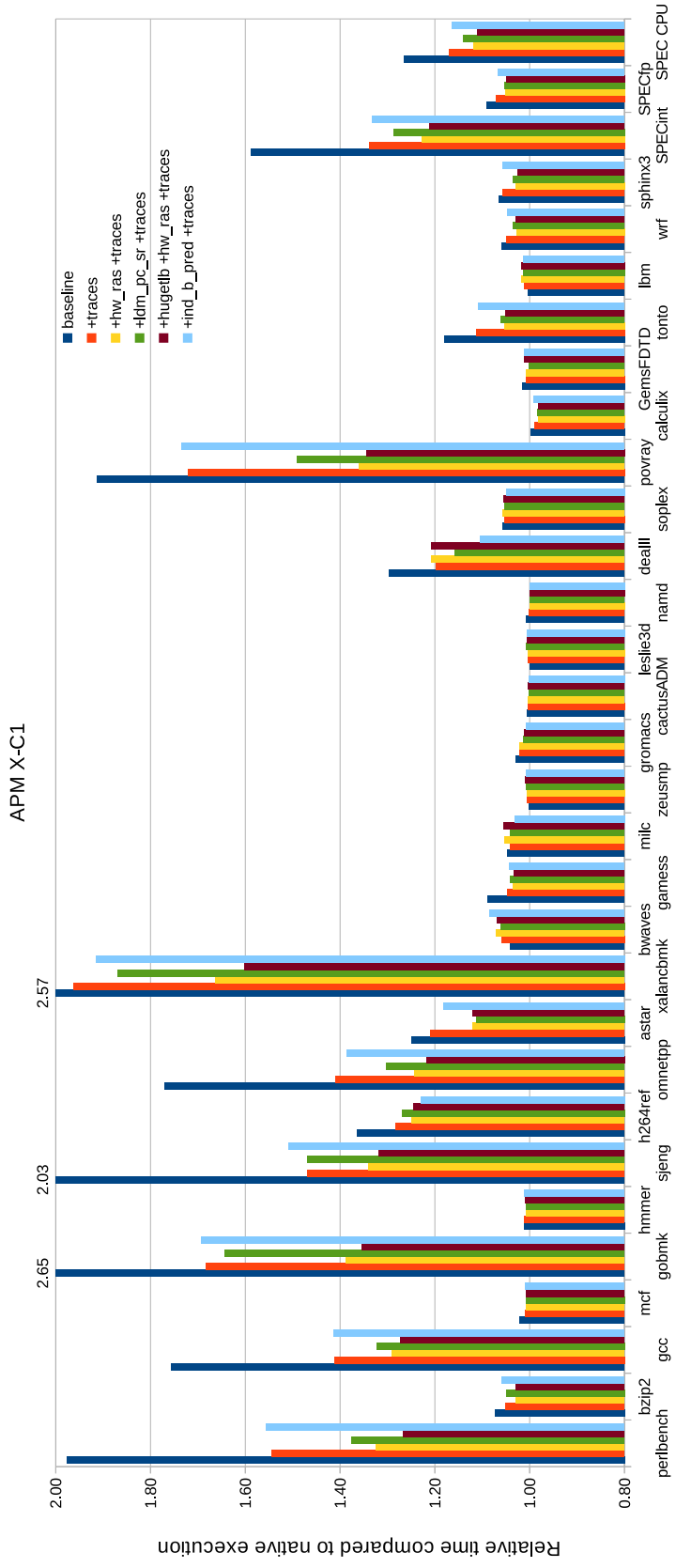


Figure B.5: Relative execution time for SPEC CPU2006 with the *ref* dataset on APM X-C1 - with microarchitectural optimisations.

Appendix C

Raw performance counter values

These are the raw performance counter values obtained by **natively** running the SPEC CPU2006 benchmarks used in the evaluation under the *perf stat* tool. These values can be used in addition to the relative changes shown in Section 5.4.4 when it is desired to determine hit and miss rates for a specific subsystem, for example. Note that these measurements have only been collected for the benchmarks which have a speed-up or slowdown higher than 15% in the *baseline* configuration of MAMBO compared to native execution.

Refer to Section 5.4.4 for the description of each event and any differences in measurement between the systems.

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	3,690,325,468	2,255,654,058	1,000,097,844	10,862,963	1,476,441,317	12,239,541	34,068,740	1,769,124	1,902,376	423,083	222,276,466	37,079,618
gcc	3,846,132,516	1,499,506,926	537,985,655	14,620,196	921,449,000	4,821,057	37,829,334	7,440,854	1,332,620	113,651	195,921,682	21,924,195
gobmk	3,524,352,529	1,968,819,357	778,212,573	12,076,407	1,495,802,883	15,917,782	40,246,232	2,037,365	481,092	91,929	183,143,333	81,787,011
sjeng	4,715,769,249	2,912,710,206	1,038,056,615	12,008,396	2,265,209,632	9,358,293	33,380,053	1,791,676	1,362,890	68,250	253,509,981	124,057,490
h264ref	5,982,552,369	4,149,274,955	2,535,575,899	17,597,956	2,054,215,898	3,506,430	39,277,937	4,801,544	1,218,269	77,428	316,855,404	23,449,963
omnetpp	3,130,035,321	599,913,093	258,691,687	19,950,398	481,122,570	1,789,768	41,693,697	11,347,091	5,237,166	657,390	85,318,171	19,189,808
astar	3,379,153,892	1,173,554,370	495,323,227	23,073,631	849,983,926	393,901	46,560,352	9,090,519	7,726,703	1,787	107,630,400	36,605,706
xalancbmk	2,982,431,874	1,111,900,059	361,063,790	19,968,726	650,599,075	2,830,111	43,149,661	7,479,865	4,089,139	467,517	127,378,383	15,739,802
dealII	4,795,041,123	2,179,895,790	708,017,730	21,122,246	1,220,042,959	1,218,917	43,583,333	8,593,370	348,650	45,781	265,472,607	13,325,493
povray	1,966,480,792	958,174,993	460,960,783	11,165,346	718,404,308	6,216,564	28,553,787	26,434	183,267	83,412	107,196,694	22,265,489
tonto	9,815,189,762	3,697,914,375	1,362,800,194	32,748,435	2,099,643,594	5,597,429	72,483,554	11,873,297	450,847	75,130	249,828,207	33,408,257

Table C.1: The raw performance counter values for the native execution of SPEC CPU2006 on ODROID-XU3 (thousands).

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-m	dtlb-m	itlb-m	br	br-m
perlbench	2,815,755,601	2,501,847,665	1,033,007,602	9,862,797	11,844,645	19,737,955	1,543,992	223,529,357	34,678,361
bzip2	4,274,908,232	2,756,683,787	1,139,199,335	36,148,464	88,497	22,687,108	13,656	176,516,775	26,157,150
gcc	2,471,368,972	1,570,401,715	403,804,940	16,690,833	4,785,611	10,521,025	474,029	192,582,150	19,253,469
gobmk	3,045,579,461	2,466,209,613	851,967,413	11,932,162	17,889,037	18,307,828	57,135	181,492,026	72,493,491
sjeng	4,151,571,112	3,673,627,317	1,136,913,786	9,982,919	5,980,475	26,306,870	17,863	253,300,729	112,303,750
h264ref	4,639,583,977	4,227,578,613	2,411,666,672	23,330,923	2,050,185	9,391,644	37,668	315,989,160	16,032,457
omnetpp	2,635,323,407	698,580,807	295,734,977	20,182,257	664,695	25,663,682	261,030	85,006,703	19,581,671
astar	3,014,912,902	1,452,102,341	552,422,578	22,441,759	62,130	14,732,420	13,659	108,525,660	26,626,309
xalancbmk	2,158,716,362	1,176,570,381	379,048,517	23,175,074	4,587,812	15,631,611	1,859,041	127,033,068	18,588,703
milc	4,009,538,523	1,114,293,575	449,608,610	43,286,231	146,361	2,977,108	160,697	57,176,972	682,451
dealII	3,217,795,455	2,254,522,911	710,584,008	39,147,036	426,812	5,861,722	59,949	264,680,741	22,202,544
povray	1,814,163,945	1,085,518,603	493,182,598	10,680,732	6,504,803	8,621,725	541,474	107,440,270	19,576,414
tonto	5,596,497,544	3,697,510,833	1,269,705,728	55,196,715	2,712,861	8,535,044	396,538	244,109,365	20,600,398
wrf	5,793,295,984	3,930,787,382	1,146,965,142	79,211,356	415,522	3,465,268	131,556	207,770,767	8,309,588

Table C.2: The raw performance counter values for the native execution of SPEC CPU2006 on ODROID-X2 (thousands).

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	2,168,767,505	2,253,036,197	964,325,146	9,394,109	805,663,534	4,421,670	15,172,706	1,237,948	16,013,148	951,928	223,272,058	21,954,908
gcc	2,025,059,232	1,446,824,663	381,266,976	13,249,741	364,520,984	1,667,436	19,201,278	6,769,026	6,643,389	429,251	192,068,767	10,641,345
gobmk	2,250,835,358	1,967,180,637	776,404,010	12,014,182	812,084,659	7,092,155	21,825,722	1,318,549	15,508,636	67,225	181,300,127	40,636,842
sjeng	3,012,537,578	2,904,914,551	990,639,390	11,152,120	1,142,850,742	1,984,026	14,297,849	1,468,681	20,653,985	336	253,162,182	49,286,264
h264ref	3,132,101,576	4,139,324,473	2,161,595,961	16,400,455	762,330,738	717,052	24,663,893	2,758,848	8,585,194	21,108	315,777,005	14,107,530
omnetpp	2,187,896,900	602,904,679	309,152,905	20,608,243	274,606,319	206,418	22,108,245	11,009,500	23,887,139	86,131	84,955,321	6,333,983
xalancbmk	1,556,378,206	1,120,952,411	408,530,333	16,330,159	236,002,694	1,005,471	22,039,424	5,805,145	12,893,907	1,744,595	126,917,594	7,155,372
gamess	4,793,126,462	7,429,543,372	2,634,679,352	29,047,669	1,569,169,353	1,066,136	34,648,716	348,612	93,889,936	3,256	233,859,529	14,621,064
dealII	1,733,460,769	2,167,379,153	688,752,433	21,073,043	574,997,602	103,323	38,403,805	5,944,837	2,699,432	36,105	264,520,962	10,629,562
povray	1,163,916,244	968,847,199	445,304,512	11,830,498	393,722,143	2,461,665	15,130,308	24,792	6,188,141	534,898	107,364,597	7,653,809
tonto	3,901,595,691	3,631,278,029	1,193,278,554	31,910,223	746,806,504	1,027,706	54,180,894	7,540,656	7,872,678	305,683	243,782,389	9,598,556

Table C.3: The raw performance counter values for the native execution of SPEC CPU2006 on Tronsmart R28 (thousands).

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m	br	br-m
perlbench	2,118,480,940	2,249,707,882	752,048,593	8,980,308	698,117,561	8,710,503	21,732,464	1,216,636	12,150,437	1,322,294	458,821,849	12,488,348
bzip2	2,732,226,671	2,538,912,900	851,337,009	23,316,724	744,188,094	288,896	40,482,169	3,643,425	12,871,476	113,362	398,653,587	19,905,785
gcc	1,876,504,654	1,450,771,425	281,667,054	11,087,930	345,879,852	2,780,627	23,450,592	5,497,961	6,588,356	871,611	306,873,127	9,112,055
gobmk	2,385,090,836	1,967,180,700	610,076,223	10,632,389	1,000,664,650	10,600,619	28,793,826	457,546	10,031,633	309,140	357,964,065	44,001,466
sjeng	3,292,060,176	2,908,622,699	788,915,176	9,801,911	1,279,591,626	6,826,186	24,365,547	1,200,646	15,922,106	222,747	530,509,921	49,163,044
h264ref	3,241,562,001	4,142,481,979	1,825,345,339	19,086,023	978,202,852	1,850,493	32,731,259	917,269	6,695,404	186,325	404,677,836	11,220,020
omnetpp	2,385,534,690	597,605,991	188,473,768	14,603,693	250,606,707	1,642,890	26,420,935	7,299,341	19,528,101	286,908	166,811,123	5,472,346
xalancbmk	1,667,922,937	1,110,291,141	303,882,355	15,620,335	261,643,258	2,458,525	29,784,576	4,978,723	11,569,883	1,458,015	296,472,672	5,980,174
gamess	5,333,438,827	7,436,407,583	2,210,687,529	33,654,119	1,824,529,462	2,221,326	53,474,750	87,027	56,558,700	255,011	401,462,187	14,068,189
milc	2,923,015,642	1,132,711,771	352,344,240	20,042,197	285,451,052	489,433	37,547,761	23,203,837	2,380,807	244,168	77,611,112	385,623
leslie3d	3,923,789,938	2,078,563,933	583,436,777	83,395,570	345,762,527	373,983	152,738,530	26,059,609	1,622,673	147,637	122,774,429	1,058,851
dealII	1,869,086,471	2,169,788,952	585,496,850	19,948,778	538,733,928	523,577	40,342,755	5,470,150	4,287,744	157,613	400,104,751	9,303,474
soplex	1,995,235,573	850,094,371	262,142,090	24,943,445	255,710,191	249,814	51,530,347	20,871,742	4,015,777	116,076	125,581,449	6,089,528
povray	1,038,052,554	955,896,562	351,879,320	11,607,134	360,559,270	3,502,942	17,369,218	2,408	5,096,517	676,573	172,098,812	6,271,126
tonto	4,267,856,090	3,628,224,150	932,804,529	23,584,143	921,119,034	2,287,193	51,546,401	4,634,693	6,825,843	898,021	395,978,019	12,350,902
sphinx3	4,317,078,877	3,780,959,416	873,298,213	50,044,017	819,893,160	588,883	107,591,074	41,047,238	10,088,672	175,266	313,858,445	14,431,067

Table C.4: The raw performance counter values for the native execution of SPEC CPU2006 on Jetson TK1 (thousands).

Benchmark	Cycles	Insts	L1d-l	L1d-m	L1i-l	L1i-m	L2-l	L2-m	dtlb-m	itlb-m
perlbench	2,520,106,048	2,245,395,251	1,183,838,376	8,436,096	755,369,805	6,197,691	198,176,717	2,463,757	134,595	5,988
gcc	2,107,242,303	1,447,655,775	653,121,583	10,342,843	492,770,737	1,869,990	126,648,843	8,993,241	589,646	3,383
gobmk	2,654,963,492	1,963,229,930	1,024,663,026	5,787,026	932,737,664	7,342,406	168,771,329	3,177,745	80,059	2,036
sjeng	3,756,174,373	2,903,627,121	1,252,758,465	4,876,171	1,272,166,730	917,219	191,465,805	2,435,304	843,408	4,300
h264ref	4,203,087,556	4,139,876,485	3,013,954,577	10,250,889	917,634,316	666,064	356,462,112	5,909,471	239,263	3,245
omnetpp	2,147,649,392	593,759,385	340,205,967	14,245,003	304,473,802	177,213	74,845,173	14,075,923	2,786,365	22,913
astar	2,506,341,256	1,166,529,496	729,092,673	13,743,938	699,565,596	40,777	93,747,895	11,397,448	4,001,454	443
xalancbmk	1,439,543,532	1,110,057,362	437,194,965	14,669,555	377,742,010	889,747	54,768,435	8,693,653	1,067,476	22,654
dealII	2,400,600,161	2,167,438,971	751,933,478	19,757,600	719,213,113	109,185	92,129,686	14,024,577	125,591	202
povray	1,512,631,808	954,360,864	625,803,161	9,466,681	381,331,095	1,595,696	91,636,915	81,807	11	8
tonto	5,920,119,194	3,618,640,681	1,484,005,028	28,604,646	825,142,901	1,077,524	285,447,366	13,860,932	51,361	2,142

Table C.5: The raw performance counter values for the native execution of SPEC CPU2006 on APM X-C1 (thousands).