THE UNIVERSITY
of EDINBURGH

Applications of Information Sharing for Code Generation in Process Virtual Machines

Stephen Kyle

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2015

Abstract

As the backbone of many computing environments today, it is important that process virtual machines be both performant and robust in mobile, personal desktop, and enterprise applications. This thesis focusses on code generation within these virtual machines, particularly addressing situations where redundant work is being performed. The goal is to exploit information sharing in order to improve the performance and robustness of virtual machines that are accelerated by native code generation. First, the thesis investigates the potential to share generated code between multiple threads in a dynamic binary translator used to perform instruction set simulation. This is done through a code generation design that allows native code to be executed by any simulated core and adding a mechanism to share native code regions between threads. This is shown to improve the average performance of multi-threaded benchmarks by 1.4x when simulating 128 cores on a quad-core host machine. Secondly, the ahead-of-time code generation system used for executing Android applications is improved through the use of profiling. The thesis investigates the potential for profiles produced by individual users of applications to be shared and merged together to produce a generic profile that still provides a lot of benefit for a new user who is then able to skip the expensive profiling phase. These profiles can not only be used for selective compilation to reduce code-size and installation time, but can also be used for focussed optimisation on vital code regions of an application in order to improve overall performance. With selective compilation applied to a set of popular Android applications, code-size can be reduced by 49.9% on average, while installation time can be reduced by 31.8%, with only an average 8.5% increase in the amount of sequential runtime required to execute the collected profiles. The thesis also shows that, among the tested users, the use of a crowd-sourced and merged profile does not significantly affect their estimated performance loss from selective compilation (0.90x-0.92x) in comparison to when they they perform selective compilation with their own unique profile (0.93x). Furthermore, by proposing a new, more powerful code generator for Android's virtual machine, these same profiles can be used to perform focussed optimisation, which preliminary results show to increase runtime performance across a set of common Android benchmarks by 1.46x-10.83x. Finally, in such a situation where a new code generator is being added to a virtual machine, it is also important to test the code generator for correctness and robustness. The methods of execution of a virtual machine, such as interpreters and code generators, must share a set of semantics about how programs must be executed, and this can be exploited in order to improve testing. This is done through the application of domain-aware binary fuzzing and differential testing within Android's virtual machine. The thesis highlights a series of actual code generation and verification bugs that were found in Android's virtual machine using this testing methodology, as well as comparing the proposed approach to other state-of-the-art fuzzing techniques.

Lay Summary

Many programs today are written in expressive programming languages that often rely on programs called virtual machines (VMs) to execute their code. This differs from the past where the human-readable code had to be first translated (or "compiled") into the binary code understood by that computer's processor before it could be executed. Removing this translation improved the speed at which program designs could be iterated on, and provided new opportunities for improvements in language design. The VM is so-called because it could be said to be simulating a virtual computer executing an input program - some VMs even simulate real computers, allowing a person to run a program even if they lack the required hardware.

Programs run in VMs are often slower than their compiled cousins, and so the idea of producing pieces of binary code that mimic the function of parts of these programs as they are actually being executed is a well-established technique that improves performance. Some of the work done in order to produce this binary (or "native") code is a waste of resources, and this thesis explores opportunities to mitigate this, improving both the performance and reliability of VMs. In all cases there is identification of information that is or could be shared, and this fact is exploited to improve the VM.

First in this thesis, the VMs that simulate future processors (that may contain many more internal processing cores than today) are improved. In this VM, each core is simulated in an independent executing "thread", and does its own work to identify which parts of the input program should be translated to native code. The translation itself is performed by a separate thread, and this translator is modified to recognise when multiple cores request translation of the same code. By doing this, the translation need only be performed once, allowing all cores to finish faster.

Secondly, the VM that runs Android applications is improved. Whenever an Android app is installed, the entire program is translated to native code, when usually only a fraction of the program needs to be translated without an impact to performance, meaning valuable storage space is being wasted on the device. The trick is knowing which fraction needs to be compiled, and this normally requires the app to be run first, which is difficult if the code is generated as the app is installed. The thesis investigates the possibility of using the knowledge gained from other users running an app to determine how an app should be compiled for a new user. It shows that, on average, there is little drawback to using this information - the new user's app will perform as well as the others, regardless of how they use the app.

Finally, the same Android VM is targeted for reliability improvements, by exploiting the fact that VMs usually have multiple of ways of execution when running an input program, many involving translation. It should normally be expected that all ways produce the same result, but by randomly generating test cases and running them with the variety of methods of execution, differences were seen that lead to the discovery of real defects in Android's VM.

Acknowledgements

First, thanks to my two supervisors, Dr. Björn Franke and Dr. Hugh Leather. Without your endless patience, support, and good humour I would've never made it all the way to the end, and in my unbiased opinion you two probably deserve some sort of medal for getting me there.

Thanks to the people who've haunted the PASTA office throughout the years (and the problem with taking so long to finish this thesis, is that this becomes a long list...) - Tobias, Oscar, Matthew, Igor, Chris T, Chris M, Harry, Volker, Spink, `pjm`, Erik, Thibault, Damon, Zheng, Daniel, you've all been good office-mates providing interesting discussions, hopefully that's a reciprocal evaluation? Especially I'd like to thank Tobias for all the fun times in Austin, Chris for helping me to realise one of my dreams of making an award-winning video game, and Harry, maybe for showing me that there was always a better way to do something, but also just for being a good friend.

I'd like to thank Mario Guerra for offering me an internship with him in Austin, and for helping me out in one of my lowest moments.

From ARM, thanks to Matt Evans for helping me get where I am today, and also to Dave Butcher and Stuart Monteith for their support and advice during my internship there.

Thanks to friends near and far who've supported me through these years, but also read various drafts of this thesis, Rachael Smith, Joe Bazalgette, Sarah Wiseman, Robert Künnemann, Chris Thompson, Robert Peacock, Simon Vansintjan, Richard Littauer. Joe, you deserve special mention for all your help.

Thanks to all my family for giving me all the opportunities in life that they could, but especially to my granddad for sowing the seeds of my interests, and always being interested in my work.

Further special thanks to Rachael for helping me to overcome myself, and always being there for me. I know I say this every day, but you really helped.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own, except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

S. Kyle, I. Böhm, B. Franke, H. Leather and N. Topham. Efficiently Parallelizing Instruction Set Simulation of Embedded Multi-Core Processors Using Region-based Just-in-Time Dynamic Binary Translation. In *Languages, Compilers, Tools and Theory for Embedded Systems,* LCTES'12, June 2012.

S. Kyle, H. Leather, B. Franke, D. Butcher and S. Monteith. Application of Domain-aware Binary Fuzzing to Aid Android Virtual Machine Testing. In *Virtual Execution Environments* VEE'15, March 2015.

(Stephen Kyle)

To Dad and Granny

Table of Contents

Chapter 1

Introduction

Process virtual machines[1] are a fundamental component of many popular computing environments today. JavaScript, the implementation language of many complicated web applications used by millions of people, is executed within a virtual machine in a web browser. Android, the mobile operating system that was shipped on over a billion devices in 2014 [Str15], executes its applications within a virtual machine. Oracle report that over 97% of enterprise desktop computers have a Java runtime environment installed on them [Ora], running Java applications within a virtual machine. There are many popular programming languages that run within virtual machines - GitHut [Str] frequently tracks the popularity of programming languages by ranking them by the number of active repositories they have on GitHub, a popular project hosting provider. Eight of GitHut's top ten ranked languages in Q4 2014 are primarily executed in virtual machines. These languages are JavaScript, Java, Python, CSS, PHP, Ruby, Shell and C# - only C and C++ are not commonly run within some kind of a process virtual machine.

With presence in mobile, personal desktop, and enterprise environments, it is important that programs executed in these virtual machines perform well. Responsiveness is often a concern, for example in interactive applications or when rendering web pages. Energy consumption is a concern for battery-powered mobile devices, that performance improvements can often assist with. "Race-to-idle" [AA12] is a scheme that helps to save energy that can always be enhanced by performance improvements. The techniques presented in this thesis are intended to help improve the performance of these pervasive virtual machines, while also taking measures to ensure such improve-

[1] In this thesis, the terms process virtual machine (PVM), virtual machine (VM), and virtual execution environment may be used interchangably. A process virtual machine refers specifically to software that is designed for the purpose of executing a single other program within it.

ments do not cause the virtual machines to become incorrect. The work presented in this thesis will primarily exploit the sharing of information to reduce the amount of wasted effort during certain aspects of code generation in VMs, namely the actual construction of native code, code profiling, and code generator testing.

1.1 Code Generation for Process Virtual Machines

Process virtual machines (PVMs) can be categorised into two main varieties. High-level language virtual machines (HLLVMs) are VMs designed to execute programs written in high-level languages (such as Java, JavaScript, or Python) that have possibly been compiled to a bytecode format more amenable to efficient execution by the VM. Alternatively, emulator virtual machines (EVMs) are VMs used for the execution of programs that have been compiled with the intention to be executed directly on a processor. The VM in this case is designed to emulate the functionality of the target processor. Many PVMs of both varieties are initially developed with an interpreter intended for the sequential fetching and execution of statements or instructions that comprise the program.

Sequences of code executed in interpreters were found to perform poorly in comparison to equivalent natively compiled and executed code, and this lead to the adoption of *just-in-time* (JIT) compilation in VMs. This is a form of code generation where a sequence of bytecode instructions are translated to equivalent machine code as the program is being executed. In the case of EVMs, this technique is often called *dynamic binary translation* or *dynamic recompilation*. While there are many strategies surrounding how and when to perform code generation in concert with (or even instead of) interpretation, this ultimately allowed implementations of high-level dynamic languages, such as PyPy for Python [BCFR09] or V8 for JavaScript [Tea], to execute languages that could offer features like dynamic typing and removal of explicit compilation phases with performance approaching that of native execution.

1.2 Wasted Effort During Code Generation

There are many aspects of code generation where wasted effort can be identified. For example, as a program is executed in a virtual machine, sequences of bytecode instructions may be JIT compiled as they are reached, or once they have exceeded an execution threshold. These fragments of native code are typically stored in a code cache that

is lost when the application terminates. This would require the re-generation of this code on subsequent executions, and so a mechanism for storing native code fragments for immediate use in future executions was able to reduce this wasted effort [RCCS07].

Other issues with JIT code generation have been noted [RBJ01], such as the overhead of blocking to produce native code for a bytecode sequence that will only be executed a few times, when it would have been faster to simply interpret the sequence. Radhakrishnan et al. also show that compilation can form a significant fraction of the execution time of particular Java benchmarks, and so propose hardware translation to overcome some of these issues. This thesis will investigate an opportunity to lower the amount of redundant JIT compilation performed in an EVM, in order to improve performance by reducing wasted effort.

Another example of wasteful code generation can be seen in the generation of native code on new Android mobile devices. This is performed by millions of users who all download the same application bytecode and generate identical native code on the device. This wasteful process could be better justified if a mobile device performed profiling in order to produce a uniquely optimised binary tailored for the user's use of an application, but then what is the probability this profile will be highly similiar to other users' profiles? Potential exists here to remove the wasted effort of profiling for every user, and this thesis will investigate this possibility.

Additionally, the development of new code generators can waste developer time when trying to identify virtual machine bugs. Testing of VMs could be improved by exploiting the fact that code generators and interpreters in virtual machines share a common set of semantics about how programs are executed. This thesis will investigate how this fact can be combined with existing testing techniques in order to improve the correctness of code generators.

This thesis explores three topics concerning code generation and in all of these cases, it is the exploitation of information sharing that will allow improvements in code generation to be made through the reduction of wasted effort. In the first, the performance of code generation is improved through the sharing of information within a single VM. Secondly, the profiling used for code generation is improved through the sharing of information between multiple VMs. Finally, the testing of code generation is improved through the sharing of information between multiple code generators in a single VM.

1.3 Goals and Contributions

In summary, the goal of this thesis is to reduce redundant effort and exploit information sharing in order to improve the performance and correctness of process virtual machines that perform code generation. The three areas investigated are the actual generation of code itself, profiling to direct where and how to perform code generation, and the testing of code generators. Contributions arising from this work are as follows:

- An approach to generating JIT-compiled code regions that can be used by multiple threads wishing to execute the same region of a binary is presented, as well as a technique for sharing translated regions between threads within a dynamic binary translator.
- An evaluation is presented of performance improvements obtained when the proposed system for sharing generated code regions is used to execute the Splash-2 multi-core benchmark suite.
- An evaluation is presented of how profile-based selective compilation can reduce the code-size and installation times for standard Android benchmarks without unduly sacrificing performance.
- An investigation is made into into how use of different profile merging schemes will affect the estimated performance for Android users using popular applications when they selectively compile an application with a merged profile they did not participate in the creation of. This can remove the burden of profiling for many users. The analysis uses profiles gathered from real users' use of these applications.
- An evaluation of how these merged profiles can affect the installation time and code-size for popular Android applications is presented, which may be useful to low-end devices where compilation of the whole application is infeasible.
- An approach is presented where binary fuzzing is combined with differential testing to exploit the presence of multiple VM code execution engines in order to improve the testing of the VM used in Android. A comparison of this approach to both naive and intelligent mutative fuzzing systems is made.

- Characterisation is made of the binary mutations that were used to find bugs in Android's VM, with identification of which mutations could be beneficial when applying this technique to the testing of other VMs.

1.4 Publications and Impact

Several publications and software tools have arisen from the work performed in this thesis. Listed below are these in which the author was significantly involved:

- First-author of *Efficiently Parallelizing Instruction Set Simulation of Embedded Multi-Core Processors Using Region-based Just-in-Time Dynamic Binary Translation*, found in the proceedings of the conference **Languages, Compilers and Tools for Embedded Systems 2012**.
- First-author of *Application of Domain-aware Binary Fuzzing to Aid Android Virtual Machine Testing*, found in the proceedings of the conference **Virtual Execution Environments 2015**.
- Creator and maintainer of the *dexfuzz* utility, that can be used to stress-test the various code generators of the ART VM used in the Android mobile operating system. This tool is freely available for use in the Android Open Source Project, and has been used by developers of ART within their test infrastructure.

1.5 Summary

This thesis will investigate areas in which opportunities to share information can be exploited in order to improve multiple facets of code generation in process virtual machines. These topics are the actual generation of code itself, profiling to direct focal points for generation, and code generator testing. Improvements are driven primarily through the reduction of wasted effort, but the final chapter also provides a system for improving the correctness of code generators as they are developed.

To provide an overview of the remaining chapters in this thesis:

- In Chapter 2, background is provided on the subject of process virtual machines, considering both high-level language and emulator virtual machines, with a focus on the VMs used in this thesis. Additionally, an introduction to the concepts used in code profiling and VM testing is provided.

- In Chapter 3, the prior work undertaken in improving performance and correctness of process virtual machines is discussed, particularly focussing on recent improvements in the area of VMs for mobile devices, such as the ART VM that forms part of the Android OS.

- Chapter 4 presents an investigation into how to efficiently generate code in a dynamic binary translator intended for the simulation of a multi-core processor, reducing the redundant generation of native code.

- Chapter 5 presents a strategy for sharing profiling information between users of Android applications in order to improve code generation as new applications are installed on the device. These profiles can be used not only to reduce code-size or installation time, but also to improve the performance of applications if a more powerful code generator is used to focus compilation on vital code regions.

- Chapter 6 presents a method of testing new code generators – such as the one proposed in the previous chapter – as they are added to a process virtual machine. This can be done by exploitation of the fact that different methods of execution such as interpreters or code generators must share a common set of semantics with respect to how the process virtual machine executes programs.

- Finally, in Chapter 7, the findings of this thesis on improving code generation for process virtual machines are summarised and potential future work arising from this research is discussed.

Chapter 2

Background

This chapter presents an overview of the various topics concerning process virtual machines (PVMs) that will be discussed in this thesis. These topics include the classification of different types of PVM, methods of execution in PVMs, the ARCSIM dynamic binary translator, Android's Dalvik VM, program profiling for code generation, and code generator testing.

2.1 Process Virtual Machines

Virtualisation as a term has quite a broad scope, and the use of the term "process virtual machine" in this thesis comes from the taxonomy of virtual machines proposed by Smith and Nair. [SN05] A *process virtual machine* is a program that provides a virtual environment for a single application to execute within, as opposed to virtualising a complete system for one or more operating systems to run in.[1] The virtual machine provides an interface that allows the executing program to access features such as network data transmission, graphical output, and file access, while also placing restrictions on which programs can access these features. For example, a process virtual machine can be used as a sandbox in order to prevent a running application from accessing files on the system. This sandboxing of programs for secure execution is one of the strengths of virtualisation.

Smith and Nair further distinguish process virtual machines by whether or not they execute an instruction set architecture (ISA) that differs from the ISA of the physical system being used for execution, so called *same-ISA* and *different-ISA* process virtual machines. This thesis primarily focusses on different-ISA PVMs, subdividing them

[1] Such systems are called *system virtual machines*.

further into high-level language VMs and emulator VMs.

2.1.1 High-level Language Virtual Machines

In a *high-level language virtual machine* (HLLVM), the executed program is usually written in a high-level programming language, and then possibly translated into a binary format for execution within the virtual machine. This binary format is typically a series of bytecode instructions.[2] These instructions can be considered to conform to a *pseudo* ISA that differs from the ISA of the host machine. Other high-level languages may not have an explicit pre-translation step, and will perform the compilation automatically within the HLLVM when provided with the program in its original source code form. This is common for scripting languages such as Python that promote quick iteration during development. Other languages may delay parsing of all functions or methods of a program's source until the method or function is called, such as JavaScript.

Once the input program has been loaded, the VM will execute the program from a standard entry point. In the case of scripting languages, this will typically be some anonymous method or function that contains all program statements written in the outermost scope of the program. As the program executes, the VM tracks how its state changes, with two common designs used for representing this state and determining the internal representation of the program that is to be executed:

- In a **stack-based design**, all intermediate values are stored on a stack, which is implicitly manipulated by the bytecode instructions. Such designs usually also include the concept of "local variables", where intermediate values can also be stored for later use. See the first example in Figure 2.1 for an example of stack-based bytecode in operation.

- A **register-based design** exclusively uses virtual registers as locations for value storage. (Local variables used in a stack-based design could be considered to be analogous to virtual registers.) Bytecodes designed for this architecture must explicitly provide operands to operate on. See the second example in Figure 2.1 for an example of stack-based bytecode in operation.

While a stack-based architecture usually contains smaller sized instructions, and therefore produces sequences with a smaller code-size, it typically requires a greater

[2] Some languages may not define a standard for a bytecode format that is to be used for program execution, and the implementation of the HLLVM is therefore free to use any format.

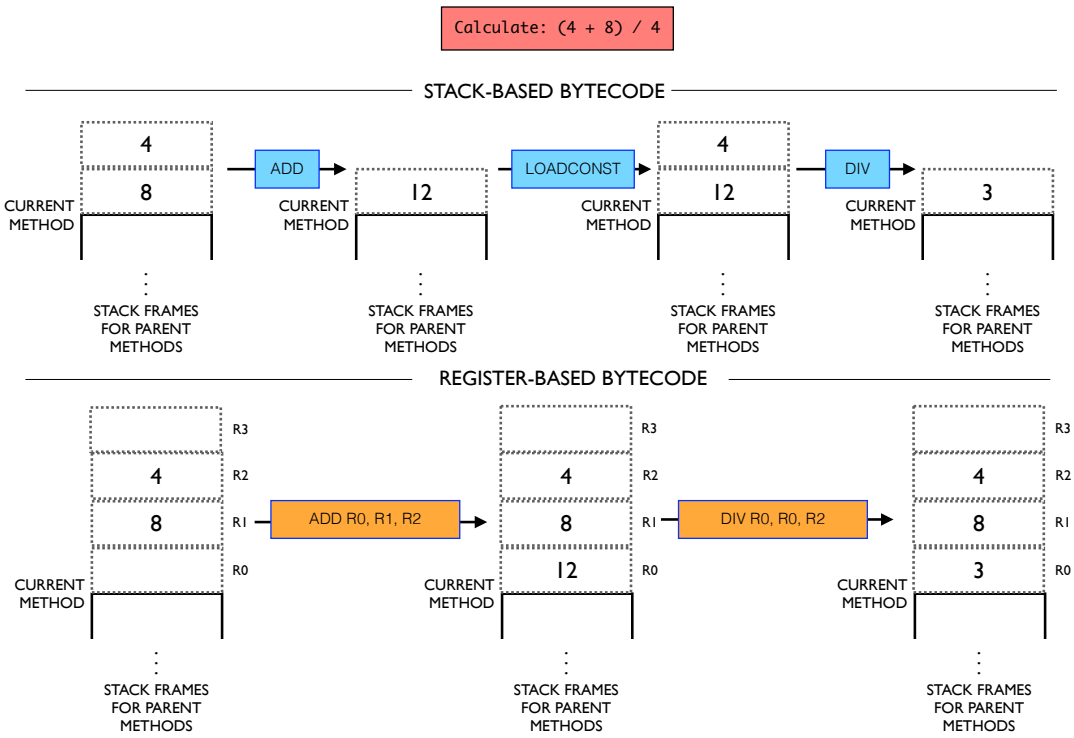


Figure 2.1

Comparison of stack vs. register bytecode, showing how $(4+8)/4$ might be calculated. The initial loads of the 4 and 8 constants in both cases are not shown, but register-based bytecode requires two instructions with explicit operands, while stack-based bytecode requires three, albeit shorter, instructions.

number of instructions in order to perform a comparable sequence of calculations expressed as register-based bytecodes, requiring more frequent decoding of bytecode [SCEG08]. Designs are not limited to these two architectures however, as for example some JavaScript engines will use a tree-based representation for execution.

In all designs, an overarching "stack" is still used to track the current method invocation hierarchy. Whenever a method is to be invoked, either the correct number of virtual registers or the maximum size of the operand stack used in the callee method is allocated as a new frame on the stack. This frame is then deallocated when the callee method returns.

Chapters 5 and 6 will explore the high-level language virtual machine called ART that is used on the latest Android mobile devices.

2.1.2 Emulator Virtual Machines

In an *emulator virtual machine* (EVM), the executed program is provided in the binary format that a physical processor would directly execute, merely a raw grouping of machine instructions. The EVM is designed to execute these instructions by emulating the functionality of the processor that normally executes them. This may involve modelling the state of the target processor, and the EVM may be designed to track statistics about the processor as the program is executed. An EVM could be used to allow a catalogue of legacy programs developed for previous generations of systems to still be used on modern hardware with a different ISA. Another use is for the execution of programs that are compiled for processors that are still in development. As there is a large lead time for the development of a new processor that may have its ISA standardised at quite an early point, it is useful to be able to have some software ported to the new processor by the time it is physically available, with some confidence that the software will work correctly with the processor.

An EVM will model all the hardware registers of a processor, as well as the memory space that the emulated processor can access. Execution takes place by fetching instructions, decoding them, and then updating the state of the modelled registers and memory according to the specification of the emulated ISA. Additionally, the EVM may also model micro-architectural details such as the operation of memory caches and execution pipelines. This allows for the more realistic simulation of the processor, introducing stalls from pipeline bubbles and memory access delays that will lead to a more accurate simulation of how the binary would be executed on the processor. This process is known as cycle-accurate instruction set simulation. This also enables the collection of micro-architectural statistics such as cache misses, and the overall number of cycles required to execute the program.

While an HLLVM-targeting program is likely to have a well-defined structure – usually a collection of methods with a clear delineation of code and data – this is rarely the case in binaries designed for direct execution on a processor. For this reason, some exploration of the binary is required as it is executed. In general, it is impossible to look at a particular sequence of bytes in such a binary file, and know with certainty that it contains a machine instruction. Consider also a case where a processor is capable of executing multiple ISAs, such as the family of ARM processors that can execute both the 32-bit ARM and mixed 16-/32-bit Thumb-2 ISAs [Lim07]. Binaries that target these processors can contain both ISAs, and byte sequences are not explicitly marked

for either ISA. The context of execution is key to knowing how to decode a particular sequence of bytes, and this is made difficult with both implicit and explicit ISA state switching available.

Chapter 4 of this thesis will explore an emulator virtual machine called ARCSIM that executes programs compiled for the ARCompact instruction set.

2.2 Interpretation

When creating a new PVM, an interpreter is the execution mechanism most likely to be created due to its ease of implementation. These are simple pieces of software that load a program's executable format and then perform a fetch-decode-update loop until the end of the program is reached. There are a number of designs for producing interpreters, the most basic being a switch-based fetch-decode-update system. Whether register-based or stack-based, all bytecode instructions will include an *opcode* that identifies which variety of state update is to be performed. For example, it may multiply two numbers, write a value to memory, invoke a method, branch to location x if y is non-zero, etc. After loading the instruction, the opcode is found and switched upon, as typically seen in a C switch statement. Execution then passes to the case that handles the implementation of that particular opcode, e.g., load two values from virtual registers or a stack, multiply them, and store the result somewhere. Determining the opcode of an instruction is not always as simple as checking one location in an instruction however, and real processor ISAs are notorious for requiring complex decode operations. After reaching the end of the opcode's implementation, control will jump back to the fetch stage again, fetching and decoding the next instruction found after the program counter was updated. Other designs for interpreters also exist, primarily threaded interpreters [Bel73, PR98] where the implementation of each opcode also contains the code necessary to decode the next instruction and then jump to its opcode's implementation. Each of these implementations are stored in a table designed for efficient retrieval. These interpreters were designed to overcome some of the inefficiencies of switch-based interpretation.

2.3 Just-in-time Compilation

While interpreters are a simple way to decode and execute instructions, they have been shown to suffer from poor performance in comparison to *just-in-time* (JIT) compilation

[RRJ+99]. While an interpreter takes a particular *source* ISA and executes it on a different *target* ISA, JIT compilation actually generates new code sequences in the target ISA. A mapping exists between a particular sequence of source instructions and a piece of native code, which updates the state of the VM according to the semantics of the source instruction sequence. In order to execute the source instructions, the VM need merely jump to the beginning of the native code chunk, which can be optimised and specialised in various ways to improve the rate of instruction execution for the VM. This results in a higher initial amount of work – the process of code generation – which can then be amortised with respect to interpretation if the generated code is executed frequently enough. Because of this initial investment into code generation, it may not always make sense to perform JIT compilation for a particular instruction sequence. For example, if a sequence is only executed once, then it will be faster to simply interpret it, instead of generating code and then executing the generated code.

A number of choices are available for the granularity of code sequence selected for code generation, called the "unit of compilation". The two most common examples are method-based and trace-based, however this thesis covers another style called region-based that can offer some advantages over the others.

2.3.1 Method-based JIT Compilation

In method-based compilation, entire code methods[3] are chosen for translation to native code. This is only available in VMs where the concept of a distinct method unit exists. Binaries intended for execution in EVMs often contain just a raw binary collection of mixed code and data, precluding functions from being statically identified. Even if the locations of function entry points are known, it is not possible to know with certainty which of the following bytes are instructions within the function. A method will often only be selected for JIT compilation once it has reached a certain threshold of executions, but this is not always the case. For example, the popular V8 Javascript HLLVM will immediately generate code for a method when it is reached for the first time, and forgoes the use of an interpreter. One drawback to method-based compilation is that while it aims to translate only methods that are frequently executed, or *hot*, these methods may contain a large percentage of code that is itself *cold*. In this case, effort is being wasted producing native code for instruction sequences within the method that may never be executed. The ART HLLVM discussed in Chapters 5 and 6 relies on a

[3] Use of the term method here can also apply to functions.

method-based approach to compilation.

2.3.2 Trace-based JIT Compilation

Trace-based compilation is designed to only generate code for instruction sequences that have been observed to be frequently executed – typically loops – meaning they are often smaller in size than method-based compilation units. This tracing also allows code and data to be distinguished in binaries executed by EVMs, as the code is initially executed in an interpreter. Tracing relies on the identification of frequently executed *trace heads*, commonly the target of backward branches or function entry-points. Once a trace head has been executed frequently enough, the interpreter switches into a tracing mode, recording every instruction it subsequently executes. This tracing will proceed until the original trace head is reached, a particular instruction is reached that requires tracing to stop, or a maximum number of instructions have been recorded. All instructions recorded after the trace head are referred to as the *trace tail*. This recorded trace of source instructions is then translated to native code, and a mapping between the trace head location and the native code is produced. Whenever the interpreter reaches the trace head again, it can jump to the associated native code, in order to execute the instructions that form the trace.

A limitation with trace-based compilation is that it typically records a single-entry-multiple-exit linear trace of instructions. This may be a problem if, for example, a loop contains basic blocks, one of which is executed 1% of the time, and the other is executed 99% of the time. If the tracing mode happens to trace through the loop when the 1% basic block is touched, the resulting trace will not contain the 99% statement - see the unfortunately timed trace in Figure 2.2. Eventually the early exit from the native code trace to the 99% statement will trigger a new trace, which will contain the 99% statement. Now overhead exists in having to transition between two traces to execute the loop in the common case, and more code has been generated than is necessary.

2.3.3 Region-based JIT Compilation

In region-based compilation, some of the limitations of trace-based compilation are addressed while still aiming to produce only hot code, as opposed to method-based compilation. As native code is executed, frequently executed instructions are recorded immediately without entering an explicit tracing mode, and a control flow graph (CFG)

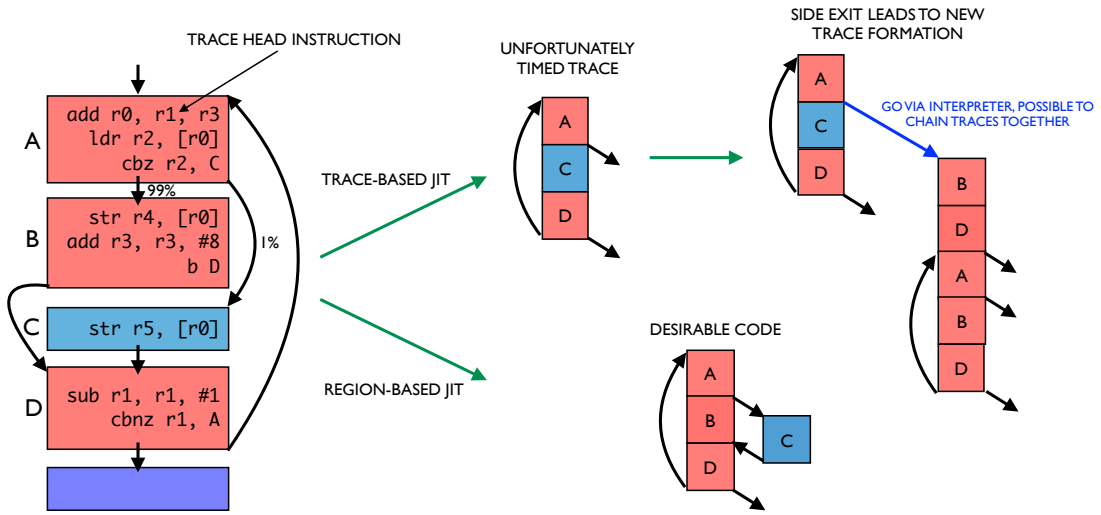


Figure 2.2

Comparison of the results of trace- and region-based JIT compilation from an example piece of ARM assembly. Basic blocks are red to denote frequent execution. An unfortunate tracing of this code sequence causes a suboptimal trace of native code to be generated, necessitating the generation of more code to capture the hot basic block that is missing.

of a "region" is constructed. After a given interval, this CFG can then be translated into native code, and a mapping from every possible entry point into the region to the native code is provided. This avoids the code explosion [BCW+10] of trace-based compilation by constructing CFGs. This also produces less native code on average than the method-based approach. Chapter 4 of this thesis discusses an EVM that takes a region-based approach to compilation.

2.4 The ARCSIM Dynamic Binary Translator

Dynamic Binary Translation (DBT) is a term often used to refer to the process of JIT compilation in EVMs. As mentioned in Section 2.3.1, this requires the use of trace-based or region-based JIT compilation, as method-based is not a possible approach. The DBT used in this thesis, ARCSIM, is an instruction set simulator (ISS) designed to execute binaries compiled for the ARCompact ISA on x86 systems. ARCSIM initially executes binaries in an interpreter, with a region-based system for recording code regions for eventual JIT compilation continually recording how the binary is executed. After a set number of machine instructions have been interpreted, touched code regions

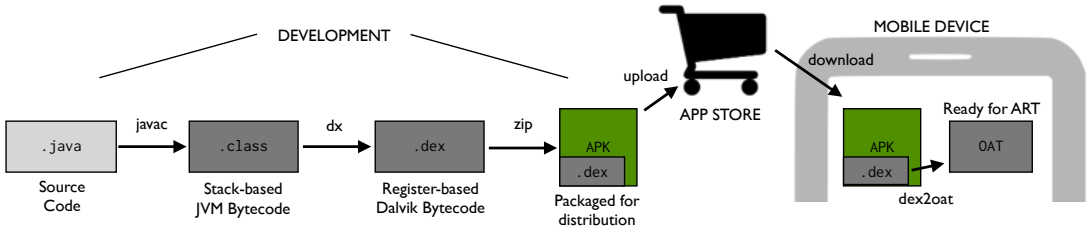


Figure 2.3

Overview of the development and distribution of Android applications.

are dispatched to a work queue for JIT compilation, which is performed concurrently by JIT worker threads. Pointers to translated regions are stored in a lookup table, which is referenced by the interpreter as it fetches instructions, passing control of execution to native code regions when they are found in the table. The decoupled JIT compilation system [BEvKK+11] of ARCSIM is powered by the LLVM code generation framework, which provides tools for generating and optimising native code from the internal representation, LLVMIR, that ARCSIM is capable of generating.

2.5 Dalvik Virtual Machines

In the Android mobile operating system, applications are designed to be written in Java. Rather than producing an implementation of the JVM for execution on mobile devices, the developers of Android chose to design a new HLLVM for the execution of Android applications, called Dalvik. This VM itself implemented the Dalvik Virtual Machine specification, a formal specification of which is not publicly available. Android applications are written in Java, compiled to stack-based JVM bytecode using the Java Development Kit, and then further translated into register-based bytecode that Dalvik is capable of executing. This bytecode is then packaged into an Android Application Package (APK) file along with any other application assets for distribution to users. Figure 2.3 provides an overview of this development and distribution process. As Dalvik was originally designed for use on memory-constrained mobile devices, it was necessary to produce a new bytecode format that combined the multiple JVM class files of an application into a single Dalvik Executable (DEX) binary file allowing the method, string, and type lookup tables to be shared by methods from different classes.

Originally designed with a threaded interpreter intended for efficient execution on ARM processors, a JIT compiler capable of producing Thumb-2 code was added in version 2.2 of Android, improving performance. This trace-based JIT compiler was

quite limited, producing very small traces that provided little opportunity for code optimisation.

2.5.1 ART - The Android RunTime

In version 4.4 of Android, the developers released a new experimental HLLVM for use with Android. The Android Runtime, or ART, was a new *implementation* of the Dalvik VM *specification*, intended to replace the original Dalvik implementation in the future. While Dalvik used an interpreter-plus-JIT design, ART proposed the idea of install-time code generation, somewhere in between just-in-time and the original ahead-of-time code generation VMs were designed to avoid. When an APK is obtained for installation on an Android device, the DEX bytecode is translated into native code designed for later loading and execution. Each method within the application is translated into native code, and stored alongside the original DEX code in what the developers of Android have called an OAT file. In the event a method cannot be translated (for instance, if it exceeds a particular size), then an interpreter is still available to execute DEX bytecode.

The native code generated for ART is designed to conform to the Application Binary Interface (ABI) that ART specifies for a given host ISA. For example, to invoke a method, the arguments to the callee method are assigned to physical registers and the stack, the address of the native code implementing the callee method is loaded (or alternatively the address of the interpreter if no native code exists), and then execution jumps to this address. As this native code is produced ahead-of-time as an application is installed, ART must perform work to patch these addresses into the native code as an application is loaded, analogous to linking in the execution of regular, natively-compiled programs.

2.5.2 DEX Bytecode

The Dalvik Executable (DEX) bytecode format [AOS] relies upon a register-based architecture for the design of the HLLVM. Bytecodes are variable length, constructed from multiple 16-bit *code units*, reaching ten bytes in length at most. The virtual registers are specified to store 32-bit values, with certain opcodes operating on 64-bit values stored in register pairs. Arithmetic operations have explicit input and output types for the bytecode, but constant loading and data movement operations only distinguish between 32-bit, 64-bit, and object reference values. Most bytecodes are only

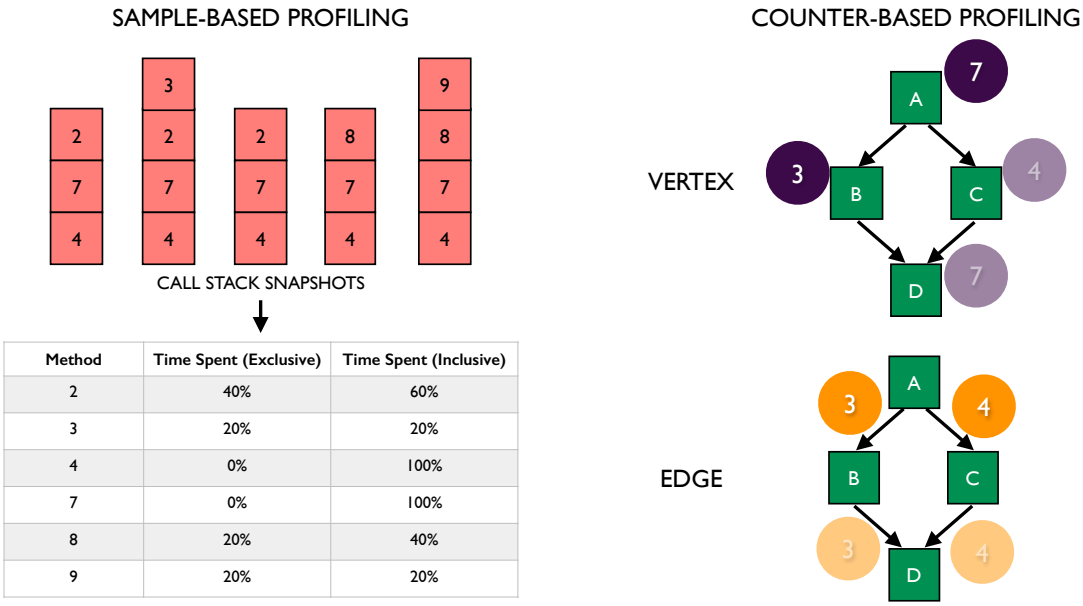
capable of operating on the initial 16 or 256 virtual registers, although up to 65,536 registers can be allocated within a method, with some bytecodes dedicated to moving values between the high and low registers. Bytecodes can reference classes, strings, and methods with an index into a table allocated at the beginning of a DEX file - these tables are shared between all classes declared within the DEX file.

2.6 Profiling

Profiling is the collection of information about the way a program is executed. This can be done in order to provide information about where time is often spent in the program. This information can be used by developers in order to identify bottlenecks and direct where their optimisation efforts should be spent. However, profiling is also useful for the optimisation of code without developer intervention. Code generators can use this information in order to perform a number of code optimisations, such as the reordering of code to reduce the dynamic number of instruction branches taken in a program, or to bring commonly executed code closer together in order to fit more of it into a cache line. In dynamically typed languages this profiling could also be used to obtain type information in order to optimise code for data types it is seen to frequently operate on. The use of this information is referred to as profile directed or guided optimisation. By identifying which functions are commonly executed, this can also determine where efforts to perform optimisations should be focussed, if code generation is being performed at run time. This profiling information can range from obtaining function/method execution counts, down to hot code paths in an individual method's control flow graph. Two common approaches to performing this profiling, sample-based and counter-based, are illustrated in Figure 2.4. Information on prior work into profiling techniques can be found in Section 3.1.5.

2.6.1 Sample-based Profiling

In order to perform sample-based profiling, the original program is executed as normal. Periodically the current function call-stack of the program is observed, requiring execution to be paused in certain VMs. Snapshots are taken of where execution is currently taking place, building up a rough overview of which functions time is commonly spent in. This information can be used to focus optimisations on commonly executed methods in code generators, but could also be obtained at an instruction-level granularity



Method	Time Spent (Exclusive)	Time Spent (Inclusive)
2	40%	60%
3	20%	20%
4	0%	100%
7	0%	100%
8	20%	40%
9	20%	20%

Figure 2.4

Simple examples of basic approaches to code profiling, comparing stack sampling to vertex or edge counting. The faded counters in the counter-based profiling could be removed without information loss, and the information in the vertex case could be recreated from edge profiling.

to identify frequently touched basic blocks within a method. This technique generally trades off accuracy for low profiling overhead. The Android mobile software development kit has a sample-based profiling tool available for use by developers, called TraceView [Gooc].

2.6.2 Counter Profiling

In comparison to sample-based profiling, counter-based profiling maintains individual counters to perform profiling. In vertex-based counter profiling, counters are kept for every basic block in the program that is of interest, and these are incremented whenever the relevant basic block is touched. Alternatively, edge-based counter profiling tracks the branches[4] of a function's control flow graph instead. This approach typically incurs more overhead than sample-based profiling in exchange for its increased accuracy. This profiling may also require the instrumentation of the profiled binary in order to be performed efficiently. Chapter 5 discusses a system for vertex profiling Android

[4]Including the fallthrough "branches" that are transitions between basic blocks without an actual branch.

applications to identify important methods to compile and optimise.

2.7 Process Virtual Machine Testing

A process virtual machine is a complicated piece of technology that requires rigorous testing. As new PVMs are developed, or extended with new methods of execution, bugs are likely to be generated, despite the best of developer intentions. Unit testing of components within the VM is a good basis for testing. The VM is typically comprised of multiple subsystems, such as the execution engine (e.g. interpreter or code generator); the memory system (e.g. memory management unit or garbage collector); the binary loader (e.g. the class loader in a JVM); or the threading system. Extra testing infrastructure can be added to the VM code base that can run these subsystems in isolation, and test that they conform to expected behaviour for given inputs. While a good basis for sanity checking all individual components, this fails to catch bugs that can be introduced through interactions between components.

Another way to test for more bugs within PVMs is to create a suite of programs that can be used to test for expected behaviour. Each test is designed to produce some result, typically console output, that is compared to expected output, to determine if the test ran successfully. The programs can be designed to test the generally expected behaviour of, for example, particular VM opcodes or mathematical operations, but can also be created as bugs are found and fixed, in order to test that future modification of the system does not reintroduce the bug. These tests are commonly called "regression tests". Use of this testing methodology treats the PVM as a black-box system that should produce x output when provided with y input.

Formal methods also exist for testing PVMs, although these are typically applied to the specifications of VMs, rather than their implementations. A major part of preventing bugs in a HLLVM is the verification of input programs that are supplied to the VM. This is done both to enforce access controls (e.g. ensuring private methods within a class are not externally invoked) and to allow code generators to make safe assumptions about the bytecode they are generating native code for. Verification of bytecode is generally simpler for stack-based VMs than register-based ones [SCEG08]. Despite this, the verification of stack-based bytecode for the Java Virtual Machine requires specification in Prolog, hinting at its potential complexity. The VM testing performed in this thesis frequently found bugs to arise from the incorrect verification of bytecode.

2.7.1 Fuzz Testing

While test suites are one form of black-box testing, this thesis deals with another form called fuzz testing, or *fuzzing*. The concept of fuzzing initially referred to the random creation of input to test the capabilities of a program or Application Programming Interface (API), particularly to test its ability to gracefully handle erroneous input. This testing can also be referred to as *black-box testing* - by considering a program or interface as a black box, it can then be bombarded with random data in order to test it. Most programs fall into one of two categories. They may be true positives, or valid programs that are accepted and then correctly executed. Alternatively, they may be true negatives, or programs that are rejected due to being invalid. Fuzzing can be used to identify false positive programs that are accepted, but then execute incorrectly, indicating bugs in the PVM.[5] This incorrect execution can manifest as crashes that are immediately and conspicuously incorrect, or as programs that run to completion while producing a subtly incorrect result, that may be more difficult to identify when testing at scale. This fuzzing can also be used in order to increase total code coverage[6] as well as dynamic path coverage, where the fuzzing attempts to find new unique paths of execution through the code. The strategy for creating new programs with fuzz testing is can be divided into two categories: mutative and generative.

2.7.1.1 Mutative Fuzzing

When performing mutative fuzzing, an initial input test program is taken as a basis for the generation of new programs. Mutations are applied to this initial test program, producing alternatives that can explore the boundaries of the program that is actually being tested. These mutations can be performed through the simple stochastic flipping of bits, or can take place at a higher level. For example, some mutative fuzzers perform mutations on the abstract syntax tree (AST) of a program, by taking sub trees of the AST and replacing them with other sub trees, potentially new ones, or otherwise trees copied from other parts of the AST. There has also been a recent rise in the use of mutative fuzzers that attempt to use information about the way the tested program has executed a mutated input in order to direct further mutations. In Chapter 6, one of these tools, called American Fuzzy Lop (AFL) [AFL], is compared to the fuzzing tool presented in this thesis.

[5] Extending this categorisation to its conclusion, finding false negatives, or programs that are rejected despite being valid would also be interesting, but difficult for the fuzzing system in this thesis to find.

[6] i.e. the percentage of the statements in the tested program that are ever executed

2.7.1.2 Generative Fuzzing

With generative fuzzing, test programs will be randomly generated without using an initial input program to base the generation upon. This can be used to generate programs in either a source code or binary format. One approach to this could be to use a grammar that describes how a program can be generated, for example, taking grammar rules used to parse C code and applying it in reverse. This would mean expanding a function AST node into a random number of block AST nodes, which would be expanded into a random number of statement AST nodes, each of which would expanded into expression AST notes, and so on. This would be done while tracking variable and symbol names along the way, to make sure the program remains syntactically legal.

Chapter 3

Related Work

This chapter presents related work in the fields of code generation for process virtual machines. In particular, the subjects of JIT compilation, dynamic binary translation, code generation on and for mobile devices, the crowd sourcing of information to aid code generation, and fuzz testing and its application to virtual machines are covered.

3.1 JIT Compilation

JIT compilation is a well-studied field that has existed since the 1960s. Aycock [Ayc03] provides a comprehensive history of this technique between 1960-2000. Aycock proposes that the earliest work on JIT compilation was performed by McCarthy [McC60], who mentions that LISP functions executed by his proposed interpreter could also be compiled into machine language. Another early example of JIT compilation was presented by Thompson [Tho68], where regular expressions were dynamically compiled into IBM 7094 code in order to perform fast pattern matching in strings. Other notable instances of JIT were found in implementations of the Smalltalk [DS84] programming language, and particularly its Self dialect [HU94]. Sun's own research into Self would go on to inspire design choices in Java, which resulted in the adoption of *just-in-time compilation* as a common term. Many Java Virtual Machines (JVMs) today such as HotSpot [PVC01] use JIT compilation to accelerate their performance, and research into JIT compilation is performed with JVMs such as Jikes RVM [AAB+00]. This thesis splits recent work in JIT compilation into the categories of tracing (both trace-forming and region-forming) JIT compilation, non-tracing parallel JIT compilation, and tracing parallel JIT compilation.

3.1.1 Tracing and Region-based JIT Compilation

Tracing is a well established technique for dynamic profile guided optimisation of native binaries. Dynamo [BDB00] introduced tracing as a method for runtime optimisation of native program binaries. YETI [ZBS07] uses tracing in a mixed in-line, subroutine threading interpreter to avoid additional virtual method calls and unnecessary branches.

Whaley proposes partial method compilation [Wha01], and uses profile information to detect never or rarely executed parts of a method in order to ignore them during compilation. If such a part gets executed later, execution continues in the interpreter. Suganuma et al. [SYN06] use region-based compilation to overcome the limitations of method-based compilation. They eliminate rarely executed sections of code, but rely on expensive runtime code instrumentation for trace identification.

Gal et al. investigate the use of dynamically profiled trace trees [GPF06, GES+09] as a compilation unit. In this, traces are aggregated into trees as they are executed. Trace trees suffer from the problem of code explosion when many control-flow paths are present in a loop, causing them to grow to very large sizes due to excessive tail duplication. Bebenita et al. [BCW+10] remove the need for interpretation by first compiling the method with instrumentation but no optimisation. When suitable thresholds are reached, execution switches to compiled tracing version of the code; finally, the selected traces are aggressively optimised.

Häubl et al. [HWM14] identify problems with the inability to arbitrarily transition between interpreters and traces of compiled code, presenting a system that allow this to happen efficiently. Optimisations in tracing JIT compilers are often simple, thanks to assumptions that the optimisations need only be correct for a linear piece of code. Loop-invariant code motion is one such optimisation that cannot be performed in this way, so Ardö et al. [ABF12] propose a scheme for making optimisations loop-aware, adding this to the PyPy VM used for Python, an idea inspired by similar work on the LuaJIT VM. Although JVMs are a popular platform for tracing research, there are examples of tracing found in other languages. For example, a trace based JIT compiler was implemented to support the Common Intermediate Language by Microsoft [BBF+10], and this was used to accelerate an implementation of JavaScript in C#.

3.1.2 Non-tracing Parallel JIT Compilation

Krintz et al. [KGL+00] pushed JIT compilation into the background while it continued to interpret bytecode. Only single compiler and execution threads were employed and hot method detection was performed by offline profiling. The ability of background compilation to be more aggressive is exploited by Unnikrishnan et al. [UKL06] to choose the best way to compile a program on an embedded device as its battery level changes. Again, only single execution and compiler threads are used. Kulkarni et al. [KAH07] dynamically increase the priority of its compilation thread to increase compiler throughput. Their technique is useful when the number of application threads is greater than the number of physical cores. The authors then considered the potential to extend their work to multiple compiler threads [KF11], exploring the impact from varying iteration count thresholds and the number of compiler threads. The Java Hotspot Server Compiler [PVC01], allows for the creation of multiple compiler threads via a command line option. Azul VM may use as many as fifty compiler threads for large programs [KAH07]. These techniques all focus on compiling whole methods, not traces.

The Ultra-fast Instruction Set Simulator [QDZ06] pioneered the concept of concurrent JIT compilation workers to speed up DBT, but suffers from a number of flaws. First, rather than taking a trace-based compilation approach entire pages are translated – this is unnecessarily wasteful in a time-critical JIT environment. Second, there are no provisions for a dynamic work scheduling scheme that prioritises compilation of hot traces – this may defer compilation of critical traces and lower overall efficiency. Third, JIT compilers reside in separate processes on remote machines – this significantly increases the communication overhead and limits scalability. This last point is critical, as the results shown by Qin et al. [QDZ06] are based solely on CPU time of the main simulation process rather than the more relevant wall clock time that includes CPU time, I/O time and communication channel delay. A parallel JIT compiler for C is demonstrated by Plezbert et al. [PC97]. Whole translation units are compiled at once by one background compiler thread. Execution is able to jump from interpretation to native whenever the compilation is complete. A form of region based parallel JIT for DBT appears in JPSX, a PlayStation emulator written entirely in Java [San06]. Multiple compilation threads translate R3000 code into Java bytecode with the different threads managing progressively higher optimisation levels. Code is initially interpreted or translated via an in-line threading interpreter or, when suitably hot, translated with

the most optimising compiler thread. A unit of compilation is selected by including all instructions following an entry instruction up to an unconditional branch. JPSX uses Java's own on-demand class loading mechanism to initiate compilation of unvisited code. The HotSpot JVM then provides additional optimisation, being able to make use of the fact that all code blocks are translated to static methods, leading to a system which is so fast that it requires careful synchronisation, not present in the original binaries, to ensure that video frames are not produced faster than the screen can display.

3.1.3 Tracing Parallel JIT Compilation

Ha et al. [HHCM09] attach a 'Compiled State Variable' (CSV) to each trace anchor. The CSV allows a tracing JavaScript interpreter and background compiler to manage transitions from interpreted execution to native without locks. Their approach, however, is only applied to single execution and compilation threads. DynamoRio is a dynamic binary translation system that is extended by Bruening et al. [BKGB06] with the ability to share traces between parallel workers within the system to improve memory use and increase execution throughput. Mojo [CLCG00] is a system that is similar to Dynamo, but extends it to permit proper exception handling. Individual blocks are compiled and as hot traces are identified they are translated. Mojo also offers support for multi-threading; threads maintain a private list of individual blocks but share the set of translated paths. Inoue et al. [IHWN11] implement tracing in the IBM J9/TR JVM. They use a global trace cache and note that the time spent searching the cache is significant. Only one background compilation thread is used. A similar global trace cache is used by Häubl et al. [HM11] for the Hotspot JVM. Wimmer et al. [WCB+09] note that tracing enables simple phase change detection by comparing the ratio of side exits taken to the time spent in the trace itself. Traces can be discarded and recompiled when a phase change is detected. Their work uses a global trace cache and permits only one background compiler thread.

Some approaches [GBCF07, CAR08] have attempted to exploit pipeline parallelism in the JIT compiler. However, pipelining of the JIT compiler has significant drawbacks. First, compiler stages are typically not well balanced and the overall throughput is limited by the slowest pipeline stage – this is often the front-end or IR generation stage. Second, unlike method based compilers, trace-based JIT compilers operate on relatively small translation units in order to reduce the compilation overhead to a bare minimum [GAS+00]. Small translation units and long compilation

pipelines, however, increase the relative synchronisation costs between pipeline stages and, again, limit the achievable compiler throughput. Third, compilation pipelines are static and do not scale with the available task parallelism in inherently independent translation units.

Mehrara et al. take a different tack [MM11]. They note that a considerable time is spent executing trace guards, which would trigger a side exit but are rarely taken. They offload the checking to another, background thread, enabling the main execution thread to speculatively continue along the trace. If the background thread discovers a guard violation, the main thread is halted and execution resumes at the appropriate side exit.

3.1.4 Execution Path Recording Approaches

Recording of execution paths in terms of traces is either triggered by detecting a special construct (e.g. loop header, method entry) [GES+09, HHCM09, WCB+09, BCW+10, IHWN11], or always enabled when interpreting code [BEvKK+11]. Various backbone data structures have been suggested to capture recorded traces such as trace-trees [GBCF07], control flow graphs (CFGs) of traced basic blocks [BEvKK+11], or hybrids between trace-trees and CFGs called trace-regions [BCW+10] or trace-graphs [HM11]. Hsu et al. [HLW+13] propose the use of Early-Exit Guided Region Formation to improve the quality of traces and regions by merging regions into larger regions for recompilation.

3.1.5 General Code Profiling Approaches

While profiling is necessary in order to perform dynamic code generation, it can also be used to direct efforts in static code generation, such as selecting which parts of a program should be compiled or optimised, or as an input for certain code optimisations, such as basic block reordering. While the information gained from profiling will be discussed in detail in Chapter 5, this section takes a look at prior work into performing profiling.

Ball and Larus [BL94] present algorithms for profiling programs, with the aim to find the optimal placement of probes that minimises execution overhead, while still providing a full view of the program's execution, either a report of vertex or edge execution counts. They also investigate how to optimally perform vertex profiling using edge counting. The profiling scheme used in this thesis relies on vertex profiling to

drive selection of methods for install-time compilation, but does this with traditional vertex counting. The overhead of this simpler profiling scheme was not found to be detrimental to the performance of typically executed applications, particularly in comparison to the default profiling system available on Android. If the profiling overhead was found to become significant, the profiling technique could be switched to that of Ball and Larus' proposed method, in order to improve performance.

Path profiling is another form of profiling where the counts of unique acyclic paths of execution within a program are identified. Ball and Larus [BL96] demonstrate that such profiles cannot be correctly determined from edge counts, necessitating extra tracking and overhead, but present an algorithm that places instrumentation intelligently in order to minimise this overhead. Path profiling was also later investigated by Apiwattanapong and Harrold [AH02], showing that in particular cases (e.g., when there are many paths of execution within a function), significant overhead savings can be made, compared to the scheme proposed by Ball and Larus, when only certain paths are of interest.

3.2 Dynamic Binary Translation

QEMU [Bel05] is an extremely popular implementation of DBT, that can translate a wide variety of architectures to x86, and can perform simulation of multiple CPUs. Developers wishing to enable support for a new architecture in QEMU are required to write code that translates their instruction set to the internal TCG ops format, which QEMU can then translate to the instruction set of the host architecture, and this is what enables QEMU to support as many architectures as it does. Optimisations are also performed on this intermediate representation to improve simulation performance. QEMU uses a single thread to simulate multiple CPUs, and consequently has a single code cache used for multiple threads.

COREMU [WLC+11] extends QEMU to perform multicore simulation through the use of multiple QEMU instances in separate threads, where each has its own code cache, meaning each simulated core will do its own independent translation. HQEMU [HHY+12] also extends QEMU with parallelism through the addition of a parallel LLVM-powered JIT compilation worker thread. All simulated threads still execute within a single thread, and continue to perform compilation on a on-demand per-basic block basis. However, these blocks are profiled, and any hot blocks discovered are sent to a queue that feeds into the JIT compilation worker thread. This thread will merge

multiple hot blocks into a single trace, which can then be used in place of QEMU's standard generated blocks, improving performance.

The Mimic simulator [May87] simulates IBM System/370 instructions on the IBM RT PC and translates groups of target basic blocks into host instructions. Shade [CK94] and later on Embra [WR96] cache translations to accelerate execution, however both pause execution during the translation process. Daisy [EA97] translates PowerPC code onto a VLIW machine. The system uses page faults to trap necessary translation tasks. The Crusoe processor from Transmeta [Kla00] also translates to a VLIW machine in software, but from source x86 code. Nvidia offer a similar system called Denver, that executes the ARMv8 ISA on its underlying VLIW processor through a software layer that performs DBT. This naturally requires different design decisions to support this DBT on a much lower power budget in comparison to Transmeta's system.

Just-In-Time Cache Compiled Simulation (JIT-CCS) [NBS+02] executes and caches pre-compiled instruction-operation functions for each function that is fetched. The Instruction Set Compiled Simulation (IS-CS) simulator [RMD03] precompiles the source binary but notices changes to the instructions and invokes a runtime recompilation as needed. The system proposed by Bradner et al. [BFKR09] merges hot basic blocks from MIPS source binaries into regions and JIT compiles them using LLVM.

More recent work on DBT includes HERMES [ZGC+15], a system that performs post-optimisation on its compiled code to achieve superior performance to equivalent systems such as QEMU. Lockhart et al. [LIB15] present Pydgin, that uses architecture description languages written in Python to generate high-performance DBT instruction set simulators. The DBT instruction set simulator used in this thesis, ARCSIM, has been improved by Spink et al. [SWFT14], where a complete approach to region-based JIT compilation for DBT is presented. They highlight a number of aggressive code optimisations implemented in LLVM that result in high performance in comparison to QEMU. Finally, Almer et al. [ABVK+11] extended ARCSIM to support the simulation of multi-core environments, a development that inspired some of the work performed in this thesis.

3.3 Crowdsourced Profiling Related Work

Mobile code generation has been a focus of research for several years now. This section provides a brief overview of the relevant techniques in the areas of Dalvik VM and Java ahead-of-time code generation, install-time code generation, and crowd-sourcing for

use with compilation and optimisation.

3.3.1 Dalvik and Java Ahead-of-Time Code Generation

AOT compilation has been studied in the context of the JVM [PTB+97] long before ART introduced it to the Dalvik VM. Whilst conceptually similar, the mobile nature of the computing platforms used by Android – with their tight constraints on energy consumption and generally lower performance levels than desktop or server platforms – motivates research into specialised AOT compilation approaches suitable for mobile platforms. Icing, an ahead-of-time compiler for Android applications, is proposed by Wang et al. [WPC+11]. Icing converts hot methods of an application program from DEX code to C code and uses the GCC compiler to translate the C code to the corresponding native code. With the Java Native Interface (JNI) library, the translated native code can be called by the Dalvik VM. The execution flow of Icing starts with identifying hot methods of a DEX bytecode application. This is done with the help of a static profiling model. In order to find hot methods, first, the execution frequency for each method is calculated. Then, the high frequency methods (that also pass other constraints, such as being at least a minimum size) are selected as hot methods and presented to the ahead-of-time compiler, whereas the remaining methods will be compiled as usual by the Dalvik JIT compiler. Icing modifies the original DEX bytecode by replacing the original hot method's bytecode with a native header after compilation. The modified DEX code and the native code generated by the ahead-of-time compiler are then packaged into a new application. Whilst conceptually appealing, this approach incurs substantial performance overheads due to the use of JNI for linking in native code, which despite aggressive optimisations limit the overall performance of Icing.

The effect of ahead-of-time machine-independent optimisations on the performance of mobile code applications is examined by Amme et al. [AvRAF09]. This work is not based on Dalvik, but uses an SSA-based SafeTsa mobile code format. The findings include that the effects of performing classical machine-independent optimisations ahead-of-time and on the producer side are – in fact – quite machine-dependent, but applying such optimisations in a mobile code system can nonetheless be beneficial.

Lim et al. [LPKL12] present a selective AOT compiler for Android. Despite extensive use of profiling, its optimisations applied to selected hot methods only result in an overall performance improvement of about 5%.

A more specialised use for hybrid AOT and JIT compilation has been demonstrated

for Java-based TV set-top boxes [JMO14]. An efficient native function interface for Java, effectively replacing the existing, but inefficient Java Native Interface (JNI) is presented by Grimmer et al. [GRS+13] Whilst this has been shown to improve performance for a Java platform, it remains unclear what adaptations would be necessary to facilitate native code interoperability on the Android/Dalvik platform.

3.3.2 Install-time code generation

As JIT compilers convert bytecode to native code when individual methods are called, this necessarily involves a performance hit at run time. Install-time code generation avoids this penalty incurred during each execution of an application, through ahead-of-time compilation at install-time. For example, Microsoft's CLR, a part of the .Net framework, supports install-time code generation in their native image generator (Ngen.exe) to convert MSIL/CIL assemblies to native code. It also persists the generated code in the native image cache as a file on disk. Microsoft's Windows Phone 8 uses a slightly different install-time code generation approach. When an application is built in Visual Studio, code is not compiled into a native image, but into a machine-independent Common Intermediate Language (CIL) binary file. The software developer then submits this CIL file to the app store. At that time, the binary file is converted from CIL to optimised Machine Dependent Intermediate Language, or MDIL. Finally, when the user downloads an application to a device, the MDIL file is linked to produce a native image. These steps are repeated in the development environment whenever an application is (re)deployed to a Windows Phone 8 device. Install-time code generation was also part of the technology request for an Architecture Neutral Distribution Format [Mac93] by the Open Software Foundation. This technology proposed a compilation system where after a program's source code was compiled to an intermediate representation by a "producer" for distribution, it would be compiled on the end user's machine by an "installer" that targets that particular machine's architecture, for subsequent execution.

3.3.3 Crowd-sourcing Information for Compilation

The idea of using information gained through crowd-sourcing to improve the results of compilation is still largely unexplored. Auler et al. [ABdH+14] apply the idea of crowd-sourcing to discover which of their proposed optimisations should be applied when generating JavaScript for their TouchDevelop web-based development environ-

ment. Clients report their achieved performance for a number of test benchmarks with different optimisation flags, along with the platform they are using. The system then uses this information to decide which optimisations to apply when new applications are being developed, based on the developer's computing platform. Otherwise, the only other application of crowd-sourcing, when applied to the domain of automatic code generation, known about is in the MILEPOST project [FMT+08], which also focusses on its application to optimisation – using data from many users to discover which optimisation flags are best to use for a given application through the use of machine learning. A broad and comprehensive survey of all applications of crowd-sourcing to the field of software engineering up to 2014 is presented by Mao et al. [MCHJ15], and includes a discussion of the work by Auler et al. [ABdH+14]. Most of the surveyed uses of crowd-sourcing apply to situations where participants are actively asked to produce some work that is to be crowd-sourced, but this thesis' investigation to produce compilation profiles from many users automatically also falls under the umbrella of crowd-sourcing.

3.4 Fuzz Testing Related Work

The use of fuzzing as a testing strategy started with Purdom [Pur72], where a system that generates random sentences to test parsers was presented. Fuzzing was then used to test UNIX utilities for vulnerabilities by Miller et al. [MFS90]. There have been a number of recent approaches to black-box fuzz testing for file formats, such as Peach [Pea] and AFL [AFL], as well as approaches to fuzzing of x86 instructions [LMMP07]. This section focusses on the previous work into applying fuzzing to the testing of compilers and runtimes, considering other mutative approaches and the use of differential testing.

3.5 Differential testing

Yang et al. [YCER11] present Csmith, a tool that can generate millions of C99 compliant programs, and find over 300 bugs in open-source and commercial C compilers. They use differential testing to find these bugs, checking for a consensus in output between these different compilers, which inspired the use of multiple "backends" in the ART runtime in this thesis. However, their main concern is with avoiding undefined behaviours of C programs, and as such their system does not actually use all features of

any C standard. Additionally, this system is only designed to test the static compilation of C programs, instead of the combination of verifier and code generator used within a VM as performed in this thesis.

Vu et al. [LAS14] present Orion, a tool that uses Equivalence Modulo Inputs (EMI) to reveal more miscompilations than Csmith, which mainly highlights compiler crash defects. EMI is a form of mutative fuzzing, using code coverage information about programs to prune test programs into smaller programs with equivalent semantics. The set of equivalent programs are then executed with the same compiler to ensure they all produce the same result, with differences indicating compiler bugs. Using coverage information to direct mutations would be an interesting direction to take this thesis' work into in the future. Again, a concern of Orion is avoiding generating programs with undefined behaviours, a constraint that does not apply for fuzzing DEX bytecode as this thesis does.

3.6 Fuzz testing within VMs

The published fuzzing of a Java virtual machine has been attempted before, in the tool jFuzz by Jayaraman et al. [JHGK09], built upon an explicit-state Java model checker and a system that fuzzes Java source code to find new program paths. However, we have not found any published literature concerning the application of fuzz testing at the bytecode-level for JVMs or the Dalvik/ART VMs. One advantage of fuzzing bytecode rather than source code is that features of the bytecode that the source language doesn't use can be tested. For example, while the *invoke_dynamic* JVM instruction isn't used by Java, but it is used by other source languages that compile to JVM bytecode. Some preliminary research was performed into fuzzing the ART VM by Sabanal at the Hack in the Box Amsterdam 2014 conference [HIT], but was not formally published.

Papadakis et al. [PM10] use jFuzz to evaluate their technique for automatically generating mutation-based test cases. This approach improves over previous techniques allowing a greater range of programs to be generated. Robinson et al. [REP+11] fuzz Java programs using Randoop, extending this system to produce a tool where regression tests can be automatically generated.

Research has been made into other virtual machines that take source code rather than bytecode as their primary input. In particular, these tools have applied mutative and generative fuzzing to the source code using grammars. Holler et al. [HHZ12] present LangFuzz, a language-agnostic system that has been tested against the JavaScript

and PHP runtimes. Dewey et al. [DRH14] introduce the use of constraint logic programming into fuzzing, building upon the generation goals of stochastic context-free grammars that systems like LangFuzz use for program generation. They apply this technique to the JavaScript virtual machine. A contribution in this thesis, domain-aware binary fuzzing, is related to the use of stochastic context-free grammars, only applied to already compiled binaries instead of source code.

Wen et al. [WZLY13] apply fuzz testing to the ActionScript virtual machine, and is probably most similar to the work presented in this thesis, since ActionScript VMs accept compiled bytecode as input rather than source code. The authors' approach is to produce nearly valid ActionScript source code, parse this source and perform runtime class mutations to produce valid ActionScript programs, which then test the ActionScript VM. This approach is ultimately a generative one, and the authors compare its achieved code coverage against an existing test suite for ActionScript, whereas a test suite is used as the set of program seeds to perform the fuzzing in this thesis.

Chapter 4

Sharing Code Regions in a Multithreaded DBT System

Dynamic Binary Translation (DBT) is an efficient technique to achieve high performance instruction set simulation or emulation of a guest, or source, architecture on another host, or target, architecture with a different instruction set. Such simulators might be aimed at only functional simulation as in QEMU [Bel05], or cycle accurate simulation, or both [BFKR09].

In DBT, a thread is interpreted and frequently executed (hot) areas of code are identified for JIT compilation and, therefore, native execution.[1] When a single core is being simulated, storing the natively translated code sections is straightforward. In the multi-core case, however, a number of issues arise about how individual simulator threads will cooperate in the process of code generation. If and how they will share translated code sections, who will be responsible for performing the translations, and how the costs of any required synchronisations will be mitigated, are crucial questions that must be resolved to achieve high performance. The primary choices of compilation unit in DBT compilation are *trace-based* and *region-based* (for more information, see Section 2.3), and while some work has investigated parallelisation schemes for trace-based systems, the same cannot be said for the potentially more powerful region-based ones. This chapter examines the optimal strategies for just such a parallelisation of the region-based form of DBT simulators, utilising information sharing to improve simulation performance.

The *trace-based* variants of DBT gather linear traces of native instructions from

[1] Instruction sequences may alternatively be JIT compiled as they are initially touched, bypassing interpretation [Bel05].

a deemed hot entry point, called the trace head. These linear traces are sometimes collected into trees, with branches compiled separately and patched together. Due to the linearisation of control flow in which basic blocks may be replicated in many different traces, trace-based systems may suffer from code explosion [BCW+10]. The great advantage of trace-based methods is that the linearisation of control flow makes compilation and data-flow analysis particularly simple, allowing the compilers to be easy to write, and to have extremely small footprints. However, the benefits of applying typically loop-oriented optimisations to larger control flow graphs (CFGs) are lost.

On the other hand, *region-based* systems operate on hot, arbitrarily shaped subgraphs of the CFG. Their compilers are necessarily more complicated and difficult to write, but may yield additional optimisation opportunities that are unavailable to their trace-based cousins. The advent of retargetable, reusable JIT systems such as LLVM, has meant that compiler complexity is no longer detrimental to the development time of region-based systems. Indeed, even trace-based systems are being retrofitted with such off-the-shelf compilers [BFKR09].

In trace-based systems, a number of approaches to sharing translations and efficient synchronisation have been tried. A naïve approach to trace-based JIT compilation would operate on a single, global trace data structure, which would need to be protected from concurrent updates using an expensive locking mechanism. This approach has been taken by, for example, Inoue et al. [IHWN11] An alternative approach that avoids this problem is to maintain thread-private traces [GPF06, HM11], i.e. for each application thread a separate trace data structure is maintained. While this approach does not suffer from excessive synchronisation cost, it introduces a new problem. Multiple threads, especially in data parallel applications, may produce nearly identical traces that differ only in thread-specific constants and accesses to thread-private variables. While there is some opportunity for optimisation with thread-private code generation that may not be possible if the code were generated in a more generic fashion, clearly, this approach increases pressure on the JIT subsystem, and this will lead to degraded performance.

In the case of region-based DBT, no work has investigated different methods to parallelise the simulation environment. The work closest to that presented in this chapter is not region-based but trace-based. Bruening et al. [BKGB06] considered the benefits of sharing linear traces over keeping private traces. In their work, the compilation of traces is performed on the execution thread, rather than asynchronously in the background as in this work, resulting in progress on that thread stalling during the compi-

lation. In addition, because their compilation units are traces, rather than regions, only one linear trace can exist per trace head. As a consequence, if, for a given trace head, the JIT compiled trace tail happens to be non-representative of the general behaviour of the current thread, or indeed the many other threads in the system, that poorly chosen trace will be forever attached to the trace head for all threads. In contrast, the region-based approach builds up regions of hot basic blocks, potentially with multiple entry points per region. Each thread individually discovers which regions are important to it and shares identical regions with other threads. This system permits threads to have overlapping regions between threads. In this way, each thread is not penalised by the occasional poor choices of other threads but benefits from sharing amongst threads which exhibit the same behaviour.

This chapter presents a novel and scalable scheme for region-based JIT compilation of multi-threaded applications. The key idea is to extend the thread-private region compilation model with the capability for *sharing* of regions between threads. Central to this idea is the generation of *thread-agnostic* regions, i.e. regions that do not contain thread-specific constants or data accesses, but are generic enough to be executed in the contexts of different threads. With these two features in place, region sharing is demonstrated to be both effective and scalable.

The proposed technique is implemented and evaluated using the ARCSIM dynamic binary translator targeting a multi-core platform comprising up to 128 embedded cores, where each core is capable of supporting a single application thread using the POSIX threading API. This thread-agnostic region-based code generation is shown to not incur any performance penalty, although it may prohibit some minor opportunities for code optimisation. Region sharing, however, is demonstrated to provide great potential for performance improvement. An average 76% of all JIT-compiled regions can be shared in the Splash-2 benchmarks when run with 128 threads. This results in an overall performance improvement of 1.4x averaged across all benchmarks and over a state-of-the-art thread-private region-based compilation approach without region sharing.

4.1 Motivating Example

Consider the multi-threaded benchmark `water-spatial` from the Splash-2 benchmark suite, when executed with 128 threads in the ARCSIM dynamic binary translator. Of all the regions handled by the asynchronous JIT subsystem, 79% were actually similar to previously requested regions (see ① in Figure 4.1). This shows that there is a

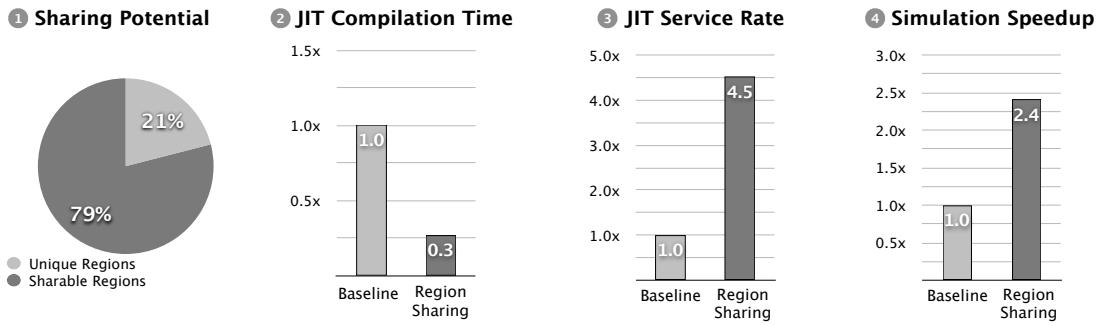


Figure 4.1

When executing with 128 threads, ① shows the scope of region-sharing for the `water-spatial` benchmark. ② shows the resulting decrease in time spent performing JIT compilation, while ③ shows the increase in the rate at which executing threads receive translations. These effects combine to provide the achievable speedup seen in ④ when using the region sharing optimisation proposed in this chapter.

large potential saving to be made when regions are shared between threads. Use of the region-sharing approach leads to a speedup of 2.4x for this benchmark (see ④ in Figure 4.1).

This is made possible through the development of regions that can be executed by any thread – these are said to be *thread-agnostic* (see ② in Figure 4.4). These regions then allow a scheme to be developed where commonality between regions is identified as they are dispatched for JIT compilation, allowing multiple threads to use the result of the region translation. This can increase the service rate of the JIT subsystem - the rate at which threads receive JIT compilation results (see ③ in Figure 4.1.) This will reduce the amount of time threads have to continue executing in interpreted mode until a region is JIT compiled. It enables earlier execution of native code and reduces the pressure on the JIT compilation subsystem, additionally reducing the amount of time spent performing JIT compilation (see ② in Figure 4.1.)

4.2 Sharing of Regions in Multi-core ARCSIM

This section begins with a description of the ARCSIM multi-core DBT system, focusing on its parallel task farm based JIT subsystem. The design of *thread-specific* and *thread-agnostic* regions is then contrasted, followed by a description of how the DBT

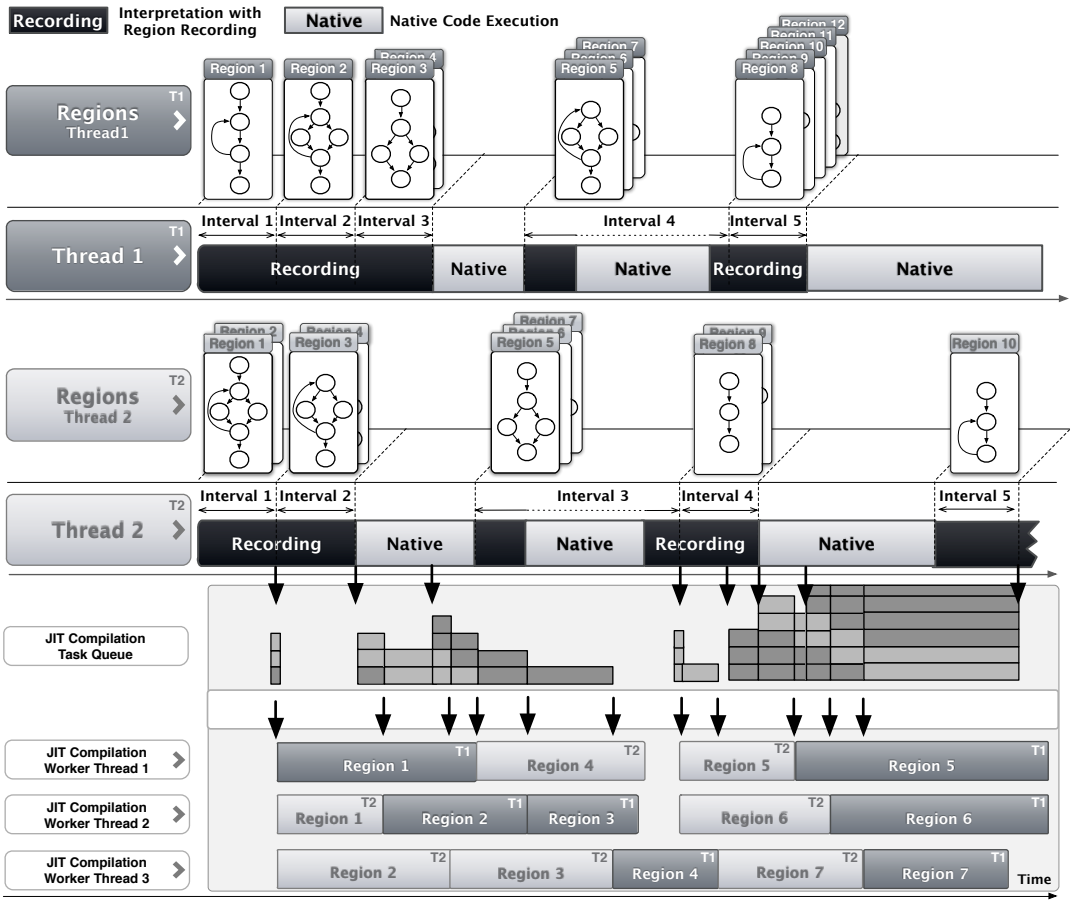


Figure 4.2

A timeline of the region recording process, with dispatch to the JIT compilation task queue that multiple JIT workers can service. Over time, this queue starts to fill up as it is overwhelmed with regions recorded from multiple threads.

has been extended to enable sharing of regions between threads of execution.

4.2.1 Multi-core DBT Design

ARCSIM, the DBT modified in this chapter (and introduced in Section 2.4), allows the execution of binaries built for the ARCompact instruction set on a host machine that implements a different host instruction set architecture (ISA). ARCSIM combines interpreted execution and native execution following JIT compilation of the most frequently executed regions of code. JIT compilation is performed asynchronously to the interpretation of the program, as a parallel task farm based JIT subsystem has been added to the DBT.

Even in a single-threaded context, light-weight region-forming is always enabled

when code is interpreted, recording the control flow exhibited by the source program at run-time. For every target memory page that is executed, a map of basic blocks executed within that page is maintained, with each basic block potentially pointing to a piece of generated native code that simulates execution beginning from that basic block.

The interpreter loop of execution will initially look up the virtual memory page for the current simulated PC, finding its basic block map, or creating it if the page is accessed for the first time. It will then use the PC to look within the basic block map, and will jump to native execution if it finds a native code pointer for the basic block, by performing a normal function call to that address. If it fails to find a basic block in the map for the current PC, it will create a new entry in the map, indicating that it has been touched in an associated counter field - it will also increment this touched counter for any basic block that it does find in the map, thereby recording relative execution frequencies for basic blocks. If it fails to find the native code pointer for any reason (the page had not been touched yet, the basic block had not been touched yet, or no translation is yet available), then the target binary's instructions will be fetched and interpreted up to the next control flow point, before another lookup is made for native code.

To reduce the number of full native code lookups that must be made, a cache mapping target PC addresses to native code pointers is also maintained. This cache is updated whenever a lookup is made through the basic block map for the current page, and the cache is always checked before a full lookup is made, in order to speed up execution.

Interpretation is partitioned into intervals (see Figure 4.2), the length of which is determined by a user-defined number of interpreted instructions. At the end of each interval, frequently executed regions are dispatched to a region translation *priority queue*, before the program continues to be interpreted. More specifically, a region is formed of all the basic blocks within a target memory page that were touched during that interval.

A detailed description of the priority function used to determine the order in which regions are translated can be found in the paper that initially described this system [BEvKK+11]. JIT workers operating in parallel threads can then claim a region from the queue, JIT compile it, and update the region translation state, thereby indicating the availability of native code for that recorded region to the interpreter loop. This is done by updating the basic block maps for target memory pages. For every basic block that formed part of the recorded region, its native code pointer is updated to point to this

new translation. Subsequent look ups of the basic block maps done by the interpreter will therefore find these pointers, and transition to native code execution.

ARCSIM has been extended by others with the ability to emulate multiple processors executing concurrently in a shared-memory environment [ABVK+11]. Each processor runs in a separate thread on the host machine, and records its own internal representation of the target binary, dispatching hot regions to a global priority queue. The JIT subsystem operates as before, except each work unit dispatched to the queue is tagged with the identity of the core that is requesting it. Upon translating the region to native code, the JIT worker updates the region translation state of the core that dispatched the region, with a pointer to the native code.

Section 3.1.4 discusses prior approaches to recording paths of execution for code generation in DBT. To expand on this, QEMU [Bel05] performs block-sized translation of code, translating code as it encounters instructions for the first time, from the current instruction until the next control-flow instruction. As QEMU is also single threaded, it generates blocks for all cores into a single code cache, allowing all cores to immediately access the results of translation. As of 2015, the developers of QEMU are moving towards a multithreaded QEMU, and are concerned [2]with the thread-safety of the global code cache.

HQEMU [HHY+12] extends QEMU with the addition of a parallel LLVM-powered JIT compilation worker thread. However, simulation of all threads still happens within one thread, meaning that there is no complication arising from synchronisation between these threads when recording code regions for translation. This is also made easier by the fact that translation is still performed purely on an on-demand, basic block basis. The extra compilation thread performs all complicated trace forming as it is notified about any hot blocks, and the resulting code generated by this block merging is immediately available for all simulation threads to use.

This chapter's approach takes the scheme shown in ② of Figure 4.3 and extends it to compile *thread-agnostic* regions for data-parallel sections of code *once* and share the compiled region with all threads that profiled the code region (see ③ in Figure 4.3). Consequently, the pressure on the underlying JIT compiler is reduced and translations become available instantaneously to all threads that execute the same region as soon as the first of a number of identical regions has been compiled.

[2]`https://lists.gnu.org/archive/html/qemu-devel/2015-06/msg03458.html`

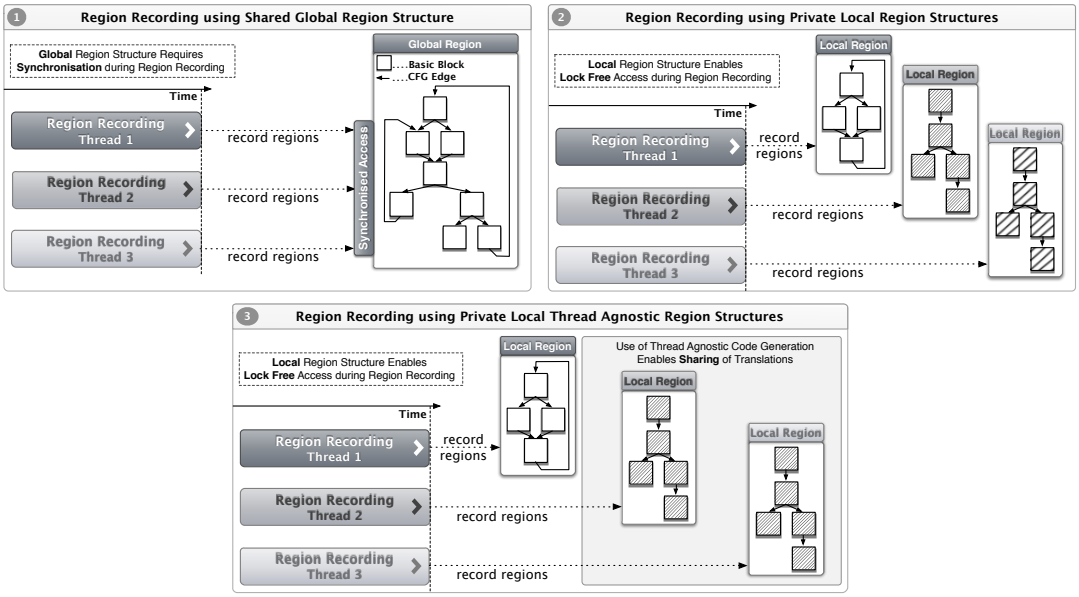


Figure 4.3

Comparison of recording approaches for multithreaded applications: ① recording using a shared global structure [GES+09, HHCM09, WCB+09, BCW+10, IHWN11] requiring synchronisation, ② lock-free recording using private local structures. Finally, ③ shows the lock-free region recording approach using private local thread-agnostic regions to enable sharing of regions.

4.2.2 Thread-Agnostic Regions

In order to enable the sharing of regions between threads, native code must access the thread state structure in a manner that will work for any thread that may execute the region. Consider first, that as this native code is generated as the target binary is simulated at run time, the memory location of each thread state structure is known. Therefore, it would appear obvious to directly reference the structure using the known, constant memory addresses. A region generated in this way is referred to as a *thread-specific* region (see ① in Figure 4.4.) However, this code generation scheme will prohibit region sharing between threads. Constant references to thread-private data will make it impossible to reuse the translated region for any other thread other than the one it has been generated for.

Instead, a scheme is proposed whereby the generated native code accesses the thread state structure *indirectly* through a base pointer. Each core's interpreter must then provide the base pointer to its own thread state structure when switching over to executing native code. These regions are referred to as *thread-agnostic* (see ② in

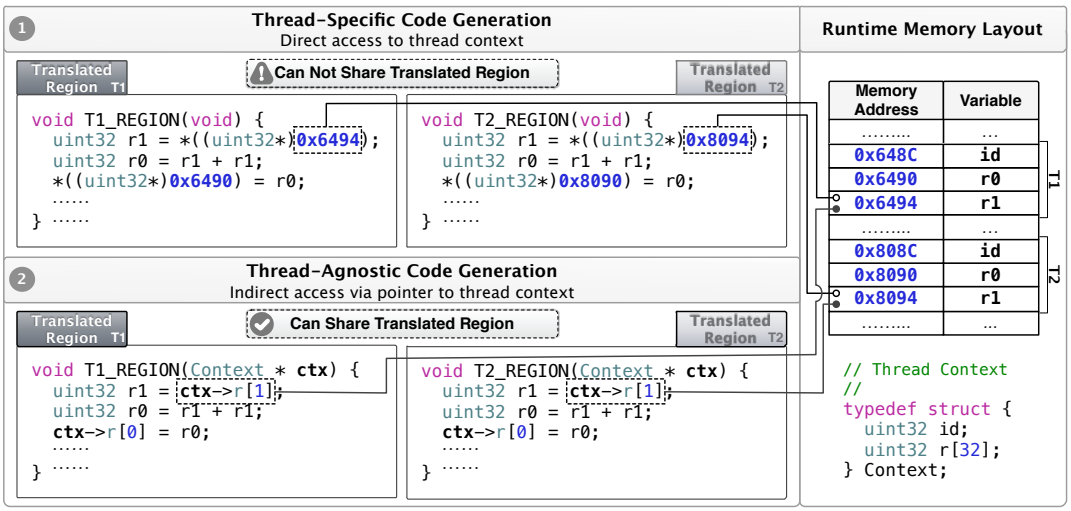


Figure 4.4

The key difference between ① *thread-specific* and ② *thread-agnostic* code generation is highlighted, namely code generation independent of thread context, *enabling* the use of region-sharing to improve performance.

Figure 4.4). In this case, both threads can use the same native code, as thread-specific constants and thread-private variables are accessed via base pointer indirections.

Due to the additional memory accesses and offset calculations, one would be quick to assume that a region generated in a thread-agnostic fashion will perform more poorly than a thread-specific region, when compared independently of the effects of region sharing. In Section 4.3.5 this claim is tested, and this thesis demonstrates that – contrary to intuition – thread-agnostic code generation does not incur any performance penalty on the x86 architecture.

4.2.3 Inter-thread Region Sharing

With a method for generating thread-agnostic regions, and as regions are dispatched to the priority queue of the asynchronous JIT subsystem, it would be beneficial to identify which compilation requests cover identical code regions. These regions can therefore be shared between threads of execution.

To quickly determine if two regions can be shared, signatures are generated for each region as it is constructed. The signature is the 32-bit result of a hash function applied to the physical addresses of all basic block entry points in the region. Only in the case where the two signatures match, will a more expensive check for equality need to be performed. This is done to establish beyond doubt that the regions indeed

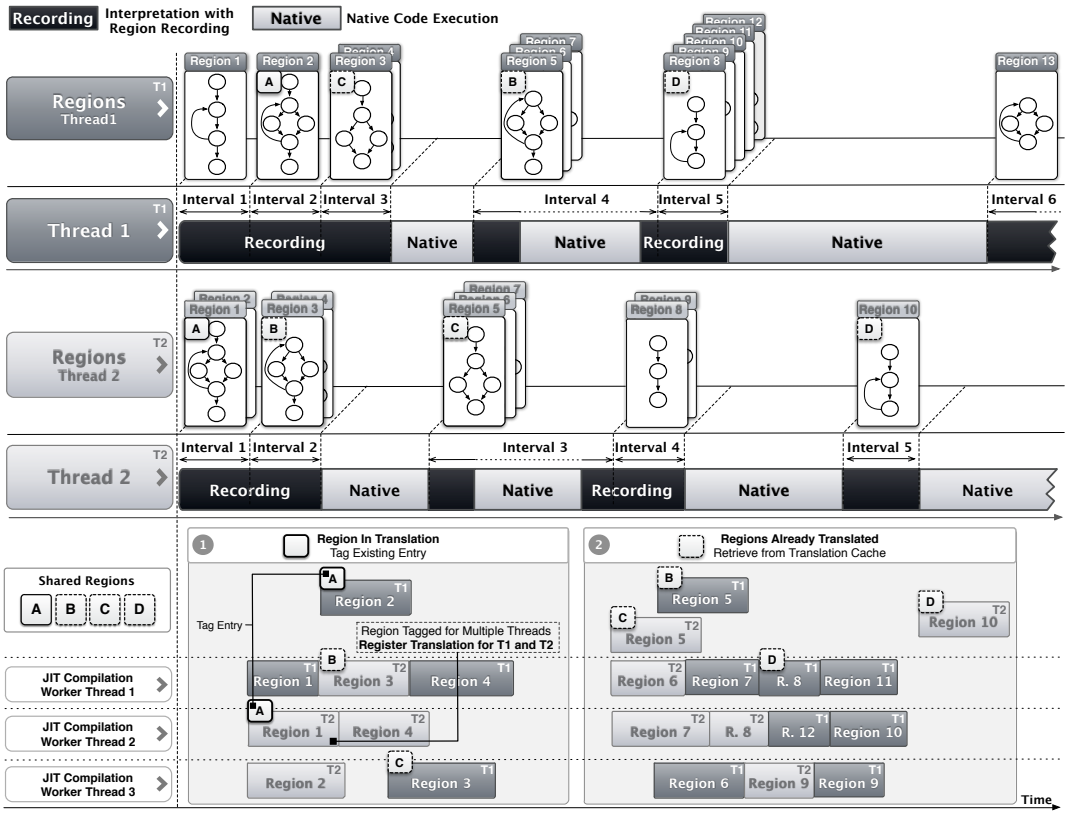


Figure 4.5

Region sharing for a sample multi-threaded application - Frequently executed program regions from two threads `T1` and `T2` are recorded as regions and dispatched for JIT compilation. As soon as a region is compiled, it is cached in a translation cache and its availability is registered with the thread responsible for dispatching the region. ① shows how `T1` records a region (Region 2) that is equal to a previously dispatched region (Region 1) by `T2`. The *A* signature indicates that they are equal. The previously dispatched region is still in translation, hence it is tagged to record the fact that its translation must be registered with `T2` and `T1` once complete. In ②, when a thread records a new region that has already been translated by another thread, its translation is immediately retrieved from the translation cache, enabling immediate availability of native code. For example, `T2`'s Region 5 matches `T1`'s Region 3, as marked by the *C* signature.

cover the same code paths and rule out hash collisions.

A hash table is maintained alongside the priority queue. For a given key signature, this table stores all threads that are interested in the associated region that is currently waiting to be handled by the JIT subsystem. Attempting to add any region that matches

signatures with a region already in the JIT subsystem will result in this region being added to the hash table, instead of the priority queue itself. These regions are bundled in a manner which includes a reference to the requesting thread, so that JIT workers can update the requesting threads with the result of the JIT compilation (see ① Figure 4.5).

The JIT compilation workers continue to fetch regions from the queue. After handling a region, the worker checks the hash table for the threads that are interested in executing this region, allowing them to be updated with the native code generated from the region.

This technique has the dual effect of reducing the period many threads must wait between the dispatch of a region and the receipt of a translation. It also reduces the number of similar regions in the priority queue, thereby reducing the pressure on the underlying JIT subsystem.

4.2.4 Region Translation Caching

The previous section describes how regions can be shared if a thread attempts to dispatch a region *while* the same region is currently in the JIT subsystem, requested by another thread. What if a particular thread reaches a hot region of code much later than other threads? In this case, there are no matching regions currently in the JIT subsystem, so a JIT worker would be required to compile the region again.

One possible solution for this problem would be to register a translated region with all threads once it has been JIT compiled. The drawback of this solution is that translations might be registered for threads that have not yet executed that specific code path, thereby adding complexity to the tracing interpreter loop. This is because the interpreter builds a discovered view of the binary as it executes it, simultaneously allocating storage for pointers to generated native code. The storage location for a given page will not exist yet if a thread has yet to discover it. Furthermore, in the case of task-parallel workloads, most – if not all – translated regions need not be shared and will only be used by one thread.

Instead in ARCSIM, a global software cache for region translations is implemented, and each JIT compiled region is added to this region translation cache. When a region is dispatched for JIT compilation, this region translation cache is first checked to see if the region has already been compiled. If so, all native code generation can be skipped, and the translation can be registered for the thread that requested the region (see ②

Figure 4.5). Region translation caching thereby removes redundant re-compilation from the critical path of execution for many threads, improving overall performance.

4.3 Evaluation

This method for sharing compiled regions in ARCSIM was evaluated using the Splash-2 benchmark suite [WOT+95] to determine the performance benefits resulting from the use of region sharing to improve region-based JIT compilation in a multi-threaded DBT. Splash-2 is a set of 12 parallel benchmarks covering a range of application domains such as linear algebra, complex fluid dynamics and graphics rendering. In cases where both contiguous and non-contiguous versions of a benchmark are provided, the contiguous version was selected. The `radiosity` benchmark was built without parallel preprocessing, resulting in the initial phase of the benchmark executing in a single thread. First, the experimental set-up used during this evaluation will be described, before the rest of this section presents performance improvements. Additionally, throughput measurements and data relating to the potential for region sharing in each benchmark will be provided.

4.3.1 Experimental Set-up

A machine running the operating system Scientific Linux 5.5 with the following specifications was used to perform all experiments: a 4-core Intel Xeon E5430 running at 2.66GHz, with 8GB of memory available.

All experiments were performed under low system load and each experiment was run at least 15 times. In all results, error bars represent the standard error. Prior investigations into the use of the parallel JIT task farm [BEvKK+11] found that assigning three worker threads was sufficient to achieve peak speedup for many benchmarks, but was also the largest amount that improved performance across all tested benchmarks, so this determined the initial choice of workers used in this investigation. Because more tasks will be submitted to the same queue as the number of simulated threads increases, it will also be necessary to increase the number of JIT compilation workers further to handle this increase. This growth should be related to the number of threads performing simulation, so it was decided to increase the number of workers by $\lfloor log_2(n) \rfloor$, if n simulation threads are used. This was also directed by the theory that a logarithmic increase would be preferable over a linear growth rate, inspired by the

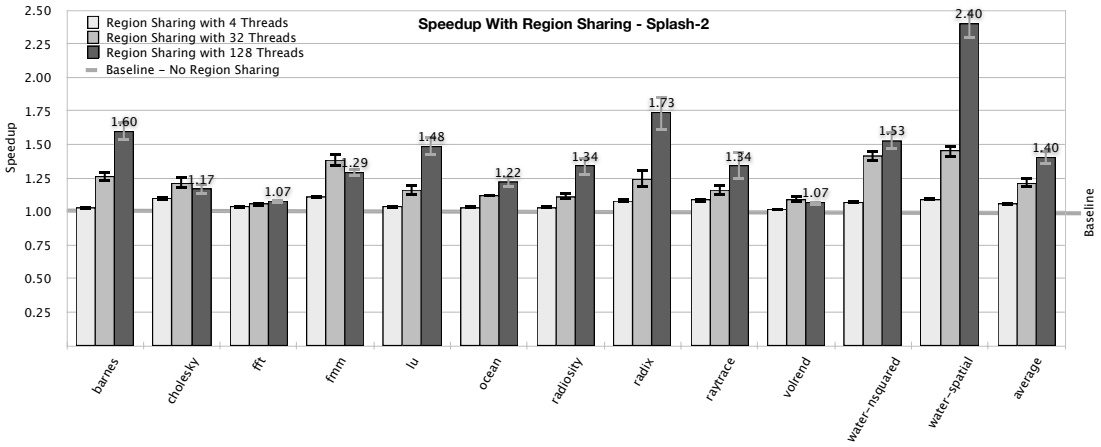


Figure 4.6

Speed-ups achieved through the use of region sharing, over a baseline execution where region sharing is not used. Speed-ups are presented when executing 4, 32, and 128 threads with all benchmarks from the Splash-2 suite.

prior work showing diminishing returns as the number of workers increases, although further work should investigate this relationship in more detail. In the end, the number of JIT compilation workers used in each experiment was fixed at $3 + \lfloor log_2(n) \rfloor$, where n is the number of threads being simulated.

4.3.2 Performance Improvements

The first experiment was to measure the runtime improvements gained through the use of region sharing when generating thread-agnostic regions. Figure 4.6 presents speedups obtained when executing 4, 32 and 128 threads on the 4-core host machine. The baseline here is the same DBT, generating thread-agnostic regions, but without the use of region sharing. As the number of threads increases, so too should the potential for sharing regions between threads, resulting in improved performance.

It can be seen that, in all cases, the use of region sharing improves the performance of the DBT. The average improvement for 4 threads is 1.06x, 1.22x for 32 threads, and for 128 threads, 1.4x. The highest speedup of 2.40x is obtained for `water-spatial` with 128 threads.

On average, the speedup obtained when sharing regions increases as the number of executed threads increases. However, three benchmarks do not follow this trend: `cholesky`, `fmm`, and `volrend`. This was determined to arise from a lower potential for region sharing, resulting from non-homogeneous compute patterns in these bench-

Benchmark	32C	128C	4C-Shared	32C-Shared	128C-Shared
barnes	3.97x	18.20x	0.98x	3.21x	11.69x
cholesky	8.58x	28.03x	0.92x	7.18x	24.46x
fft	5.03x	18.96x	0.97x	4.79x	17.70x
fmm	5.87x	20.02x	0.90x	4.32x	15.62x
lu	13.16x	53.61x	0.97x	11.57x	37.56x
ocean	25.89x	126.57x	0.97x	23.26x	105.80x
radiosity	2.23x	11.59x	0.97x	2.03x	8.98x
radix	28.69x	91.97x	0.93x	23.99x	57.49x
raytrace	7.78x	23.99x	0.93x	6.83x	19.23x
volrend	7.85x	32.73x	0.99x	7.24x	30.83x
water-nsquared	3.47x	16.21x	0.93x	2.48x	10.91x
water-spatial	5.98x	28.33x	0.92x	4.17x	12.06x

Table 4.1

Increase in total execution time when simulating more threads than the host machine provides, with and without the effects of region sharing.

marks. As Section 4.3.4 will show, these three benchmarks have a smaller percentage of sharable regions than the average. Having fewer shared regions available leads to reduced savings in JIT compilation time, when considered in relation to the overheads added by increasing the number of threads.

To provide context for how the simulation of these multithreaded benchmarks is affected by simulating more threads than the host machine machine provides, Table 4.1 lists the increase in execution time to simulate each benchmark, over a baseline of 4-core simulation with no region sharing enabled. The first two columns show an increase for 32- and 128-core simulation – without region sharing – demonstrating that the increase in overhead is often superlinear for these benchmarks. The next three columns show the increase in execution time over a baseline of 4-core simulation without region sharing for all three core configurations when region sharing is utilised. This not only indicates that the use of region sharing can be beneficial when simulating an equal number of target threads as host threads, but also that its application can help reduce the overhead introduced when performing a massively multicore simulation.

These results clearly highlight the benefits of region sharing between application threads. Extending the parallel task-farm JIT subsystem with a scheme to tag sharable regions ensures that multiple threads can be served simultaneously from just a single

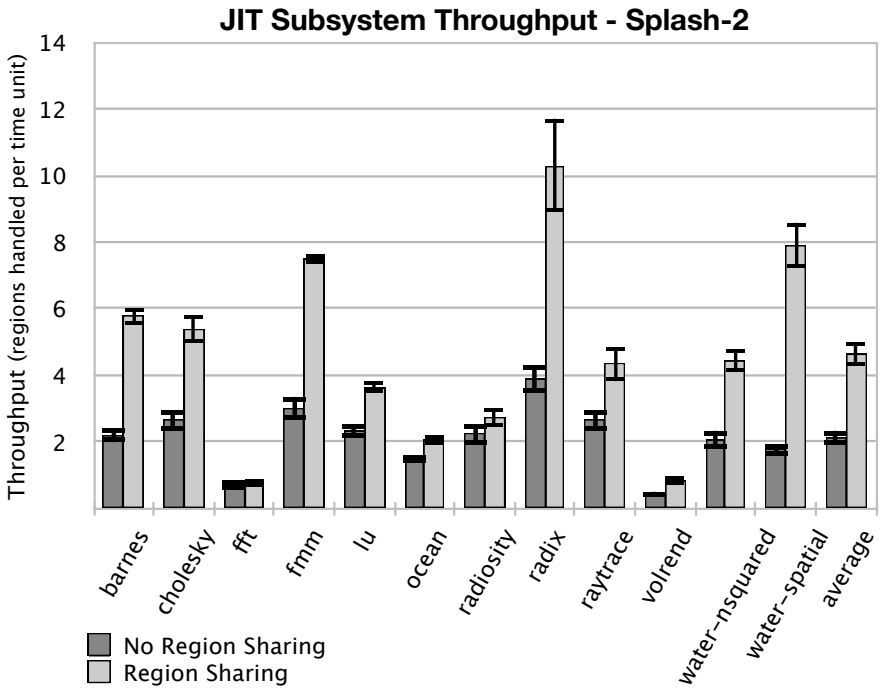


Figure 4.7

Comparison of JIT subsystem service throughput, with and without region sharing enabled, when executing 128 threads.

region translation. Additional caching of recently handled regions supports this sharing concept further. This results in more threads reaching native code execution sooner, without having to wait for a long time in an interpreter phase while their request sits in the translation queue. ARCSIM is capable of reporting the percentage of target instructions executed in either interpreted or native modes, and so a larger percentage of natively executed code was observed when region sharing was enabled.

4.3.3 JIT Subsystem Service Throughput

The JIT subsystem used in the ARCSIM DBT is essentially a decoupled service that ARCSIM can utilise to perform JIT compilation. As such, it is useful to observe the service throughput of the JIT subsystem - how many threads can it provide with native code in any given unit of time? In the region sharing case, this means that if the same region is requested by e.g. three threads, then the worker will have served three threads after performing the translation to native code (rather than just one if no regions are shared). Figure 4.7 shows how the throughput differs with and without the use of region sharing, when executing 128 threads.

These results demonstrate that, for every benchmark, the use of region sharing

increases the throughput of the JIT subsystem. The largest throughput seen is 10.3 regions per time unit when executing `radix` with region sharing, but the greatest actual improvement in throughput is an increase of 4.39x in `water-spatial`. The generally low throughput of `fft` may be due to particularly large regions being generated, or the total number of regions being generated in the benchmark being relatively small in comparison to its total runtime. The important result to note is that region sharing roughly doubles the throughput of the system on average, increasing from 2.1 to 4.6 regions per time unit.

4.3.4 Potential to Share Regions

The speedups presented in Section 4.3.2 arise from the ability to share regions between threads, reducing the time that threads need to spend interpreting until they can execute native code, and reducing pressure on the JIT subsystem. For each benchmark, it would be interesting to know what percentage of the regions that are handled by the JIT subsystem are similar to regions that have already been handled when not sharing regions. This would mean that in a perfect situation, all the time spent compiling these sharable regions could be saved. To measure this, the non-sharing version of ARCSIM was modified to output the signatures of regions that are handled. Figure 4.8 shows the percentage of unique and sharable regions for each benchmark, when executed across 128 cores.

With the exception of `radiosity`, all benchmarks have over 65% of their regions marked as sharable, with an average of 76%. The largest percentage seen here was the 94% of regions generated during the execution of `radix`. The low 47% of `radiosity` can be explained by the aforementioned fact that the benchmark was built without parallel preprocessing, increasing the total percentage of the program that was executed sequentially.

These results show the potential that exists to speed up execution of these multi-threaded programs when using region-based JIT compilation. If 94% of all regions are sharable, this means that up to 94% of all time spent compiling regions could be saved if region translations are shared between threads using thread-agnostic regions, inter-thread region sharing, and region translation caching.

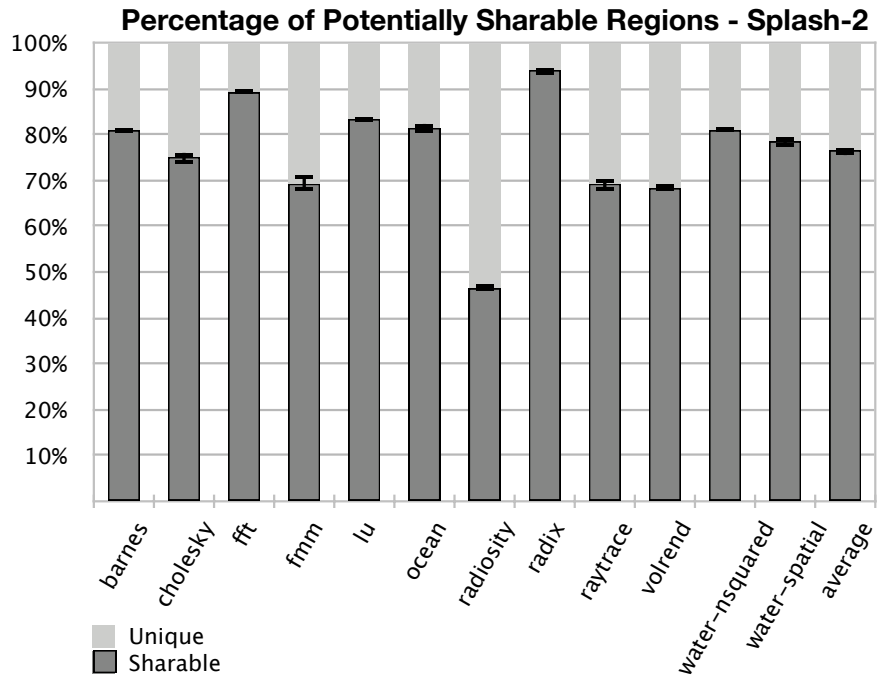


Figure 4.8

Percentage of sharable regions for the Splash-2 benchmark suite when executing 128 threads.

4.3.5 Thread-Specific vs. Thread-Agnostic Regions

Section 4.2.2 discussed the use of thread-agnostic and thread-specific regions, explaining that thread-agnostic regions allow translated regions to be shared between threads. Code generated for thread-agnostic regions requires the use of indirection via a pointer to access the thread's state, versus direct access to addresses known at JIT compilation time. It would be natural to presume that this indirection imposes a penalty on the overall performance of execution, so this section investigates the effect this tracing style has on the runtime of the Splash-2 benchmarks – when executing only one thread – in Figure 4.9.

Surprisingly, these results show that the use of thread-agnostic regions is often faster than using thread-specific regions - a speedup of 1.09x on average. One would expect that having to obtain the thread state pointer and calculate an offset would be more expensive than simply accessing a constant known at runtime. However, this does not take into account the issue of code size. On the `x86` host architecture used throughout this chapter, an instruction that accesses a memory location using register plus offset calculations will not require more than 4 bytes to encode. On the other hand, encoding a 32 or 64-bit immediate constant requires at least 4 or 8 bytes to encode the

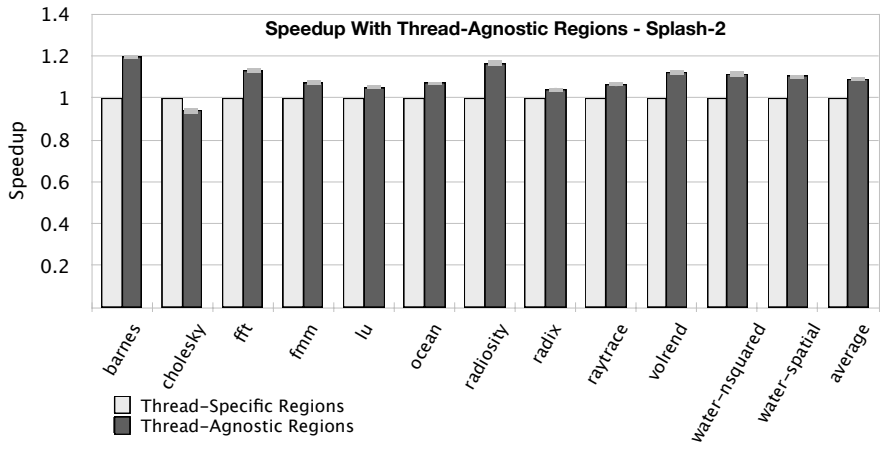


Figure 4.9

Relative performance obtained when using thread-agnostic over thread-specific regions, when executing one thread for the Splash-2 benchmark suite.

constant alone, before the rest of the instruction is considered.

These results demonstrate that the use of thread-specific tracing leads to an increase in overall code size, even if the number of instructions generated may decrease. Having a larger amount of code may lead to slower execution, for instance, if important sections of code can no longer reside completely in the instruction cache. Of course, these results may differ on other architectures. These results show that the use of thread-agnostic tracing actually results in faster execution of threads, even before enabling the sharing of regions between threads.

4.3.6 JIT Compilation Time Saved

One aim of sharing regions is to reduce the amount of time that is spent JIT compiling regions. In the extreme case where 128 threads all request that the same region be JIT compiled, then it should be possible to remove 127 compilations, allowing JIT compilation workers to quickly move onto other regions of code that are enqueued. For each of the Splash-2 benchmarks, this section presents the percentage of time that workers spent compiling regions when able to share them, compared to when region sharing was disabled. Figure 4.10 presents these results, when executing with 128 threads. The baseline here of 100% represents the time spent performing JIT compilation when not sharing regions.

On average, JIT compilation time is reduced by 56% when sharing regions. These results also reinforce which benchmarks have less potential for sharing available – the

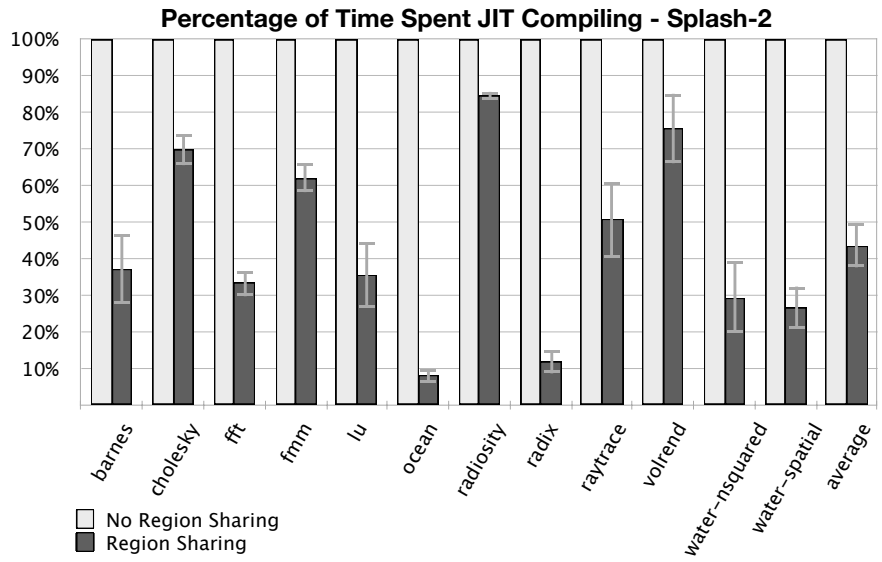


Figure 4.10

Percentage reduction in time JIT workers spend compiling when using region sharing.

compilation time for `radiosity` is reduced by only 15%, and previously (in Figure 4.8) only exhibited 47% of its regions as sharable. The greatest reductions can be seen in `ocean` and `radix`, which are reduced by 92% and 88%, respectively. These results reinforce the idea that the use of thread-agnostic regions and region sharing can greatly reduce JIT compilation time.

4.4 Summary and Conclusions

This chapter has demonstrated an example of how wasted effort during code generation can be mitigated in order to improve process virtual machine performance. In this chapter, the virtual machine considered is a dynamic binary translator used for the purpose of instruction set simulation. An approach to multithreaded region-based JIT compilation has been presented, that enables the sharing of common regions between multiple application threads. For this, the code generator must be modified to generate regions that do not depend on any thread-specific context, but are generic enough to be executed in any thread context. Surprisingly, this thread-agnostic code generation not only enables region sharing, but also facilitates a minor performance improvement when applied in isolation. Sharing of regions between application threads then reduces the pressure on the JIT subsystem and improves performance further. Experimental evaluation finds that on average 76% of all regions can be shared in the Splash-2

benchmarks when run with 128 threads in the ARCSIM dynamic binary translator. This results in an overall performance improvement of 1.4x averaged across all benchmarks and over a state-of-the-art thread-private region compilation approach without region sharing.

Chapter 5

Sharing Profiles for Selective and Focussed Code Generation for PVMs

The Android mobile operating system, which powers most of today's smartphones and tablets, has recently reached over a billion activated devices worldwide [Pic], being shipped on a billion devices in 2014 alone [Str15]. With an estimated 1 million new devices activated each day, its wide availability, and 1.5 billion application downloads per month, Android continues its upward trend [Pag].

Central to Android are applications, which can be acquired by users either through an "app store" such as Google Play, or by downloading and installing the application's package file (called an APK on Android) from a third-party site. Android programs are traditionally written in Java, compiled to a portable bytecode representation called DEX for distribution, and then executed within a *Dalvik virtual machine*, an HLLVM.

The original implementation of the Dalvik VM, itself called Dalvik, executed DEX bytecode within an interpreter, with a JIT compiler with limited optimisation potential added later - see Section 2.5 for further information about Dalvik. The latest version of Android, 5.0, introduced a new implementation of the Dalvik VM called ART (*Android RunTime*). This runtime was released as an experimental version in Android 4.4, but completely replaced Dalvik in 5.0. It replaces bytecode interpretation and JIT compilation with ahead-of-time (AOT) compilation. This chapter concerns itself with a system for selecting particular methods to be either i) the **only ones to be compiled**, or ii) the **targets of heavy optimisation**, depending on the requirements of the device.

5.1 AOT Compilation in ART

Whenever a new application is downloaded onto an Android mobile device, AOT compilation takes place, compiling all of the DEX bytecode to the native machine code of the device's CPU architecture, and packaging it up into an *OAT file*. This compilation is not performed server-side as this is a specific design decision for Android. OEMs are free to customise the ART VM implementation found within the Android Open Source Project and also provide it on a variety of different CPU architectures. The store that users download applications from cannot make assumptions about the ART VM running on the device, precluding this server-side compilation from taking place. The on-device AOT compilation aims to compile every available Java method,[1] but any method that is either unable to be compiled – or is purposefully chosen not to be compiled – can have its DEX bytecode executed in ART's interpreter instead.

5.1.1 Selective Compilation

Why select only certain methods to be compiled? While many smartphones popular in developed countries have up to 32GB of flash storage available, phones targeted at developing markets are often more restricted in terms of storage capacity. For example, the Spice Dream Uno, which has been developed under the Android One program [Goob], has just 4GB of built in flash storage [Pho]. With intelligent selection of which parts of applications are compiled, savings in on-device storage can be made. This can also affect a device's capability to run multiple applications simultaneously, if larger binaries must be kept in memory. Most devices intended for developing markets in the Android One program only offer 1GB of RAM, a large percentage of which may be taken up by the operating system and system services.

More importantly, selecting only certain methods for compilation can produce savings in installation time. The portion of code generation time within installation time may not be viewed as a significant problem when considered as part of the full process of downloading and installing an APK, as the bottleneck here is likely to be the transfer of the file from the Internet. However, all applications installed on the device must be compiled when a new device is turned on the first time. Additionally, they must be recompiled whenever the OS is updated. This can lead to a significant amount of time being spent in compilation. For example, in the initial bootup of an Android Open Source Project (AOSP) build of Android 5.0.1 on a Nexus 5 device, 56% of the

[1] With the exception of class initialisers.

time was observed to be spent in the "app optimisation" phase, where native code is generated. It should be noted that this build does not include the standard Google applications that a complete factory image for a Nexus device would. It was observed that such a factory image took approximately 5 minutes to complete its initial bootup. Due to the locked down nature of the factory image, it was not possible to obtain precise measurements of how much of this time is spent in compilation. This time will increase whenever the OS is updated, as the user installs more applications, and will be even more noticeable on low-end devices.

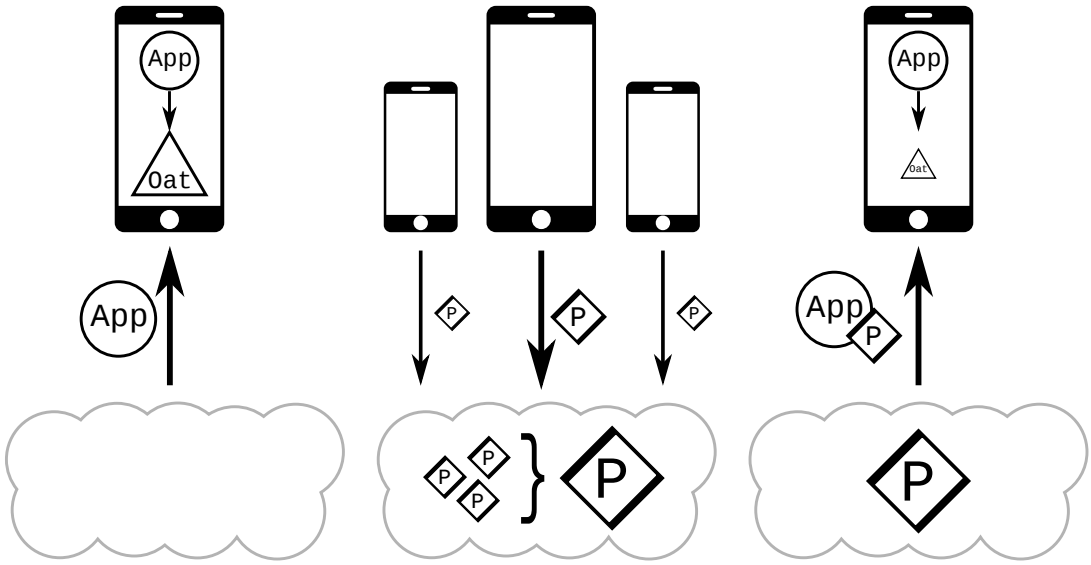
Considering potential to save code-size and installation time, it should be clear that situations exist where it is beneficial to only compile a subset of methods in an application.

5.1.2 Focussed Optimisation

It is also important to consider how identifying a subset of methods in an application could become beneficial to high-end devices in the future. The primary compiler used in ART is known as the *quick* compiler, but a code generator called the *optimizing* compiler is presently in development. With an increase in the number and strength of optimisations applied during code compilation, comes an increase in the length of time required to compile code [CMS+11]. It is feasible that some optimisations may not be included in the final version of the *optimizing* compiler, because of the need to find an acceptable balance between improving code quality and reducing installation time. The choice of which methods should be compiled could be used to instead determine which methods should be aggressively optimised, with the *quick* compiler used for the rest of the methods. This would allow the *optimizing* compiler to include significantly more powerful, but expensive, optimisations - or a standalone compiler such as GCC or Clang could even be brought to bear on the task if there were a significant improvement in performance to be gained.

5.1.3 Performing Profiling

By profiling applications, a *selective compilation* or *focussed optimisation profile* can be produced, and compilation can be focussed on only the methods of the application that are of most importance to its performance. This will lead to savings in both code-size and installation time if used for selective compilation, or performance improvements if used for focussed optimisation. As this code generation takes place when



(a) At first, users downloading the application AOT compile all of the contained bytecode, producing a large OAT file of native code. Users may then begin profiling the application at this stage. These initial users are likely to be the developers or beta-testers of the app.

(b) Once a profiling user has produced a sufficiently large profile, they can perform selective compilation or focussed optimisation. The resulting profile can also be sent to the profile server, an entity responsible for merging these incoming profiles into a unified profile.

(c) New users will be able to download the application bundled with this unified profile. The user can then immediately use the merged profile to perform either selective compilation or focussed optimisation, producing a more optimal OAT file.

Figure 5.1

An overview of the proposed system for crowd-sourced profiling to produce merged candidate method profiles for new users, allowing them to skip the profiling stage themselves.

the application is downloaded and installed, this leaves the question of how a relevant profile can be obtained. In certain situations, it may make sense to initially compile all code in the application, allow its initial execution to occur in a profiling mode, and then recompile the application with selective compilation or focussed optimisation at a later point.

Instead, this chapter proposes the idea that when a user downloads an application they can receive a profile as well, which removes the need for any individual profiling. These profiles will be *crowd-sourced* by the developers and willing users of an application. Figure 5.1 provides an overview of the proposed system.

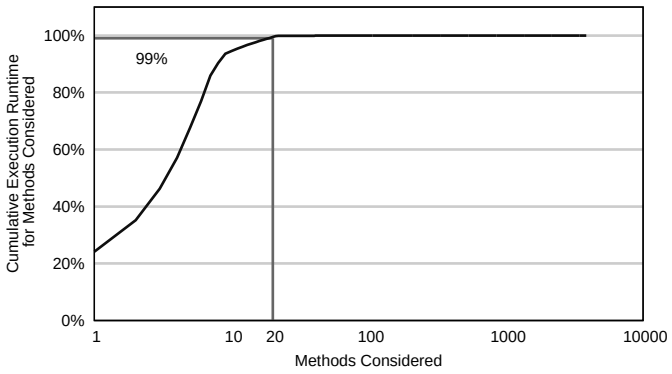


Figure 5.2

Cumulative heat graph for the Quadrant Android benchmark. While the benchmark contains over 1850 executed methods, just 20 are required to capture over 99% of the benchmark's run-time.

5.2 Motivating Example

As part of the installation procedure for applications running on ART, the DEX bytecode, which constitutes the executable code of the application, must be compiled into native code. Compiling and optimising DEX bytecode can take a significant amount of time for large applications, resulting in a lengthy and unexpected delay for the user. The output of this compilation is also a large amount of native code, much of which may not be executed frequently, if at all.

Figure 5.2 shows a distribution of the cumulative execution time, measured by this chapter's profiling system, against the number of executed methods considered for the Quadrant benchmark available on Android. While there are over 1,800 individual methods executed, and over 17,000 methods present in the APK, 99% of the benchmark runtime is consumed by just 20 methods, while over 15,000 methods are never even executed once. Clearly consideration should be made if it is wasteful to spend any significant effort compiling or optimising the remaining methods, or to consume space on the device storing the resultant native code.

If these hot methods were identified and then selectively compiled, considerable savings in both compilation time (and thus application installation time) could be made. While also allowing for compiled code-size to be reduced, this motivates the use of this saved time to focus code optimisations on methods that were selected for compilation as well. In order to identify these methods, Android already has a built-in profiling infrastructure [Gooc], however use of this demonstrates a noticeable impact to responsiveness which will negatively affect a user's experience of an application.

Furthermore, this chapter proposes the idea that multiple users using the same application will use it in such a way that, for most users, the hot candidate methods of an application will be the same.

In summary, this chapter presents techniques for efficiently profiling Android applications, collecting and merging these profiles together, and then using this merged profile to intelligently select which methods should be considered hot when new users install the application in the future. This is done either to perform selective compilation or focussed optimisation, depending on the particular need of the device. This chapter will also evaluate the effects of this selective compilation and focussed optimisation on performance, code-size, and installation time. It will show that a significant reduction can be made in installation time and code-size, without unduly affecting performance for selective compilation, and that performance can be improved when used for focussed optimisation. Profiling will be used in order to generate *selective compilation* and *focussed optimisation* profiles, which will collectively be referred to as *candidate method profiles*.

5.3 Profiling Android Applications

This section will describe the profiling system that has been added to ART, that allows informed decisions to be made about which methods should be included in an application's candidate method profile. Sections 2.6 and 3.1.5 provide background information and related work about profiling, respectively.

5.3.1 Profiling using bytecode probes

In order to get information about which methods are most heavily used in an application, it is useful to gather hit counts for every basic block in the application. This information is obtained by adding additional opcodes to the DEX instruction set format, using unallocated opcode values in the specification. The first new opcode, `report-bb`, includes a probe ID as an operand. To instrument an application for profiling, this bytecode is inserted at the head of every basic block in the program, with a new probe ID sequentially associated for each basic block. Hitting such an instruction causes the VM to increment a thread-local counter for the probe ID.

There are two points when ART should dump snapshots of this information. Whenever a thread quits, the snapshot of its stored counters are dumped to a file. Applica-

tions are also instrumented with a second bytecode, `dump-profile`, that is placed at the start of any `onPause` method in any subclass of `Activity`. This method is called whenever an Activity (in other words, a view of a particular Android application) is paused or closed, as part of the Android Activity Lifecycle [Gooa]. This bytecode will dump the counters for all threads that are currently alive in the application. This is necessary to ensure that the profiles are obtainable whenever a user navigates away from an application - a user does not typically "quit" an application on Android. Figure 5.3 provides an example overview of multiple threads producing snapshots and how from these thread-local views of basic block hit counts, analysis can be performed to produce the final view of how often each basic block was executed.

For each basic block in the application, in order to calculate the total number of times the block was hit:

- Take the observed hit count at the final snapshot produced by an `onPause` method being called, $C_f(B)$.
- Every thread that hit the basic block and then quit must also be considered. Separate the counts from quitting threads ($C_q(B)$) into two sets - those that came from threads that were not alive when the final `onPause` snapshot was produced, *DQT* (Dead Quitting Threads), and those that were alive, but quit afterwards, *AQT* (Alive Quitting Threads).
- Add the observed hit counts for threads in *DQT* to the total.
- Add the observed hit counts for threads in *AQT*, but first subtract the counts seen in the final `onPause` snapshot.

This leads to the following equation to calculate the execution count of a basic block, $C(B)$:

$$C(B) = C_f(B) + \left(\sum_{C_q \in DQT} C_q(B) \right) + \left(\sum_{C_q \in AQT} C_q(B) - C_f(B) \right) \tag{5.1}$$

This calculation provides a precise view of how often each basic block was hit by any thread in the application. It adds the count visible in the final `onPause` snapshot (the first term in the sum), to the counts seen in any threads that had quit and therefore deleted their counts. Threads that quit before the final `onPause` snapshot are accounted for in the sum's second term, and threads that quit after are handled by the third.

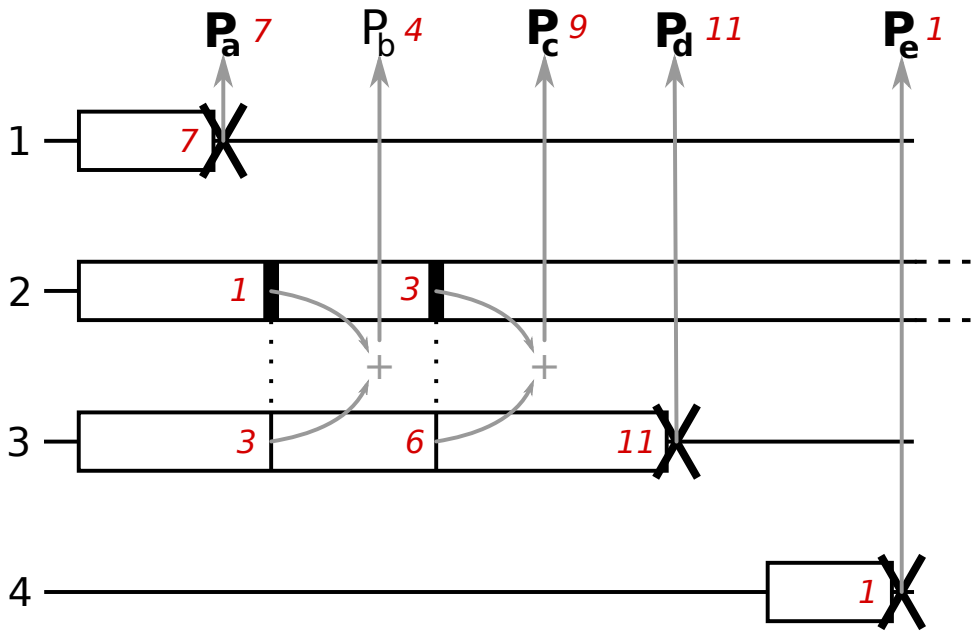


Figure 5.3

An example of a running application producing snapshots of basic block hit counts, which must be combined to produce a global hit count view. The thick black lines of T_2 are `onPause` methods being hit, and the crosses show when threads quit. P_a and P_e are combined as counts from dead quitting threads (DQT), as well as the count from the final `onPause` snapshot, P_c. The count from P_d as well is added, but because it was alive during the final `onPause` snapshot, the count "seen so far" in that snapshot must be first subtracted. The red values show the observed hit count for a single basic block. In this example, this means the global observed hit count of this basic block is $P_c + P_a + P_e + (P_d - P_c(T_3)) = 9 + 7 + 1 + (11 - 6) = 22$.

5.3.2 Finding candidate methods by heat

To determine which methods of an application should be compiled or optimised, an estimate is needed of how much thread CPU time was spent in each method, otherwise known as the "heat" of the method, or its occupancy. There is little to be gained by speeding up methods that take up a small percentage of runtime of the application, if the methods that take up larger percentages are ignored. The heat of a method, $H(M)$, is estimated by summing together the hit count of each basic block, $C(B)$, weighting this count by the sum of the weights of each DEX bytecode instruction, $W(I)$, that comprises the basic block, as shown in the following formula:

$$H(M) = \sum_{B \in M} \left(C(B) \times \sum_{I \in B} W(I) \right) \tag{5.2}$$

This heat calculation provides an estimation of how much CPU time was spent by all threads in the method. In other words, this is the estimated *sequential runtime* of the method. This is an estimation because the weight for each DEX instruction, $W(I)$, is based on the average number of ARM assembly instructions that must be executed in order to interpret the DEX instruction in the steady state. With a view of the sequential runtime of every profiled method in the application, these methods are sorted by their sequential runtime, and the hottest methods in this sorting are selected until the desired percentage of the total estimated sequential runtime is reached, typically 99%. Figure 5.4 provides an example of calculating estimated sequential runtime, or heat, for the method $M1$. The red value next to each basic block is $C(B)$, the observed hit count for that block, and the values seen in each basic block are the weights of the bytecode instructions. Figure 5.4 also shows the selection of the top 99% of sequential runtime of the application, thereby selecting methods $M1$, $M2$, and $M3$.

The technique for selecting candidate methods presented here is naturally drawn to methods with more DEX instructions. This heuristic is based on the notion that the larger a method, the more opportunity there is to perform code optimisations such as common subexpression elimination and constant folding, which may further enable other optimisations such as dead code elimination. A result from this is that a programmer may wish to manually merge methods making them a more desirable candidate for compilation. This is akin to method inlining that may be performed by some compilers in order to both reduce function call overhead, and also increase the optimisation potential of the parent method. A counter point to this is that there are diminishing returns from manually merging many methods in this way, and in some cases such inlining may actually degrade performance, if a method becomes so large that all of its hot code no longer fits within the processor's instruction cache, therefore it is not actually suggested that programmers always manually merge methods in order to improve optimisation potential.

This technique is also drawn towards those methods that contain more complex DEX instructons. Without the ability to actually assess the impact of optimisation on a particular method ahead-of-time, there is an expectation that it would be most beneficial to focus on selecting methods that contain DEX instructions that are comprised of more assembly instructions. In the same way that a larger method has more potential

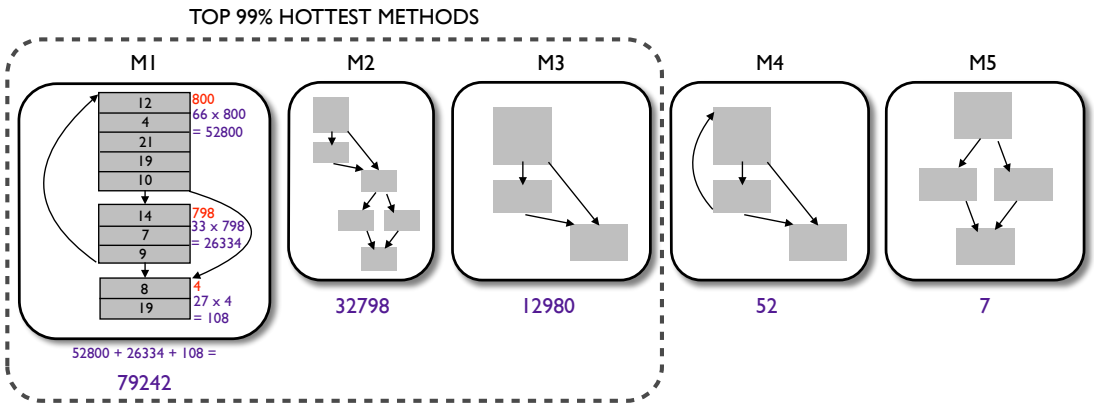


Figure 5.4

An example of heat calculations for methods observed in an application, with a selection of the top 99% hottest methods. Selecting only the first three methods is enough to capture 99% of the estimated sequential runtime of the application.

to be optimised, a "larger" DEX instruction (i.e., one that contains more assembly instructions) also has more potential to be optimised. As an example, consider an array access DEX instruction, which comprises a type check, a bounds check, and the actual array access. If a load from an array is followed shortly by a store to that same location after some calculation has been performed, the type check and bounds check of the store can be skipped. It should be noted however that this heuristic is limited by static analysis, so it may not always make the best decision compared to one made with the actual achieved speedup from compiling and optimising a method taken into account.

Despite the limitations of static analysis, with the selections made by this heuristic, code generators now have some information about which methods are worth compiling to maximise performance. This is information that can be used for both selective compilation and focussed optimisation.

5.4 Selective Compilation Impact

The impact of selective compilation on reducing code-size and installation time was evaluated for ART, while balancing performance, by profiling and executing a number of standard Android benchmarks currently available on the Google Play store. These benchmarks are CaffeineMark [Caf], Quadrant [Qua], Linpack [Lin], BenchmarkPi [Pol], Scimark [Sci], RayTracer [Ray], and AndEBench [And]. The execution times for these benchmarks were collected from a Nexus 5 device running an AOSP build of version 5.0.1 of Android. The device had its CPU frequency throttled to 1.2GHz,

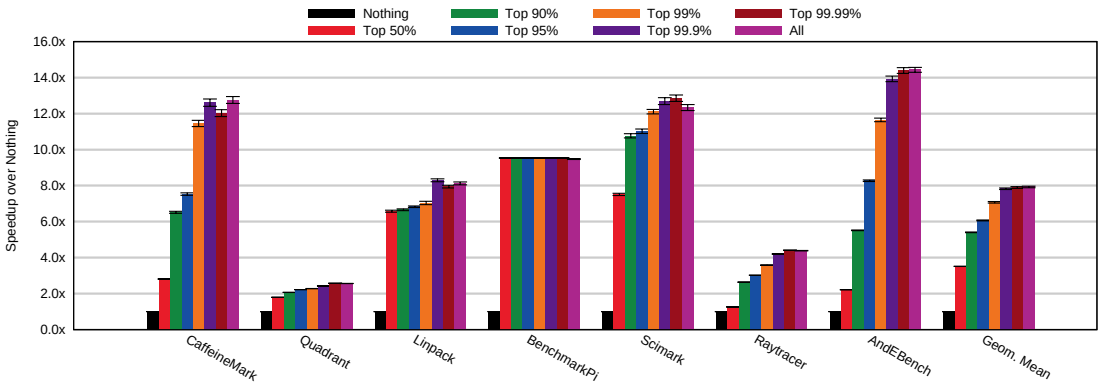


Figure 5.5

Relative performance increase when compiling only selections of methods that comprise 50%, 90%, 95%, 99%, 99.9%, 99.99% and 100% of the run time of an application. These results are compared to a baseline of compiling none of the application for common Android benchmarks. The error bars represent a 95% confidence interval.

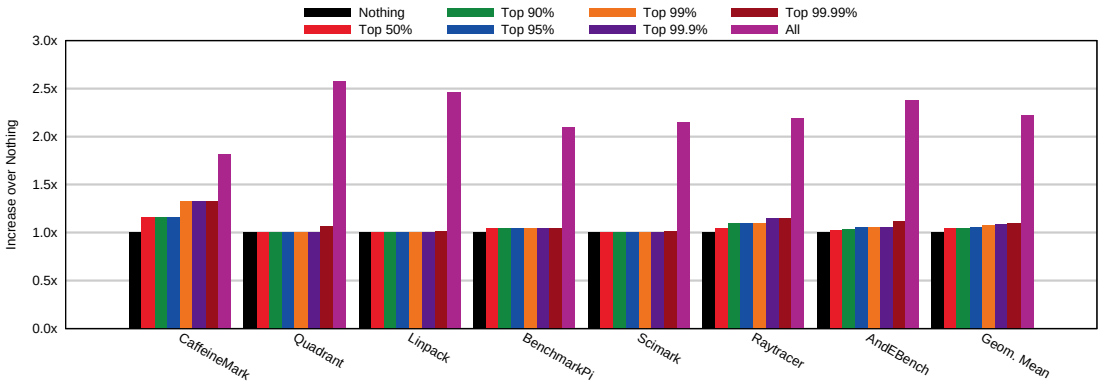


Figure 5.6

Relative increase in installed code-size when compiling only selections of methods that comprise 50%, 90%, 95%, 99%, 99.9%, 99.99% and 100% of the run time of an application. These results are compared to a baseline of compiling none of the application for common Android benchmarks.

and was configured to keep all 4 cores awake in order to reduce noise during experiments, so the change in runtime experienced through the different compilation selection schemes could be more accurately observed. Despite these efforts the platform still exhibited a large variation in execution times from run to run, so for each benchmark-scheme combination the benchmark was executed 50 times, with the arithmetic mean used as the result. Applications were installed 100 times to obtain their average install time.

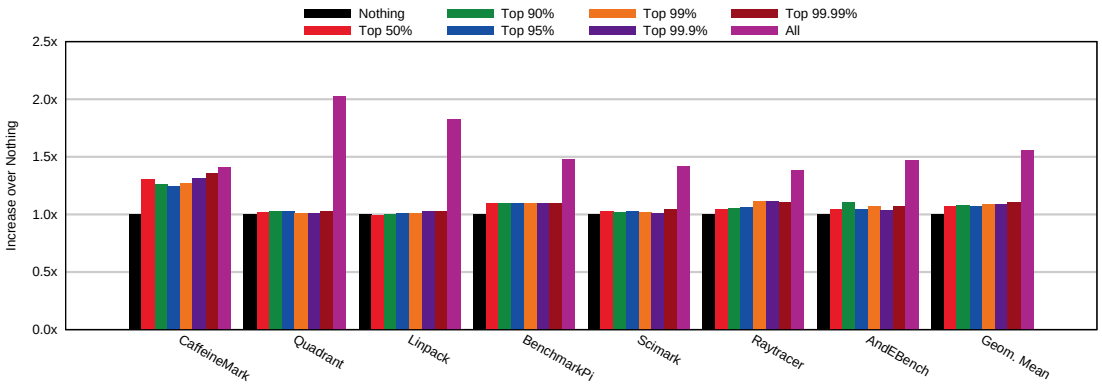


Figure 5.7

Relative increase in benchmark install time when compiling only selections of methods that comprise 50%, 90%, 95%, 99%, 99.9%, 99.99% and 100% of the run time of an application. These results are compared to a baseline of compiling none of the application for common Android benchmarks. There is a large amount of variation when compiling only a small number of methods, resulting in these unintuitive differences in installation time when not compiling the whole application. Nevertheless, compiling everything clearly significantly increases compilation time, in comparison to any selective compilation scheme tested.

Figures 5.5, 5.6, and 5.7 present views of the changes in benchmark performance, code-size and installation time as different compilation selection schemes were used. Eight compilation schemes were chosen, ranging from compiling no methods in the benchmark to compiling all methods. Between these extremes, different amounts of the top X% of the runtime of the benchmark – as estimated by the profiling system – were used to select methods for compilation. These percentages were the top 50%, 90%, 95%, 99%, 99.9%, and 99.99% of the application's profiled runtime. Table 5.1 shows the number of methods that were compiled for each benchmark, for each tested selective compilation scheme.

Figure 5.5 shows the relative increase in performance for all benchmarks, with the compilation of no methods set as the baseline. Looking at the observed geometric means for each scheme, it can be seen that 99% is often not sufficient to provide a performance improvement close to that of full compilation. On average, 99% achieves 7.07x speedup, compared to 7.93x when all methods are compiled. 99.9% is much closer, however, resulting in a speedup of 7.83x, while 99.99% achieves 7.9x. Cross-referencing this with Table 5.1 shows that this increase from 99% to 99.9% or 99.99% does not come with a significant increase in the number of methods that must be com-

Benchmark	50%	90%	95%	99%	99.9%	99.99%	Executed	All
CaffeineMark	2	5	6	7	8	11	67	96
	2.08%	*5.21%*	*6.25%*	*7.29%*	*8.33%*	*11.46%*	*69.79%*	
Quadrant	5	9	12	20	29	245	3850	17676
	0.03%	*0.05%*	*0.07%*	*0.11%*	*0.16%*	*1.39%*	*21.78%*	
Linpack	1	1	2	4	6	8	89	2414
	0.04%	*0.04%*	*0.08%*	*0.17%*	*0.25%*	*0.33%*	*3.69%*	
BenchmarkPi	1	1	1	1	1	1	81	295
	0.34%	*0.34%*	*0.34%*	*0.34%*	*0.34%*	*0.34%*	*27.46%*	
Scimark	2	5	6	7	9	13	54	3505
	0.06%	*0.14%*	*0.17%*	*0.20%*	*0.26%*	*0.37%*	*1.54%*	
Raytracer	4	16	21	27	31	32	102	261
	1.53%	*6.13%*	*8.05%*	*10.34%*	*11.88%*	*12.26%*	*39.08%*	
AndEBench	2	7	10	20	24	30	109	889
	0.22%	*0.79%*	*1.12%*	*2.25%*	*2.70%*	*3.37%*	*12.26%*	

Table 5.1

Selected method counts for each benchmark-scheme combination used in the selective compilation evaluation. Italics show the count as a percentage of all methods implemented in the benchmark.

piled, when the fraction of the total number of methods implemented in each benchmark is considered - at most, 12.26% of the methods are compiled, in the case of Raytracer. Section 5.7.3 will discuss the method inclusion rates for real-world applications, after selective compilation profiles have been produced by merging together profiles from multiple users. The gap between 99% and 99.9% shows a larger average performance increase than 1%, suggesting that while this method selection methodology has successfully selected most of the hot methods, it could still be improved. While selection of methods based on their weight as a combination of size, instruction weights, and execution frequencies is a good heuristic, this does not take into account the actual impact of optimisation on each method - not all methods enjoy the same boost in performance from compilation. It may well be the case that some methods selected at 99% are not as improved by compilation as other methods only selected in the extra .99%, and some insight into improvement potential from analysis of the methods may address this.

A final interesting observation from this is that in certain cases, it appears to make

more sense to not compile everything, compared to compiling 99.99% - this was observed for the benchmarks Quadrant, BenchmarkPi, Scimark, and Raytracer. This could be explained by the opposing idea that certain methods may actually be more quickly interpreted rather than executed as native code, if they are executed infrequently enough. Alternatively, this may be due to the effects of code locality: when only the hottest methods of an application are compiled, then they will be located closer to each other, and without cold methods between them, which may account for this improved performance.

Figure 5.6 shows changes in code-size, demonstrating that large code-size savings can be made in these benchmarks, with an average increase in code-size of 1.07x when compiling the top 99%, 1.08x when compiling the top 99.9%, or 1.1x when compiling the top 99.99%. When compared to an average increase of 2.23x when compiling every method in the application, the 99%-and-above schemes do not lead to a large increase in code-size, but, as shown above, can approach (or even surpass) the performance achieved when compiling everything.

Finally, Figure 5.7 demonstrates how installation time increases. While the noise encountered when installing the benchmarks leads to unexpected variations in time, the difference of a few methods from 0% to 99.99% of heat has little impact on compilation time in general. However, for all benchmarks a large increase in installation time can be seen when all methods are compiled, 1.42x on average. Compare this to a small increase in time when compiling only the top 99%, on average an increase of 1.06x, or at most 1.08x when compiling the top 99.99%. This saving in installation time could be used to perform focussed optimisation on these identified hot methods instead, as will be discussed later.

From this analysis of loss-of-performance against code-size and installation time savings with Android benchmarks, it would seem that just by compiling methods that comprise 99% of the runtime the performance of compiling everything is almost reached, with an average speedup of 7.07x compared to 7.93x achieved when compiling everything. Meanwhile, 99.9% and 99.99% become extremely close, at 7.83x and 7.9x respectively. There is no significant increase in code-size or installation-time for any chosen selection scheme that is less than compiling everything. From this, it would appear advisable to use either 99.9% or 99.99% as the best selection scheme.

5.4.1 Estimation Accuracy

The heat profiles gathered for each benchmark are based on observed hit counts, but also *estimated* instruction weights. This may raise the question of how accurate this estimation actually is. With data of the obtained performance impact for each benchmark affected by selective compilation, this can be assessed. Since the estimate of the heat of a given method is based on execution weights for bytecode instructions derived from the execution of the interpreter, each method's heat in the recorded profile can be modified based on whether it was compiled or not to estimate the impact of compilation. If a method is included for compilation, then its observed heat is divided by 8, otherwise it is not changed. This factor of 8 is based on the average observed speedup for the benchmarks when going from compiling no methods in the benchmark to all of them. Using the observed heat when compiling every method as the baseline, the extra observed heat when not compiling every method can be compared to it for each scheme, giving an estimation of the impact on performance.

Figures 5.8 and 5.9 plot the estimated and observed slowdowns for each benchmark through the different tested selective compilation schemes. This shows that the trend for each estimation curve roughly follows that of the actual achieved performance with a strong correlation (a coefficient greater than 0.92 for all benchmarks, averaging 0.96), indicating that the correct methods are being selected for compilation at each threshold, despite the relatively naive estimation method. Errors in the estimation may arise from the fact that the 8x improvement through compilation will not necessarily be true for all methods, as the context where a bytecode is used is also important, but is not considered by this approach. At 99.9% and 99.99% the estimated and observed performances are very close to full-compilation performance, sometimes even surpassing full-compilation performance in the observed case. This unexpected improvement over full-compilation is another effect the estimation is unable to predict. In the case of CaffeineMark, the opposite may have happened - neighbouring hot methods compiled at 99.9% may have been pushed apart at 99.99%. To summarise, the average error between estimated and observed performance was only 0.02 for 99.9%, and 0.01 for 99.99%, so it has been shown that this estimation method is capable of capturing the important methods for these benchmarks.

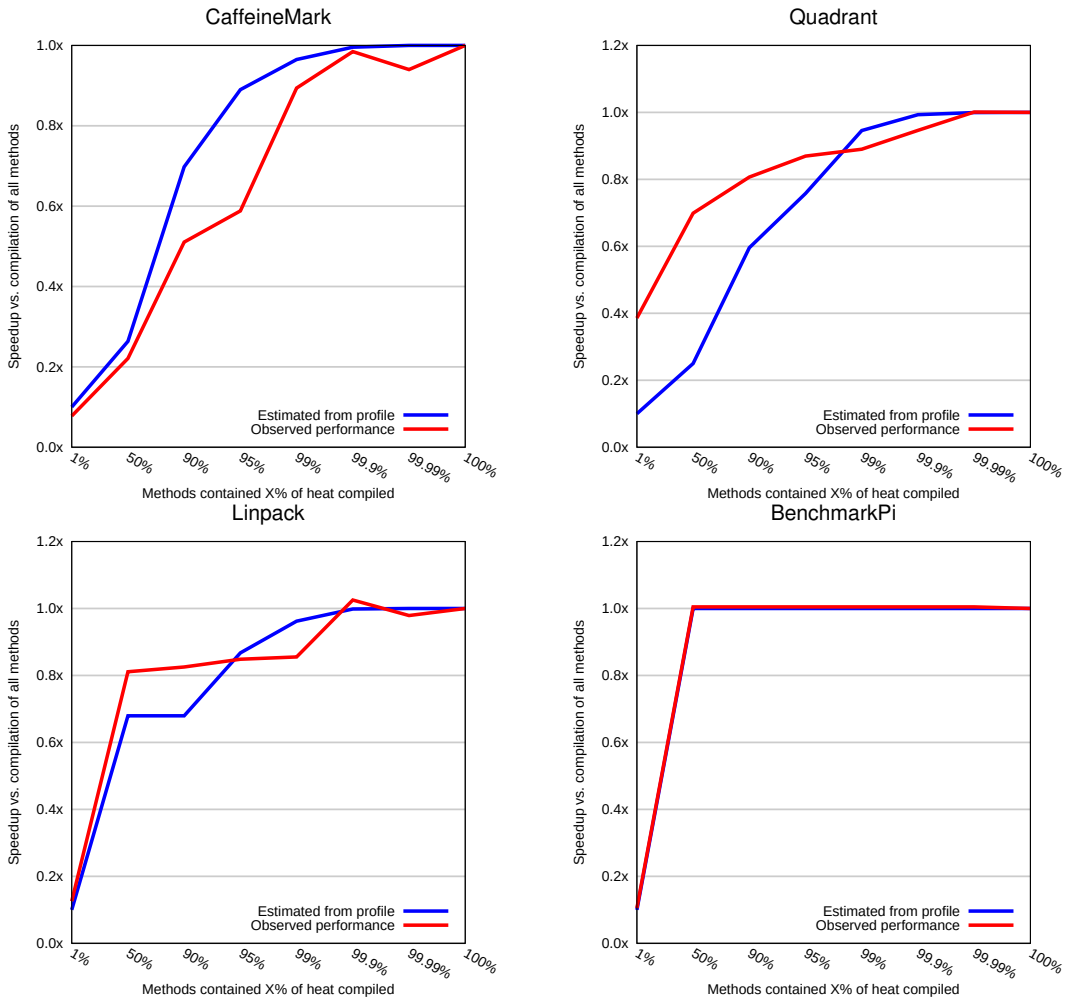


Figure 5.8

Plots of estimated performance impact against observed performance impact through selective compilation for four Android benchmarks.

5.5 Performing Focussed Optimisation

The previous section has shown how only compiling a small number of methods for standard Android benchmarks can lead to performance equivalent to that achieved when all methods in a benchmark are compiled. This can be done with significant savings in both code-size and installation time. While this itself is no great revelation, it does indicate the potential to direct optimisation effort when compiling applications. If compiling less than 10% of the methods in an application can lead to performance as good as compiling everything, how much could be gained by spending extra optimisation time when compiling these methods?

ART currently compiles all methods with its original *quick* compiler, so-called for its ability to compile code quickly, rather than its ability to produce quick code. An

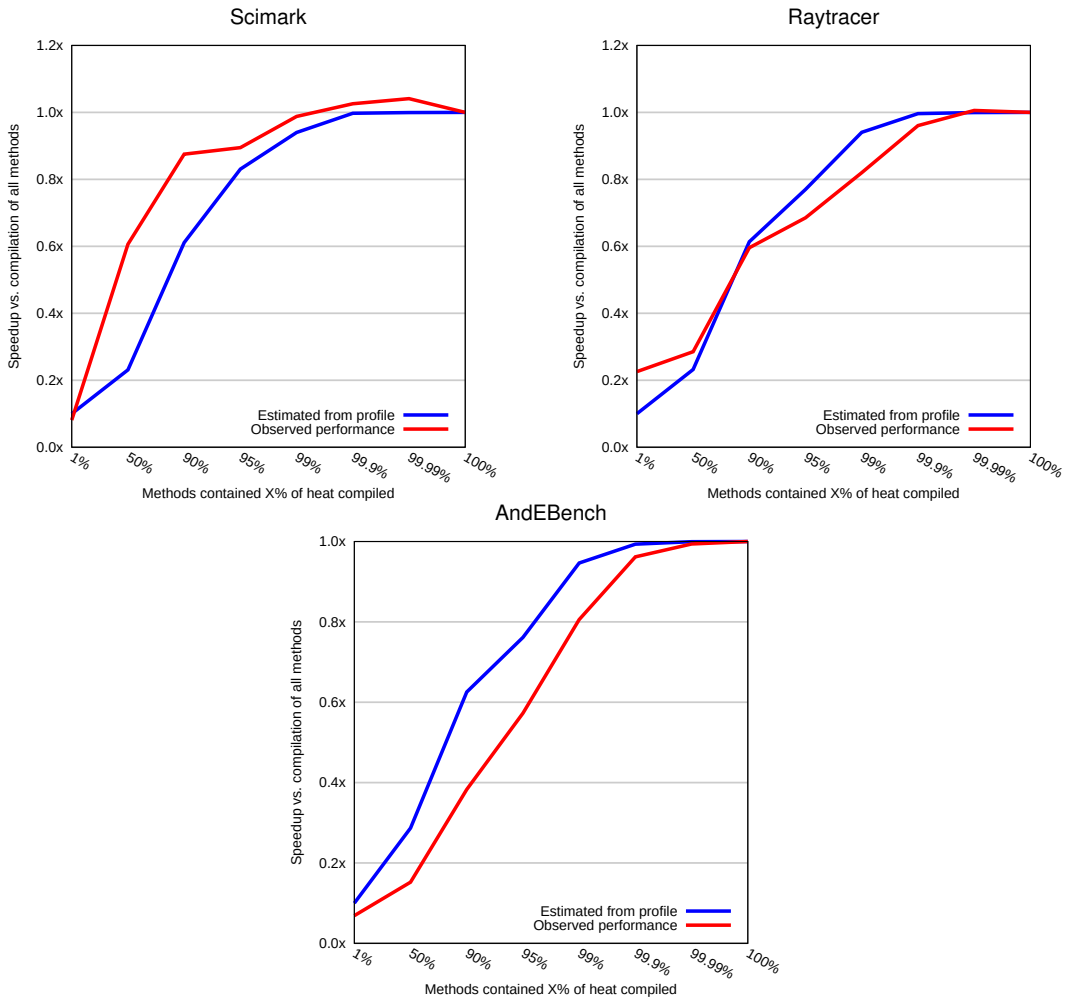


Figure 5.9

Plots of estimated performance impact against observed performance impact through selective compilation for three Android benchmarks.

optimizing compiler is also in development, and has in recent weeks been selected as the default compiler for future versions of ART. The developers of the optimizing compiler still have to trade off the potential performance gains from any optimisations against their cost to actually perform in terms of time. While this is primarily an issue for JIT compilation as the compilation cost must be paid at runtime, this is still a concern with AOT compilation performed on a mobile device for the reasons provided in Section 5.1.1.

This section looks at various methods in the Android benchmarks tested in the previous section, and investigates how more powerful optimisations could lead to performance improvements over what is currently achieved by ART.

5.5.1 Prototype for a Powerful Code Generator

The prototype code generator used in this section, called DEXTROSE, was initially developed in order to add AOT compilation to the earlier Dalvik VM, before ART was introduced to replace it. As Dalvik relied on an interpreter plus JIT to execute code, this prototype AOT compiler was designed to produce native code that could be placed immediately in Dalvik's JIT code cache as the application was initially loaded. As the VM executed the application, these blobs of optimised native code would be indistinguishable from any other piece of native code the JIT may have generated.

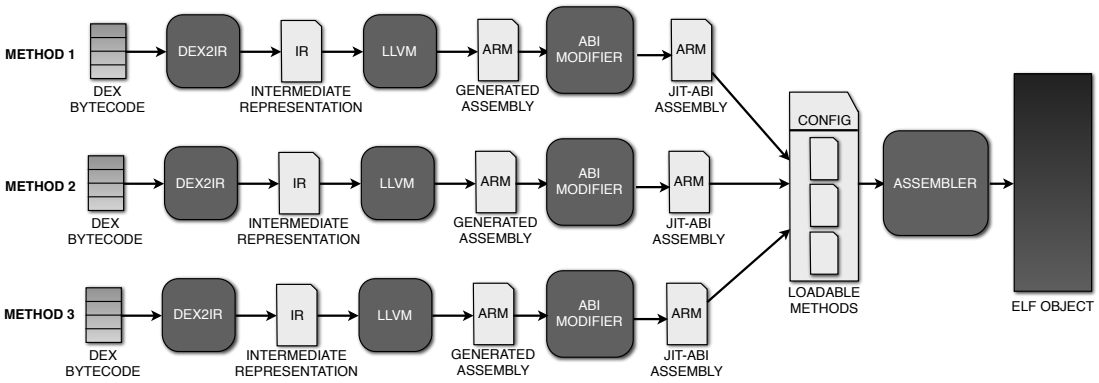


Figure 5.10

An overview of the DEXTROSE code generation flow.

The code generated by this prototype can be used as a measure of the optimisation potential for ART's code generator, in comparison to what it currently achieves. Figure 5.10 provides an overview of how DEXTROSE generated optimised native code. Each method selected for focussed optimisation was processed by a DEX2IR tool, that used SOOT [VRCG$^+$99], a Java optimisation framework. SOOT could be used to produce three-address Jimple code from a DEX file that could then be transformed into C. This C was then compiled to native code using Clang – a powerful off-the-shelf compiler based on the LLVM compiler framework. This native code was then processed further, in order to match the ABI of the JIT traces produced by Dalvik, so execution could transfer between these traces and the interpreter. Each piece of native code was then emitted into a single object file, with a table in the header describing which method each piece of native code was generated for. The resulting object file was then stored within the intended application's private data storage. When the Dalvik VM launched an application, it checked for the presence of one of these object files, loaded it, and copied it into the code cache. The pointer to this native code was then registered with the lookup structures Dalvik used to find JIT traces, as directed by the

table in the header of the object file. With this prototype, the potential for accelerating these benchmarks through AOT compilation with powerful optimisations can be investigated.

5.5.2 Optimised Code Production Examples

As this code generator is still a prototype, it is not mature enough to handle all methods that could be optimised. For this reason, this section highlights a few cases where the code generator was able to obtain a significant speedup over the code generated by ART. In each case the same Nexus 5 device was used, while the code generated by ART running on version 5.0.1 of Android is compared to the code generated by DEXTROSE running in Dalvik in version 4.4 of Android, the last version of Android that supported Dalvik. The compiler flags used to generate assembly for the C code produced from the Jimple format were `-mcpu=cortex-a15 -mtune=cortex-a15 -O3 -mfpu=neon -mfloat-abi=hard`, but it should be noted that the production of C code from Jimple code did not precisely match the Dalvik interpreter's handling of floating point values in situations such as NaNs and infinities. The author reiterates that DEXTROSE is a prototype, and is presented in this thesis to show the potential for speedups that could be achieved with more optimisation effort.

CaffeineMark – As a benchmark with multiple sub-benchmark results within it, the Logic component could be heavily optimised by DEXTROSE in order to obtain a 10.83x speedup over ART. Essentially, this large method could be completely optimised away thanks to constant propagation and dead code elimination, allowing this significant speedup.

BenchmarkPi – As Table 5.1 shows, a single method is responsible for the runtime of this benchmark. Compiling this method with DEXTROSE was able to speed up this benchmark by 1.46x on average, primarily thanks to optimisation of floating point operations within the kernel of the benchmark.

Linpack – The presence of floating point operations in this benchmark could be exploited to produce a speedup of 2.36x over ART when optimising this benchmark with DEXTROSE, primarily the `daxpy()` method.

This section has shown that speedups can still be achieved for benchmarks in ART thanks to the use of powerful optimisation, which could be afforded if profiling is performed in order to select which methods should be subject to this optimisation.

5.6 Crowd-sourced Profiling

Thus far, this chapter has discussed that profiling and the application of both selective compilation and focussed optimisation can be used when compiling DEX code for execution in the ART HLLVM. This can lead to significant install time and code-size savings without a large drop in performance for applications when used for selective compilation, and improved performance when used for focussed optimisation. However, if one of the savings is to be made at installation time, a user cannot benefit from selective compilation if they must first install the application in order to have it profiled. As for focussed optimisation, a user may be keen to enjoy the benefits of improved performance, without having to wait for a profiling phase to complete.

In order to have most users avoid the profiling stage completely, this thesis proposes that these *candidate method profiles* could be crowd-sourced. The aim with such a system is that a user can download a new application with a crowd-sourced candidate method profile attached and then use it to perform selective compilation or focussed optimisation when initially compiling the application, without significant impact to their execution of the application.

Figure 5.1 provides an overview of the proposed system. If a user downloads an application, and no profiles are currently available for the application, then the application will initially be executed in a profiling mode, until enough profile data has been generated from the user. This profiling data can then be sent back to the application store, or indeed whatever entity is responsible for this crowd-sourced profiling. As the responsible party collects profiles from multiple users, it can apply a merging scheme to produce a profile that combines execution information from all users who have submitted heat profiles. Then as new users – primarily those that will benefit most from selective compilation or focussed optimisation – download the application for the first time, they can obtain this merged candidate method profile, and use it to perform selective compilation for code-size and installation time savings, or focussed optimisation for improved performance. In some cases, this initial profiling could be performed by the developers of the application, before it is submitted to the application store. The Google Play store also supports alpha and beta distribution systems, so this could also be an opportune time to perform this profiling. It is also important to note that it is not a necessary requirement for the Google Play store to support this profile merging system; it could be performed by a third-party.

5.6.1 Merging Heat Profiles

As the party responsible for crowd-sourcing this profiling information receives heat profiles from multiple users of an application, it must somehow merge these profiles into a new candidate method profile for users who are downloading the application for the first time. This thesis proposes two schemes for merging profiles: a *conservative* approach and a more *inclusive* one.
In the **conservative** scheme, the users' raw heat profiles are considered:

- Let each user heat profile, P (taken from the set of all user heat profiles U), be a set of (M, h_M) tuples, where M is a method, and h_M is its associated absolute heat, the $H(M)$ calculated in Section 5.3.2.
- For each P, calculate R_P, a set of (M, R_M) tuples, where R_M is the runtime ratio of M in P. R_M is calculated by $\frac{h_M}{\sum_{h \in P} h}$
- For all M that appear in at least one P, calculate the arithmetic average of R_M observed across all P_{user} in U, called A_M. If $M \notin P_{user}$, then $R_M = 0$.
- Sort all M by A_M in decreasing order.
- With an empty merged candidate method profile P_{merged}, put the top ranked M into P_{merged}, adding A_M to a running count, C.
- Repeat until C exceeds some threshold T, usually defined to be 99%.

In the **inclusive** scheme, the users' candidate method profiles are combined:

- Let each user candidate method profile, P (taken from the set of all user candidate method profiles U), be the set of M methods that were selected for the user's candidate method profile.
- Let A be the union of all P in U - every method that appears in some profile.
- For each M in A, let C_M be the count of how many P profiles M appears in.
- With an empty merged candidate profile, P_{merged}, place M in P_{merged} if $C_M > 1$.

The inclusive scheme aims on average to include more methods than the conservative approach. Figure 5.11 shows a simplified example of how absolute heat profiles for five users using an application containing five methods can lead to the selection of different methods depending on the merging scheme used.

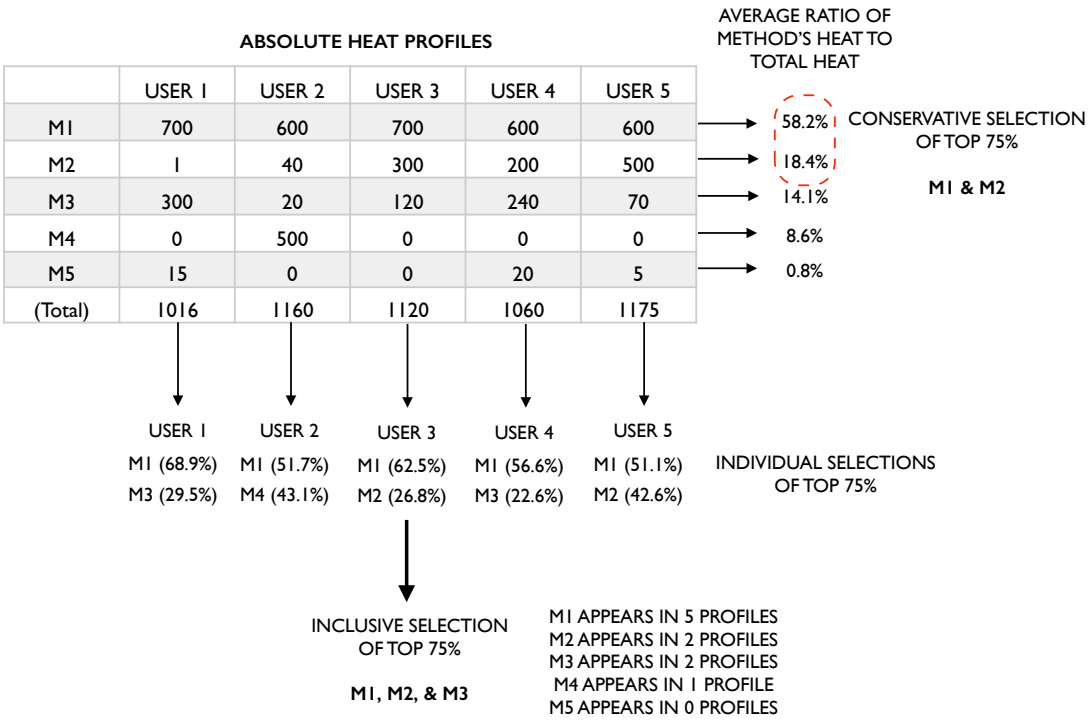


Figure 5.11

An example of how the conservative and inclusive merging schemes can merge profiles produced by five users for a single application together, when a 75% threshold is used for selection. Intuitively, the conservative scheme takes the average heat ratio seen of each method, and selects the top 75% of heat, while the inclusive scheme looks at the actual selection each user has made, and includes any method that appears in at least two users' selections.

5.7 Evaluation of Crowd-sourced Profiling

The merging of candidate method profiles from multiple users is intended to provide a generic profile that is similar enough to any participating user's profile. If a user's profile is not similar to the merged profile, then the benefits of selective compilation and focussed optimisation will not be visible. This section tests the claim that candidate method profiles can be crowd-sourced for new users who wish to download an application and perform selective compilation or focussed optimisation without having to profile the application themselves. In both use cases, using the merged profile should not result in the user failing to compile or optimise methods which take up a large amount of the observed heat of their own profile. If they do, this will either lead to drastically reduced performance in the case of selective compilation, or an extremely small gain in performance in the case of focussed optimisation, in comparison to what

could be achieved.

To test this claim, a selection of 21 free Android applications that are currently popular in the Google Play store were chosen. These applications were instrumented to obtain precise hit counts when executed on a Nexus 5 device in a modified version of the Dalvik VM. See Table 5.2 for the full list of applications. To obtain crowd-sourced data, seven volunteers were asked to act as profiling users in this experiment. Each user was instructed to use all the applications in any order. They were given no limit on how long they could use each application for, or how to use any of the applications.

The Android SDK also has a tool that allows events to be randomly generated for applications, called `monkey`. This tool was used to create two "monkey" users, that will be included in this evaluation, in order to see if the use of such random events is sufficient for profile generation. The first monkey ran each application once for 5 minutes, and the second ran each twice for 5 minutes each.

5.7.1 Estimation of Performance Impact

With hit count data obtained from each user for each application, an estimated heat profile can be calculated for each user, as described in Section 5.3.2. This provides an estimate of the time spent in each method by the user. These heat profiles are used to estimate the impact in performance for each user, when using a crowd-sourced profile compared to their own one for selective compilation. In general the measurement of performance for real-world applications is a difficult process, necessitating the use of benchmarks in most cases to evaluate performance improvements. If a heat estimation can be considered to be one aspect of the "amount of work" an application performs, then it is still a useful tool to compare the utility of crowd-sourcing of profiling. Additionally, a strong correlation was shown in Section 5.4.1 between the estimated and observed performance of Android benchmarks.

In the evaluation of crowd-sourcing of compiling, it will be assumed that 99% is the threshold used to select profiles. The rationale for this is that because this will select fewer methods compared to 99.9% and 99.99%, cases where users have produced a profile that differs significantly from the merged profile will be more prominent. As an individual user will base their selective compilation profile on the methods that comprise the top 99% heat of the program, it is to be expected that when calculating how the execution time is affected by not compiling the other 1%, a performance impact of roughly 0.935x will be seen. Consider 100 methods that each have a heat of 8, so

Application	**Author**
(1) 1010!	Gram Games
(2) Angry Birds	Rovio Entertainment Ltd.
(3) Audible	Audible, Inc
(4) Auto Trader	Auto Trader
(5) Balance 3D	BMM-Soft
(6) BBC Weather	Media Applications Tech.
(7) Candy Crush Soda Saga	King
(8) Crossy Road	Yodo1 Games
(9) eBay	eBay Mobile
(10) Google Sky Map	Sky Map Devs
(11) Gumtree	Gumtree UK
(12) Instagram	Instagram
(13) Job Search	Indeed Jobs
(14) Kik	Kik Interactive
(15) Linken	Level Ind
(16) Manga Rock	Not a Basement International
(17) NHS Health	NHS Direct
(18) Classic Notes Lite	Fluffy Delusions
(19) Oxford Dictionary	Mobile Systems
(20) Skype	Skype
(21) Wikipedia	Wikimedia Foundation

Table 5.2

A listing of all the applications used by participants in the crowd-sourcing evaluation, a selection of the top free applications currently on the Google Play store.

therefore when compiled have a heat of 1 each. If a is the heat of the top 99% of methods when compiled, c is the non-compiled heat of the single method that is not included in the top 99%, and b is the heat when this single method is compiled, then this 0.935x figure is derived as follows:

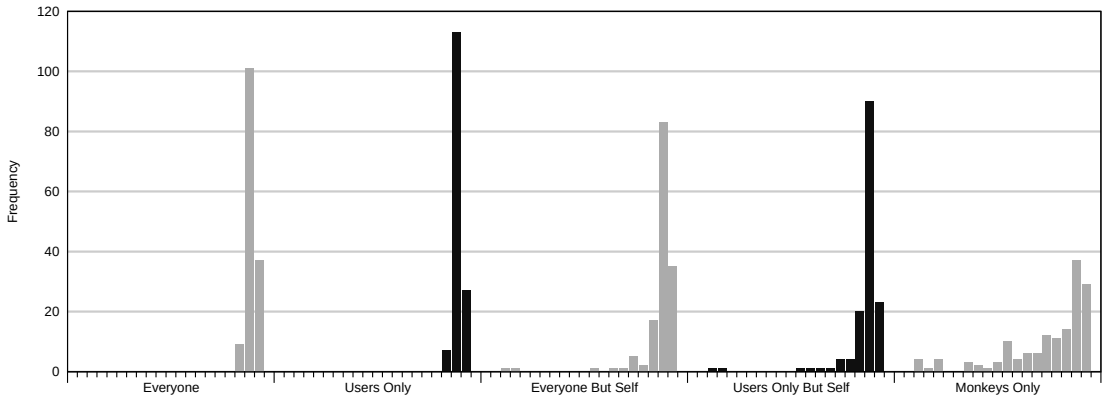
$$\frac{\text{all compiled}}{\text{top 99\% compiled}} = \frac{a+b}{a+c} = \frac{99+1}{99+8} = \frac{100}{107} = 0.935 \tag{5.3}$$

As the heat profile is effectively an estimation of the sequential runtime of the application, this is a useful way of measuring how using different candidate method profiles can affect how much effort has to be spent executing DEX bytecode in the application. In this section, this value will be referred to as *performance impact*. As this investigation is primarily concerned with how similar merged and individual profiles are, conclusions drawn when estimating performance impact with selective compilation as done in this section can also apply to its use for focussed optimisation: non-compiled methods can be considered to be non-optimised methods, and compiled methods can be considered to be optimised methods.

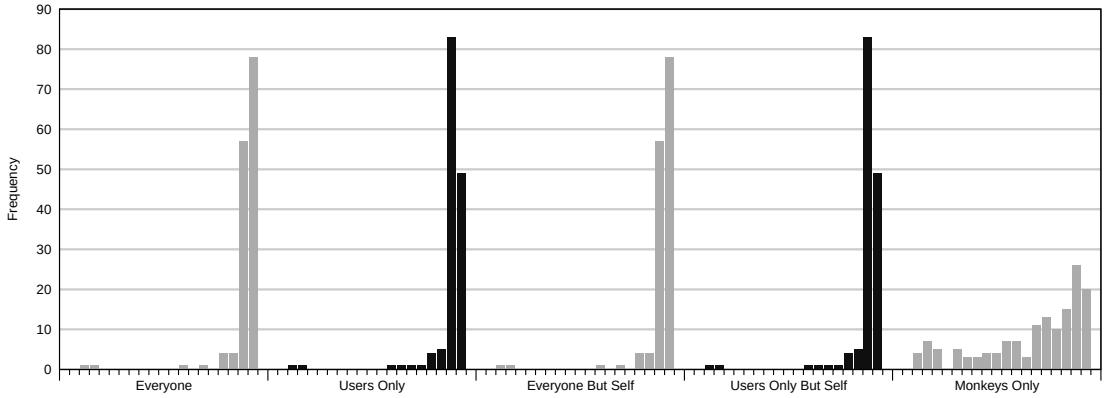
This analysis will present five different groupings for users when producing a candidate method profile, showing how the merged profile produced by each grouping will affect the performance impact for each user. This is done by applying the performance impact estimation as described above, for each user and application, when that user is given a merged selective compilation profile. The five groupings are:

- **Everyone** - both the profiles of users and monkeys are considered.
- **UsersOnly** - only profiles from real users, not monkeys, are considered.
- **EveryoneButSelf** - both profiles of users and monkeys are considered, but when merging a profile to be given to a user U, all users and monkeys apart from U are merged together.
- **UsersOnlyButSelf** - only profiles from real users, not monkeys, are considered, excluding the recipient user U as in EveryoneButSelf.
- **MonkeysOnly** - only profiles from monkeys are considered.

The inclusion of the third and fourth schemes allow for assessment to be made about whether it is possible for new users to download a merged profile that works for them, without having had to contribute to the profiling process themselves. Each



(a) Histograms of the estimated performance impact for the five different groupings within the **conservative** merging scheme.



(b) Histograms of the estimated performance impact for the five different groupings within the **inclusive** merging scheme.

Figure 5.12

Histograms of the results of applying the ten profile merging schemes. Each histogram has twenty 5% increments, showing that the majority of profile schemes fall within the [0.90x,0.95x) interval.

of these five groupings is used with the aforementioned *conservative* and *inclusive* merging schemes, producing ten assessed schemes overall.

5.7.2 Evaluation of Performance Impact of Merging Schemes

Figure 5.12 presents histograms of performance impact when each scheme is applied to each *(application, user)* data point. With 7 users and 21 applications, there are 147 data points. Each data point is placed into 5% interval buckets, so the highest bucket is [0.95x,1.00x] performance impact, the next is [0.90x,0.95x), etc.

It can be seen that in most instances the performance impact continues to fall within the [0.90x,0.95x) interval with all schemes - recall that the expected performance im-

pact when using one's own profile is 0.935x. For *EveryoneConservative* and *UsersOnlyConservative*, the impact never drops below 0.85x. This shows that there is a high degree of similarity between users' profiles when using candidate method profiles for selective compilation.

As the intended use of this crowd-sourced profiling system is to provide new users with a candidate method profile *without* requiring them to profile themselves, the *EveryoneButSelf* and *UsersButSelf* sections of Figure 5.12 show the cases where the user that is using a merged selective compilation profile, is excluded from the creation of the merged profile. In these cases, the majority of impacts still fall within the [0.90x,0.95x) interval.

It is interesting to note that the performance impact is much more widely spread when only the monkeys are used to create the selective compilation profiles. This would suggest that it is necessary to involve actual users in the profiling process, and not just monkeys with random events, as might have been expected for more complex applications.

Two outlier data points were observed, where two *(application, user)* cases lead to a significant reduction in performance. In the first, a user using the eBay application had used the barcode scanner component of the application, taking up a significant portion of the sequential runtime seen in the profile, while no other user had done this. The second user had used a color picker widget in the Classic Notes Lite application, to a similar effect. In both cases, these methods were included in the *conservative* merging scheme, as evidenced by the outliers only appearing when they themselves are not included in the merging process. In the case of the *inclusive* merging schemes, the outliers can be seen in all five schemes, because even when the outlying users are included in the merging process, they are the only user to execute these methods, and so the minimum threshold of two users is not met.

Figure 5.13 shows the geometric mean of performance impact observed for each merging scheme. This shows that in terms of minimising the amount by which performance is reduced, the *inclusive* approach is better, but this may be because it includes more methods for compilation. Whichever scheme is used, the schemes where the impacted user is excluded from the profile merging (the *ButSelf* schemes) show that it is possible to successfully merge profiles and give them to new users, without significantly impacting their performance. Additionally, relying on monkeys alone is not advisable, although they can provide some benefit if included in the merging process with other users. Overall, it has been shown that in the average case, most users can

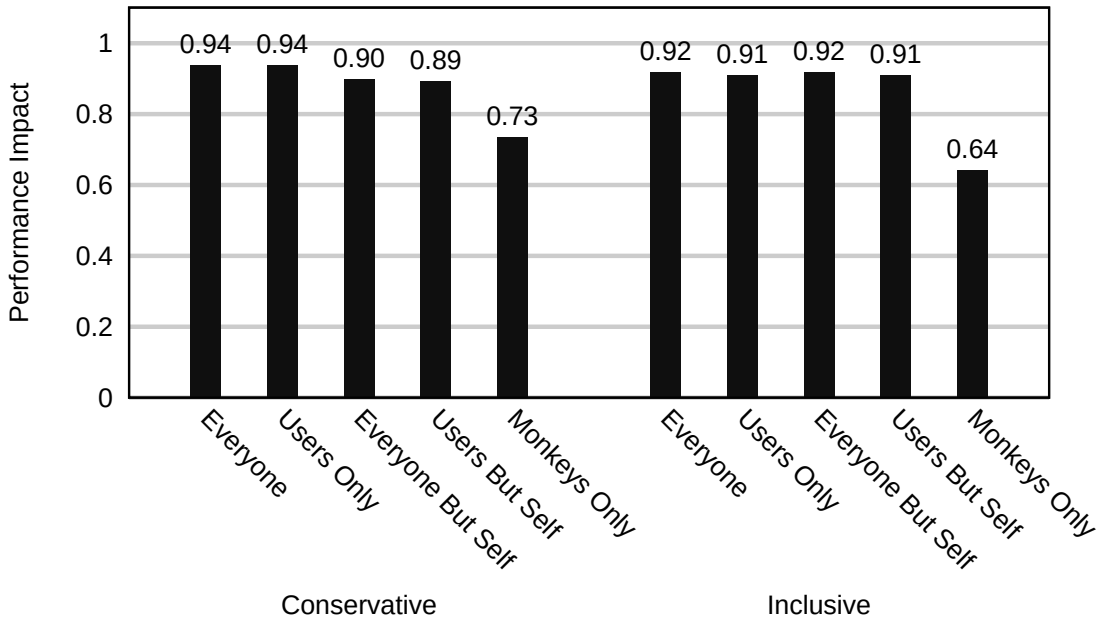


Figure 5.13

The geometric mean of speedup of sequential runtime for all ten schemes. The *inclusive* schemes slightly outperform the *conservative* schemes when not considering the user's own data.

readily use the merged profiles with no significant performance impact. As there is a possibility that the merging scheme can miss one or two methods for a minority of users, some possible improvements may need to be added to mitigate this, such as re-profiling, or JIT compilation of egregiously missing methods, although this does not appear to be a common occurrence. This will be discussed further in Section 7.2.

5.7.3 Evaluation of Method Inclusion Rates

Thus far this chapter has shown that the *inclusive* merging scheme provides slightly less impact on performance than the *conservative* scheme. It was suggested that this may be because the former scheme is likely to include more methods in its merged candidate method profiles. Figure 5.14 provides information about the arithmetic average percentage of methods in an application that are included in a merged profile. For each *(application, user)* data point, four metrics were calculated concerning methods selected for compilation by the merged profile, with respect to the user's own profile:

- **Select** is simply the number of methods included in the merged profile.
- **Extra** is the number of methods that the user was told to compile that did not

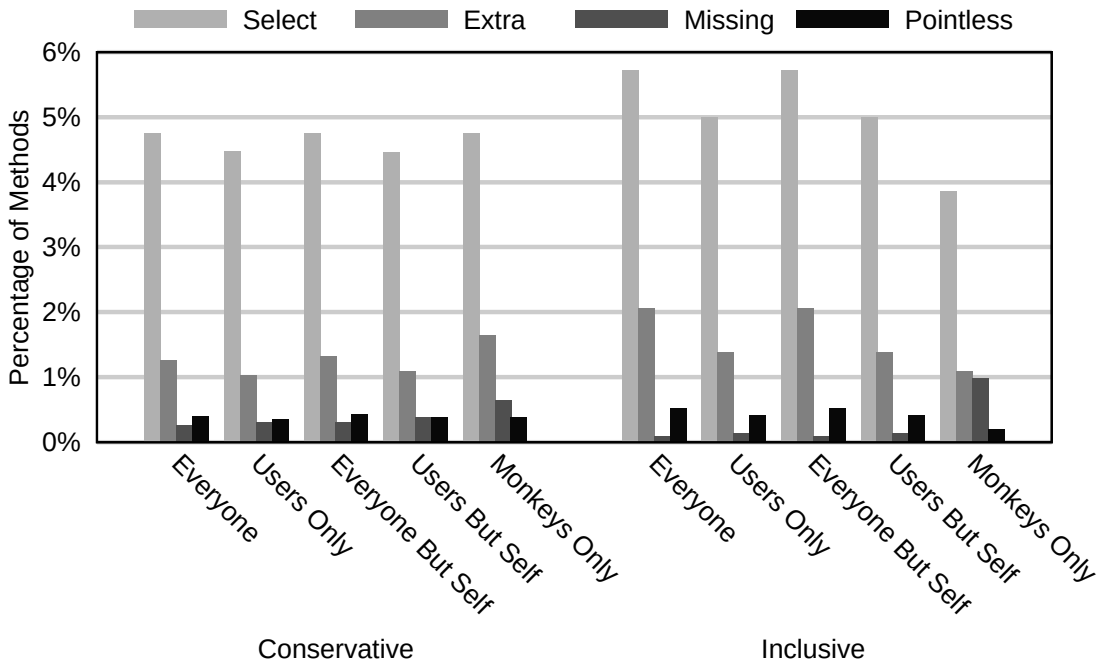


Figure 5.14

Graphs showing statistics of selectively compiled methods. The *inclusive* schemes select more methods for compilation without significantly increasing the number of *pointless* methods compiled.

feature in their own selective profile.

- **Missing** is the converse: the number of methods the user did not compile, that they would have with their own profile.
- **Pointless** is the number of methods that the user was told to compile when using the merged profile, that they never actually executed themselves.

Each metric is presented as a percentage of the total number of the methods that can be compiled in the application.

Figure 5.14 shows the arithmetic average observed percentage for each of these metrics, and shows that the *inclusive* scheme selects more methods for compilation compared to the *conservative* scheme (an average of 5.6% versus 4.6%.) The *conservative* scheme averages an inclusion of about 1.2% extra methods over a user's own profile, while *inclusive* results in around 1.5% extra methods on average. The *inclusive* approach is clearly more suitable for making sure a user's own methods are included in the profile, even in the case where they themselves are not included in the merging process - reinforcing the idea that there exists a high degree of similarity between candidate method profiles. Finally, we can see that users are not given a significantly large

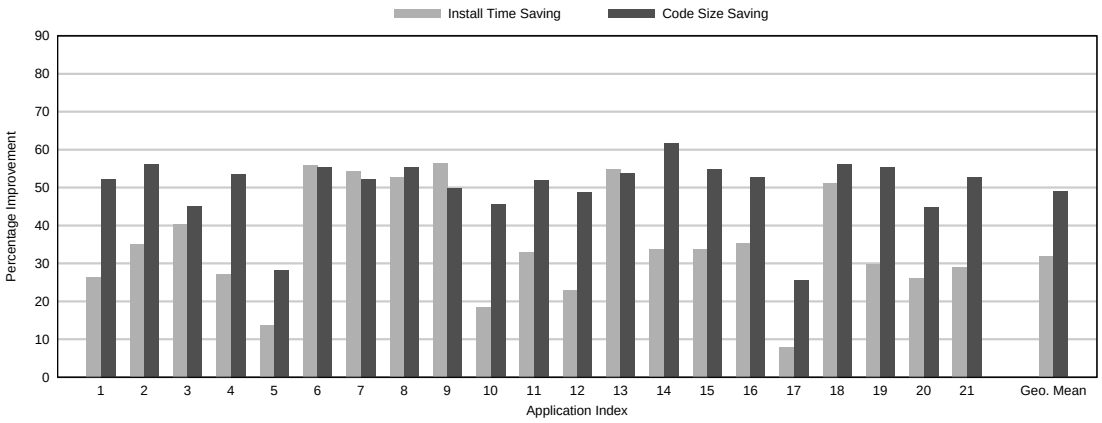


Figure 5.15

Percentage savings made in code-size and installation time when using the *Everyone Inclusive* merging scheme to merge all collected profiles for all applications from the crowd-sourcing experiment. See Table 5.2 for the application associated with each index.

number of methods to compile pointlessly, regardless of the merging scheme used - only 0.4% pointless methods as a percentage of all the methods in the application on average.

From analysis of the performance impact and method inclusion rates arising from these merging schemes, it would seem there is little difference in the conservative and inclusive schemes. Using inclusive over conservative increases performance by 0.01x on average, for an increase of 1-2% extra methods selected for compilation. Nevertheless, it has been shown that it is possible for the majority of users in this experiment to rely on a merged profile instead of their own one. This indicates that this could be a beneficial system for the developers of Android to adopt, in order to remove profiling burden from individual users, but still enjoy the benefits that profiling can provide to code generation, namely selective compilation and focussed optimisation.

5.7.4 Code-size and Installation Time Impact

As a final evaluation the profiles produced by the *Everyone Inclusive* merging scheme will be applied to all of the applications used in the crowd-sourcing experiment, with report given of the observed effects on code-size and installation time. Figure 5.15 shows the percentage savings in code-size and installation time made when using the selective compilation profile rather than compiling all methods in each application. An average saving of 49.1% in code-size, and 31.8% installation time was observed when

using these selective compilation profiles. Compare this to the initial evaluation of the effects of profiling on Android benchmarks, where an average saving of 52% in code-size, and 25.4% in installation time was observed. The code-size savings range from 25.5% to 61.3%, while installation time savings range from 8.8% to 76%. While the 99% threshold for *Everyone Inclusive* selected 5.6% of the methods of an application on average, 99.9% and 99.99% selected 12.3% and 16.3%, respectively, still significantly smaller than compiling the whole application.

5.8 Conclusion

This chapter has presented a method for reducing the compiled code-size and installation time for applications running on the Android operating system, also opening avenues to perform focussed optimisations where most needed. Modern profiling and crowd-sourcing techniques are exploited to determine which code regions are critical to the run time of the application. Once these profiles are collected and merged, the merged profile can be distributed to new users of the application, who can then skip the profiling stage, instead using the merged profile to selectively compile and optimise important code regions. Observations from recorded application profiles show that, on average, users do not see a significant change in their run time when using a merged profile (0.90x-0.92x) instead of their own (0.93x). In other words, a high degree of similarity was observed between profiles when used for such selective compilation. It has been shown that with the best profile merging scheme used to crowd-source the identification of methods that comprise 99% of the runtime for users on average, this technique can provide compiled code-size reduction of 49.1%, and installation time reduction of 31.8% across a number of popular Android applications. Where space may be at a premium, these savings come with a reduction in performance of only 9% across a range of Android benchmarks. A prototype code generator implemented for Dalvik for focussing optimisations on important code regions has shown to provide a 1.46-10.83x speed up in performance for some benchmarks as well. The crowd-sourcing of profiling allows new users to benefit from such speedups immediately. In general, it has been shown that it is perhaps redundant for *all* users in such an environment to perform their own profiling, and the sharing of profiling information with a merging system can prevent every user from having to waste time in a profiling stage, and this could be a useful system for the developers of Android to adopt.

Chapter 6

Differential Fuzzing for PVM Code Generator Testing

The development of a new process virtual machine (PVM) is a complicated task and, like the creation of any piece of complex software, the process is extremely likely to generate bugs. Once the initial development is complete, there is the potential for new features and performance improvements to introduce bugs as well. PVMs will generally have multiple methods of executing a supplied program, as initial development will start with a simple interpreter, and code generators will be added later in order to improve performance. The developers may wish to optimise these methods of execution further or even implement additional ones. As an example, a new code generator called Crankshaft was added to the V8 JavaScript engine a few years after its initial release [MS]. When this happens, how can the developers easily test that new bugs are not being introduced? While scrupulous developers will always aim to cover every corner case, it must be assumed that bugs will inevitably make their way into software.

This chapter will focus on the development of the new ART runtime that is used to execute Android applications, as this is an example of a relatively immature PVM, with new compilers in development, that stands to be used by millions of people in the future. Additionally, this was the PVM discussed in the previous chapter, where a prototype code generator was proposed in order to improve performance. The Android mobile operating system continues to enjoy increasing popularity, recently reaching over a billion active users measured over a 30-day period. It is therefore important to use rigorous testing to find and remove as many defects that may be introduced during development as possible, before the new software is released to the public.

Test suites are a good way to catch bugs during development, but are typically

limited in size, and cannot easily capture all possible interactions between code optimisations in a compiler. The unit test suite of the ART runtime discussed in this chapter currently stands at around 200 tests, taken and extended from the test suite of the original Dalvik VM that Android previously used. Meanwhile, GCC's unit test suite numbers over 100,000 tests, although GCC has admittedly been in development for over 25 years. Nevertheless, how can a test suite get from 200 to 100,000 test cases without waiting for 25 years of bug reports? Additionally, regardless of size, these test cases are usually created when bugs are found, and are only intended to prevent regressions. How do developers test for bugs that they are not yet aware of?

When multiple modes of execution are available within an HLLVM, differential testing can be used, where all modes of execution are given the same program, and are expected to form a consensus about how the program should have been executed. With a "write once, run anywhere" VM that is supposed to provide a single platform and produce programs that execute the same way regardless of the underlying execution platform, this is a sound expectation. Figure 6.1 visualises how the fact that these methods of execution all obey the same VM specification can be exploited to test for bugs in the VM. As there is no single canonical reference specification for the ART VM, an assumption must be made that the conceptually simplest (and so typically slowest) mode can be considered to be the reference mode, as it will not contain many performance-increasing optimisations that have the potential to introduce incorrect behaviour.

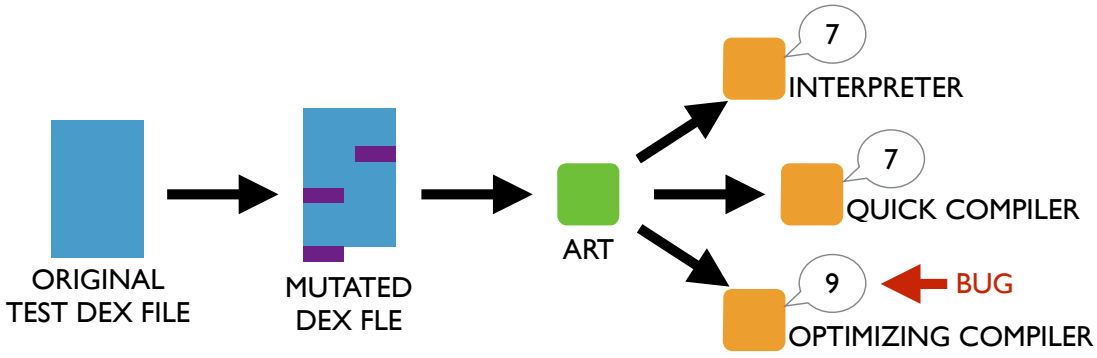


Figure 6.1

An overview of differential fuzz testing for VMs. A DEX program is run using multiple methods of execution, and in this example, one of the methods – the *optimizing* compiler – reveals a defect in the way it has compiled and executed the program.

Another common approach to testing is fuzzing, creating random test cases through either *generative* means, where a new test is produced from no initial seed, or *mu-*

tative, where a test is mutated from a seed program. The combination of fuzzing and differential testing, while applied to domains such as C compilers in the past [YCER11, LAS14], seems a good candidate for testing a VM with the desired rigor that Android's VM would need.

This chapter presents DEXFUZZ, a tool that takes an existing test suite and performs mutative fuzzing in order to produce new tests, in an attempt to explore the boundaries of the VM. These tests are executed by the various methods of execution of ART, such as the production-ready *quick* compiler, the interpreter, and the in-development code generator called the *optimizing* compiler. In the rest of this chapter, these modes of execution will be referred to as "backends" of the VM. Differences in how these backends execute the mutated tests are used in order to identify bugs within in the VM. Some of these bugs are present in the common verification phase of the runtime, while others have been found in the code generation phase. This chapter also presents examples of how these bugs were found, how they came to be, and how they were fixed.

6.1 Motivating Example

The ART VM is a new implementation of the "Dalvik VM" that executes register-based DEX bytecode on Android - see Section 2.5 for background information on Dalvik and ART, and Section 2.5.2 for information about the DEX bytecode format used in ART. Figure 6.2 shows an example of a DEX `const/4` bytecode that loads the constant 8 into the virtual register `v0`. This is followed by an `array-length` instruction that reads into `v1` the length of the array whose reference is stored in `v2`. If this `array-length` bytecode is mutated to instead read from whatever "reference" is stored in `v0`, then ART's bytecode verifier should reject this bytecode, because `v0` does not contain an array reference at this point. However, the verifier previously failed to do this. The compiler backend for ART would then assume that it was safe to produce code that reads from the address stored in `v0`, plus 8, and so the resulting code would allow arbitrary reading of the VM process' address space, if the constant loaded into `v0` was also modified. With the error checking done by the verifier reducing the complexity of the compiler, this requires that verification be sound, which was not always the case, and was a common source of issues.

The Java language aims to provide a "security sandbox" for any program that executes within it. It is still possible to construct pointers and read memory arbitrarily if a

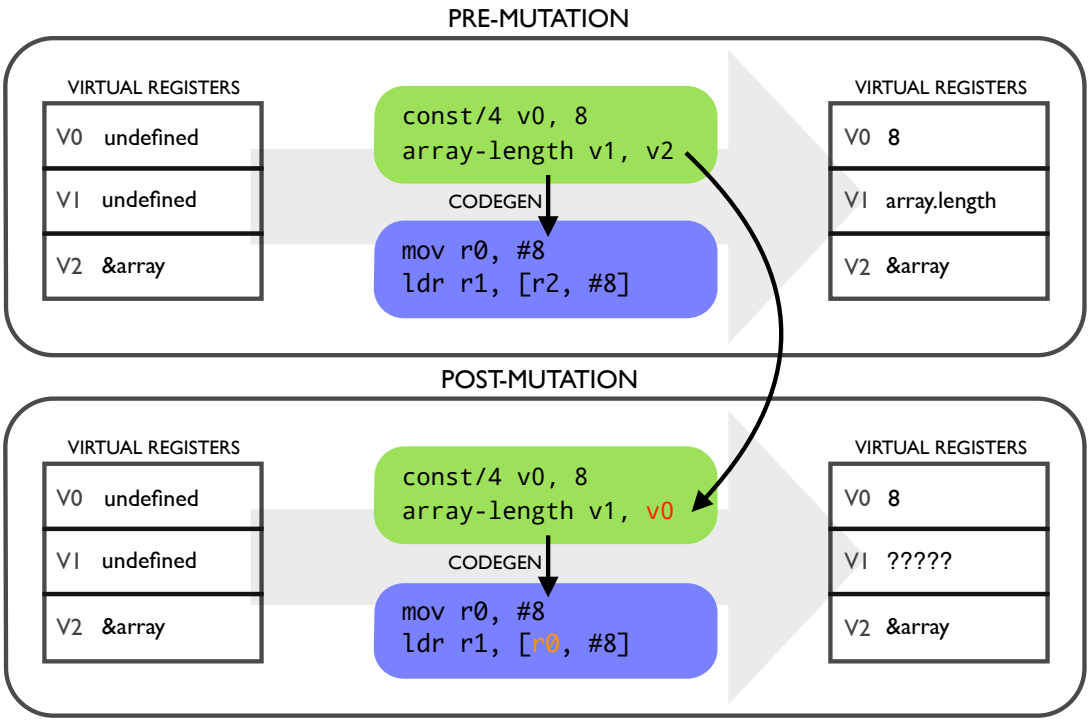


Figure 6.2

An example of a DEX mutation leading to a compilation bug, due to incorrect bytecode verification. The compiler will always produce code that reads from the provided array pointer plus 8 bytes. By mutating the array-length instruction to take v0 as an input, the program now illegally reads from address $8 + 8 = 16$.

mechanism such as the Java Native Interface (JNI) is used, so more trust can be placed in an application being "well-behaved" if it contains no JNI code. Therefore such an application may come under less scrutiny by security analysts, while a piece of crafted DEX code as presented in Figure 6.2 could be used in combination with other exploits to read private data. With the open model of Android, and ease with which applications can be accepted into the various application stores that use Android, there is a significant possibility of encountering accidentally or even maliciously malformed DEX files.

6.2 Code Generator Testing with DexFuzz

Fuzz testing as a concept initially referred to the random creation of input to test the capabilities of a program or Application Programming Interface (API), particularly to test its ability to gracefully handle erroneous input. Fuzz testing is typically divided

into two categories: *generative*, and *mutative*, where generative does not require a seed to generate a new piece of input, while mutative does. Refer to Section 2.7.1 for further information about fuzz testing, and Sections 3.4, 3.5, and 3.6 for related work.

6.2.1 Test Generation

The most basic form of mutative fuzzing is to take some seed input and randomly flip bits in order to produce some new, mutated input. This technique could be applied immediately to produce test programs for the VM, but this unlikely to yield very useful results. The ART VM is supposed to provide a secure sandbox for program execution, and so it must verify any bytecode that it is expected to execute. This verification forms a hierarchy of checks that range from checking that the two input registers to an `add-float` bytecode actually contain float values, to calculating an Adler-32 checksum on the file, and checking this against a provided checksum in the header. Because of these checks, it is extremely likely that any bytecode that is produced through random bitflips will fail some part of verification, if not at least the checksum that protects the entire DEX file.

This claim was tested by producing a simple fuzzer that fuzzed a test program a million different ways for each of seven different fuzzing strategies. In all cases, the DEX file header is untouched, and then the Adler-32 checksum is recalculated after fuzzing, to ensure that other verification checks are able to be executed. In all strategies, the naive fuzzer iterates through every byte after the header, with a chance to mutate each byte. Each strategy is a subset of a set of three possible mutations: randomly change a byte's value, add a random byte, or delete a byte, giving seven overall strategies. The naive fuzzer selected a mutation probability related to the length of the test program, in order to achieve an average mutation rate equal to that of DexFuzz. With 25 test programs taken from the ART test suite used as mutation seeds, this fuzzer produced a rate of roughly **1 in 100,000** programs that showed divergent behaviour between different ART backends, when tested with version 5.1.0r3 of Android.

For comparison, DexFuzz was also used to test this version of Android, producing a rate of roughly **1 in 10,000** programs that showed divergent behaviour. Focussed mutation can be used to more intensively test the code generators of ART. Additionally, such a naive approach as presented here is extremely unlikely to insert or delete instructions, or insert even more complex constructs such as new methods or classes.

Typical generative approaches to fuzz testing for compilers have focussed on pro-

ducing valid programs in the relevant source language. In the case of Csmith [YCER11], this desire for valid program generation stemmed from the presence of undefined behaviour with certain sequences of C code. With a bytecode format like DEX, the concept of undefined behaviour does not exist. Either a sequence of bytecode has a defined effect according to the VM specification, or the sequence should not pass verification. Therefore, this test generation of this bytecode can ignore the threat of undefined behaviour. Furthermore, it should actually aim to generate some invalid programs, in a bid to ensure the verifier of ART is robust. Finally, prior generative approaches have found it difficult to produce a system that fully utilises all features of the language that they are generating code for [YCER11]. Using a set of programs from a test suite that should at the very least contain all features of the language as a basis for seeds for producing new tests seems a better approach.

None of the previously described strategies are ideal for fuzzing bytecode, so in DEXFUZZ an approach was taken that applies some degree of intelligence to the use of mutations.

6.2.2 Mutating DEX Bytecode

The fuzzing process in this chapter can be characterised as *domain-aware* mutation of bytecode. Figure 6.3 provides an example of some mutations performed within a single method. The intention is that the application of a small number of relatively simple, but domain-aware, mutations can lead to both programs that verify, but also highlight bugs in the ART VM through unexpected handling of mutated bytecode.

Algorithm 1 shows pseudo code for the mutation process. DEXFUZZ will parse a program's DEX file and produce a set of mutatable code items, each of which represent a method within the program. In order to increase the likelihood that the program will successfully verify, DEXFUZZ limits the number of methods that will be mutated to a random value between 2 and 10. A random subset of the available mutatable methods are then selected using this value, and each in turn is mutated. Each method has between 1 and 3 mutations applied to it by default, although this value can be configured. DEXFUZZ randomly selects from the list of available mutators that are listed in Table 6.1. Because each mutator checks and reports if it is able to mutate the given method (for example, *BranchShifter* will only mutate methods containing branches), this process repeats until the desired number of mutations has been applied, or a maximum number of attempts is reached. After all mutations have been applied

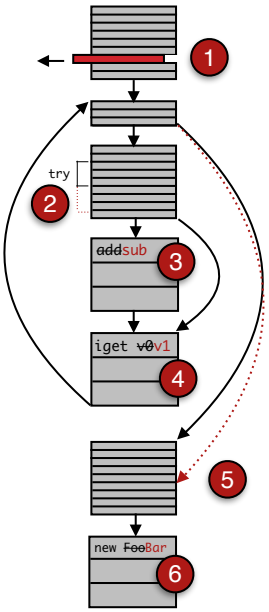


Figure 6.3

Some examples of DEX bytecode mutations applied to a method formed of six basic blocks. In ①, an instruction is deleted. In ②, the boundaries of a try block are expanded. In ③, an add operation is replaced with a subtract operation, operating on the same types and virtual registers. In ④, the virtual register specified by an `iget` instruction is changed. In ⑤, the branch from the 2nd basic block to the 6th is shifted forward by a few instructions, splitting this basic block, and rendering some code unreachable. Finally, in ⑥, a new instance of an object of class Bar is created instead of Foo.

to the correct number of methods, a new, mutated DEX file is produced by DEXFUZZ.

Why does this mutation strategy improve over simple bit-flipping fuzzing? The mutation process can ensure that branches always remain within the code area, type or field pool indices fall within the range of types of fields available in the DEX file, and a specified virtual register in an instruction is actually allocated within the given method. These are all simple structural constraints that the verifier can easily check for, in comparison to complex constraints like checking that a virtual register contains the correct type during all of its uses. For example, the type checking constraints of the JVM are complicated enough have been specified in Prolog [LYBB], although ART does not perform its own type checking with Prolog.

Why does it improve over generative or mutative fuzzing that only produces legal programs? As stated previously, while some verifier constraints are simple, others are more complicated, such as checking the types of values in virtual registers are

```
parse DEX file into mutatable methods;
Methods = set of mutatable methods;
Mutators = set of mutators;
// Select a number of methods to mutate
methodCount = random(2,10);
for i in methodCount do
    // Select a random method
    method = selectRandom(Methods);
    // Select a number of mutations to apply to the method
    mutationCount = random(1,3);
    exitEarly = False;
    // Attempt to mutate until enough mutations have been
       performed, or a threshold is reached
    while mutationCount > 0 and !exitEarly do
        // Select a random mutator to use
        mutator = selectRandom(Mutators);
        // Check if the mutator can mutate this method
        if mutator.canMutate(method) then
            // It can, so mutate the method
            method = mutator.applyMutation(method);
            mutationCount = mutationCount - 1;
        else
            if reached mutation attempt threshold then
                // Give up mutating this method, it may not be
                   amenable to mutation
                exitEarly = True;
            end
        end
    end
end
write methods and mutated methods into new DEX file;
```

Algorithm 1: Mutative fuzzing process for DEX files.

valid wherever they are used. It is important to test these constraints, and so the generation process does not provide any guarantee that a mutation will leave the bytecode in a completely legal state. A few of the bugs described in this chapter that were found in ART were actually found in the verifier, and this was made possible by allowing the production of invalid programs.

6.2.3 Differential Testing of Multiple VM Backends

Test suite programs are typically used to test for VM correctness by executing the program, and comparing its output against what it is expected to produce. However, with the generation of new test programs, the expected output is unknown. Therefore, differential testing can rely on the fact that the multiple methods of execution available to a VM are intended to produce the same result. This is certainly one of the desired properties of many *write-once-run-anywhere* VMs, particularly ones related to executing Java code, therefore the same property holding of the execution methods can be exploited to test these mutated programs.

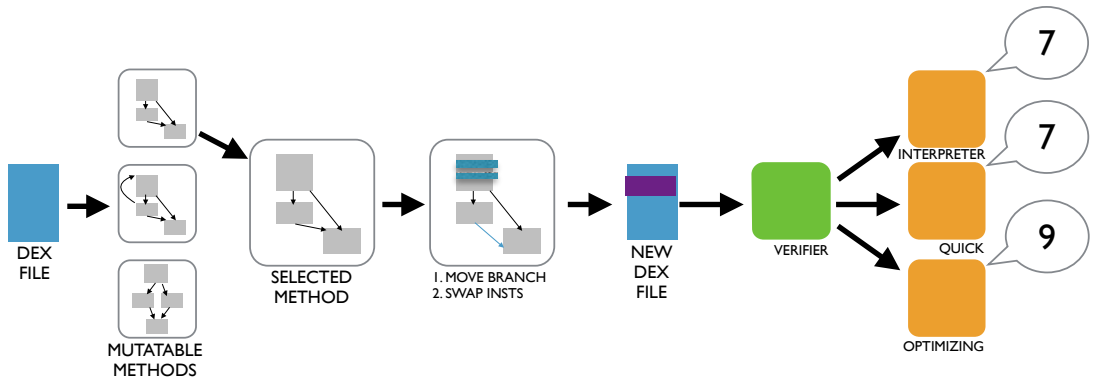


Figure 6.4

A larger overview of combining domain-aware binary fuzzing and differential testing for testing code generators in VMs. The DEX seed program is parsed into its constituent code items, and a subset of these are chosen for mutation. Random mutations are applied to these code items, and a new, mutated DEX test case is produced. This test case is then run using multiple methods of execution, and in this example, one of the methods – the *optimizing* compiler – reveals a defect in the way it has compiled and executed the program, which may have not been visible prior to the seed case's mutation.

Figure 6.4 visualises the differential testing strategy. Mutated programs can be

executed using the *quick* and *optimizing* compilers, as well as the interpreter. Additionally, mutated programs can be run on both 32 and 64-bit variants of the VM, when executed in an environment where this is possible, such as a processor that implements the ARMv8 ISA. When all tested methods of execution fail to reach a consensus about the output of the program, it is highly likely that one of them has encountered a bug.

While the code generators of a "write once, run anywhere" HLLVM are expected to produce the same result by following the single specification for the VM, with well-defined effects for all operations, there are a few corner cases where it is acceptable for different backends, or even the same backend, to produce different results from execution to execution of the program. This is due to situations like the operation of multi-threaded programs, where the VM provides no guarantee on the visibility of memory operations performed by multiple threads. Indeed, as some of the test programs used as seeds for mutation are multi-threaded, it is expected that if they are mutated in a way that causes them to crash, then they may crash in ways which are non-deterministic, despite their pre-mutated output being deterministic. For this reason, whenever a divergence between outputs for different backends is encountered, the same program is run multiple times with a single backend, to determine if its output diverges from itself. If so, these divergences are tagged as such, and are not counted as "true" divergences. Another corner case is floating point operations, where a VM running on different ISAs may produce different results, depending on, for example, the way a particular architecture rounds certain values - this particular case must also be identified and not considered a true divergence until further tested.

6.3 Evaluation

DEXFUZZ has been used to find defects in the in-development version of ART when running on an ARMv8 platform, in both AArch32 and AArch64 execution modes. The version of DEXFUZZ used to find bugs is designed to send programs to the target platform via the Android Debug Bridge (ADB) for execution. Because of the overhead involved in sending programs to the platform, all mutated programs are verified on the host machine first using the DEX compiler, `dex2oat`, that has been compiled for the host machine, and only programs that pass this initial test are uploaded. The host and target versions of `dex2oat` verify all DEX files the same way.

6.3.1 Finding Bugs

This section presents some examples of the kind of code generator and verifier bugs that have been found in the ART VM in development in the Android Open Source Project (AOSP). An explanation is provided of both how they were discovered by DEXFUZZ and what lead to their presence. While the primary goal of this test generation was to find bugs in the code generators of ART, some reported bugs were found in the verifier, before code generation took place. Although ART does not have two verifiers to perform differential testing with, the use of such testing can still lead to bugs being found in the verifier. This typically occurred when a hole in the verifier would lead to the *quick* compiler and interpreter producing different results from this falsely verified code. This was typically because the *quick* compiler would make assumptions about the code it could produce based on properties of the DEX bytecode that the verifier had allegedly proven.

6.3.1.1 Reading the instance field of a non-reference bearing virtual register

In DEX bytecode, there exist instructions such as `iget`, that allow the reading and writing of instance fields of objects. The verifier of ART checks that the data currently in input virtual registers for a given bytecode have the correct types, with respect to the types the bytecode operates on.

The first section of DEX bytecode (Listing 6.1; Lines 2-5) represents the increment of the *counter* field of a MyClass object whose reference resides in `v1`. During bytecode mutation, the *VirtualRegisterChanger* mutation changed the `v1` on line 3 to `v2` on line 9, producing a sequence of DEX bytecode that would be incorrectly translated by the code generator in ART. The verifier accepted this bytecode, despite `iget` now attempting to read the instance field of the constant 1 in `v2` rather than any reference to an object. The *quick* compiler would then assume that it was safe to emit native code that loads from the address stored in the 2nd input to `iget`, plus the offset of counter. If the constant loaded into `v2` initially was modified, then memory could be arbitrarily read from the process' address space.

6.3.1.2 Mixed float/int constant usage causes arguments to be passed incorrectly

While many arithmetic and logical operations in DEX bytecode are aware of the types of data they are using, other bytecodes such as data movement and constant loading instructions are not.

```
1  // Before mutation
2  const/4 v2, 1
3  iget v0, v1, MyClass.counter
4  add-int/2addr v0, v2
5  iput v0, v1, MyClass.counter
6
7  // After mutation
8  const/4 v2, 1
9  iget v0, v2, MyClass.counter
10 add-int/2addr v0, v2
11 iput v0, v1, MyClass.counter
```

Listing 6.1

A demonstration of DEX bytecode mutation that produced incorrectly validated code, leading to illegally generated native code. This highlighted a mechanism for arbitrarily reading memory addresses in the process' address space from within Java code.

```
1  // Before Mutation
2  const v0, 1
3  const v1, 1.0
4  invoke-static {v0}, void Main.doInteger(int)
5  invoke-static {v1}, void Main.doFloat(float)
6
7  // After Mutation
8  const v0, 1
9  const v1, 1.0
10 invoke-static {v1}, void Main.doInteger(int)
11 invoke-static {v1}, void Main.doFloat(float)
```

Listing 6.2

A demonstration of DEX bytecode mutation that produced code that still correctly validated, but was handled incorrectly by the code generators of ART.

Consider the code sample in lines 2-5 of Listing 6.2. This code loads two constants, 1 and 1.0, and passes them to methods that take integer and float arguments, respectively. In DEX bytecode, the `const` instruction is typeless, and just loads a bit pattern into a virtual register. As such, the load of the float value 1.0, and the inte-

ger 0x3f800000 (the IEEE 754 representation of 1.0) are actually indistinguishable in DEX bytecode. Code generation therefore requires that type inference be performed - the compiler must look at the uses of loaded constants to determine their types. Because `v1` is used as a float at line 5, the compiler infers that it must be a float. If the VirtualRegisterChanger mutation changes the `v0` register on line 4 to `v1` on line 10, then the constant 1.0 in `v1` is now being passed to both methods. Type inference will determine that `v1` contains both a float and an integer value, and this is legal according to DEX bytecode specifications.

This lead to a miscompilation in the *quick* backend of ART, however, because the generated code that passes arguments to the method using the appropriate calling convention would rely on this type-inference information, resulting in problems in any environment where integer and floating point (FP) values are passed in separate physical registers. Because the mutated register `v1` had been marked as a float type at line 10, the emitted native code would pass this argument in a FP physical register, when the callee method would expect to find its integer argument in a core (non-FP) register. The callee method would then use whatever data happened to be in that core register, and execute incorrectly. A patch was submitted to the AOSP that changes the calling convention code generator to consider the specified types of the callee method instead of just the type-inference information.

6.3.1.3 Compiler crash from an unreachable check-cast

Verification of DEX bytecode is only performed on reachable code - code flow analysis takes place during verification, reporting errors like execution reaching the end without an explicit `return` instruction. It is not automatically illegal to have unreachable bytecode, however.

Consider the following DEX bytecode in Listing 6.3. Before mutation, this bytecode checks that the provided argument in virtual register `v0` is a reference to an object that can be cast to a `java.lang.String`, and then returns. The first mutation applied to this bytecode sequence is the *PoolIndexChanger* mutation, that changes the checked type in `check-cast` to a `void` type. This instruction is now illegal when considered alone - it checks that it is acceptable to cast the value in `v0` to a `void` type, or throw a `ClassCastException` otherwise. In the DEX bytecode specification, `void` is never a legal type to cast to, but if a *InstructionSwapper* mutation then swaps the `return-void` and the `check-cast`, then the `check-cast` is no longer reachable. For this reason, it is then never rejected by the verifier.

```
1  // Before Mutation
2  check-cast v0, Ljava/lang/String;
3  return-void
4
5  // After PoolIndexChanger Mutation
6  check-cast v0, V
7  return-void
8
9  // After InstructionSwapper Mutation
10 return-void
11 check-cast v0, V
```

Listing 6.3

A demonstration of two DEX bytecode mutations that produced code that would lead to a compiler crash in ART.

In most situations this would not lead to any problems, since the code generator itself will only generate code that is reachable. However, the *quick* compiler performed a later optimisation where it scanned across the entire DEX bytecode it was currently translating, to find `check-cast` instructions that it could elide because they would never throw the exception. This optimisation would reach the unreachable `check-cast`, and then cause the compiler to crash when an assertion was broken. A patch was submitted to the AOSP that makes this scan only consider instructions that were flagged as visited during verification.

6.3.2 Characterisation of mutations

This chapter has presented 11 mutations in Table 6.1 that can be applied to DEX bytecode that aim to highlight bugs within the ART VM. As the choice of these mutations was primarily driven by intuition, this section attempts to characterise which mutations are more responsible for creating verified programs that still produce program divergences. Because DEXFUZZ was already used to find and patch a number of defects within ART, in this section all test programs were executed with the experimental version of ART released in Android KitKat (v4.4), where it is more certain that there were bug-indicative divergences to be found. A single test program was used as a seed in these experiments. This test program was confirmed to work in the chosen version

of ART prior to mutation, and was capable of being mutated by all available mutations. This was done in order to be sure the characteristics of each mutation was being evaluated.

DEXFUZZ was run 11 times, each time enabling only one of the available DEX mutations. For each mutation, 2,000 attempts to produce candidate programs were made. For each mutation, these results indicate how many verified, how many failed to mutate at all, how many passed verification and ran successfully on the platform without divergence, how many crashed, and how many produced divergent behaviour that may indicate the presence of a defect. The differential testing used in these experiments compares only the output of the interpreter and the *quick* compiler in this older version of ART.

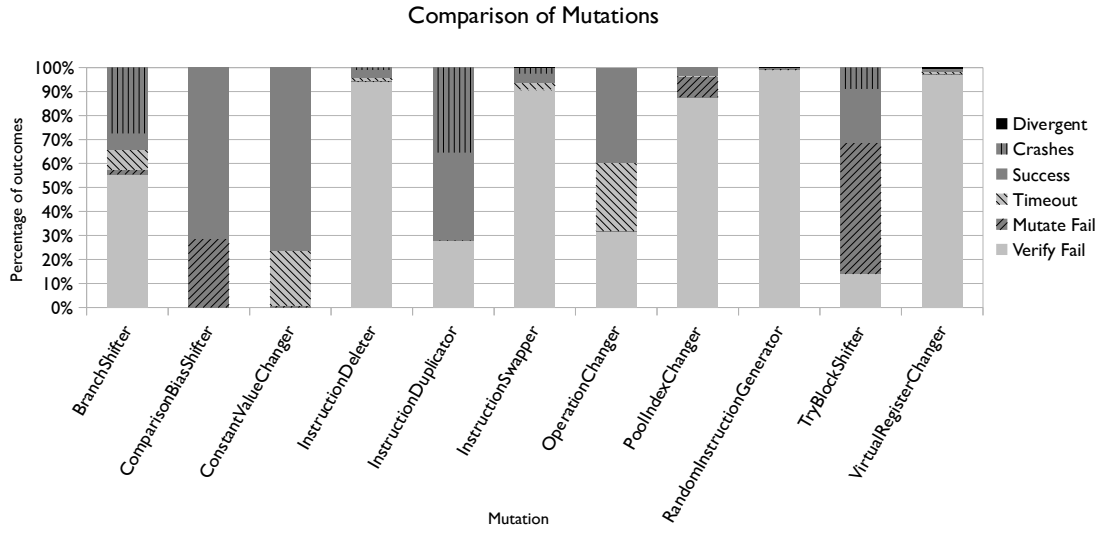


Figure 6.5

Breakdown of resulting programs created by individual bytecode mutations. In each case, DEXFUZZ has been run for 2,000 iterations, and each bar presents what percentage of these programs failed to be mutated, failed to verify, timed out, crashed, were successful, or produced program divergence, when run on the experimental version of ART available in Android KitKat (v4.4).

Figure 6.5 and Table 6.2 present the breakdown of execution outcomes for the different mutations performed by DEXFUZZ. Some are clearly more successful at finding divergent behaviour than others, with the *VirtualRegisterChanger* most responsible for producing divergences. 97% of the programs generated by this mutation failed verification however, so this clearly demonstrates the necessity of allowing the generation of potentially illegal programs in order to find divergent programs. The other muta-

tions that found divergences were *RandomInstructionGenerator*, and *InstructionSwapper*, which also generated a large percentage of programs that failed to verify, at 99% and 91% respectively. Some of these divergent programs were indications of the bug described in Section 6.3.1.1.

Many mutations were responsible for producing crashes in ART, with the *InstructionDuplicator* and *BranchShifter* mutations most likely to lead to a crash. From analysis of the backtraces of these crashes, it appeared that ART had problems with SSA transformation during the compilation stage, which may have been exacerbated by the variety of unique control flow graphs produced by the *BranchShifter* mutation in particular.

Looking at each mutation in turn, *BranchShifter* produced invalid programs for roughly half of its attempts, and was otherwise responsible for a large number of crashes. The *ComparisonBiasShifter* mutation either produced successful programs, or failed to mutate the program at all, due to maximum attempt thresholds mentioned in the description of DEXFUZZ's operation. *ConstantValueChanger* never produced any invalid programs, but created a number of long-running programs, presumably when changes to constants meant loops ran longer than the permitted timeout. *InstructionDeleter* was another mutation that led to a large percentage of invalid programs, but was also able to find a few crashes.

InstructionDuplicator had a roughly three-way split in terms of crashing, successful and invalid programs produced, with no divergences found, while *InstructionSwapper* found a few crashes, but mainly produced invalid programs. Despite *ConstantValueChanger* only changing an operation with a particular set of input types to an operation with the same set of input types, 30% of its programs failed to verify, contrary to expectation. This was traced to bytecodes that operated on 64-bit integers like `add-long` being changed to shift operations on longs, such as `ushr-long`, where the second input, the shift amount, is actually expected to be a 32-bit value rather than 64-bit.

PoolIndexChanger resulted in a lot of invalid programs, as for example, method invocations were often changed to call new methods with incompatible argument types. As might be expected, *RandomInstructionGenerator* lead to the largest percentage of invalid programs, but did also find crashes and divergences. The *TryBlockShifter* mutation found a large number of crashes too, which lends weight to the idea that control flow issues were a major source of crashes in the compiler at the time. Finally, as stated above, *VirtualRegisterChanger* was the most successful mutation, finding the largest

number of divergences, but failed to find any crashes at all.

Type	**Description and Example**
BranchShifter	Change the target of a branch by a small delta. *if-eqz v0, +05 → if-eqz v0,* ***+07***
ComparisonBiasChanger	Change the bias of a comparison operation. *cmpg-double v0, v1 → cmp****l****-double v0, v1*
ConstantValueChanger	Change a constant used for a constant load, or an immediate in an arithmetic operation. *add-int/lit8 v0, v1, 7 → add-int/lit8 v0, v1,* ***18***
InstructionDeleter	Delete a bytecode. *const v0, 2; const v1, 4 → const v0, 2*
InstructionDuplicator	Duplicate a bytecode. *or-int v0, v0, v2 → or-int v0, v0, v2;* ***or-int v0, v0, v2***
InstructionSwapper	Swap two bytecodes. *monitor-enter v3; move v4, v2 →* ***move v4, v2; monitor-enter v3***
OperationChanger	Change the arithmetic or logical operation performed by a bytecode, preserving types. *add-int/2addr v0, v1 →* ***div****-int/2addr v0, v1*
PoolIndexChanger	Change the index into a type/method/field pool used by a bytecode. *new-instance v2, type@007 → new-instance v2,* ***type@023***
RandomInstructionGenerator	Generate a random bytecode, with random operands, and insert it into a random location within the method. *nop; nop → nop;* ***throw v6****; nop*
TryBlockShifter	Move the boundaries of a try block. *try (move v0, v1;) move v2, v6 → try (move v0, v1; move v2, v6)*
VirtualRegisterChanger	Change one of the virtual registers specified by a bytecode. *iget v1, v2, field@015 → iget v1,* ***v0****, field@015*

Table 6.1

A complete listing of all the DEX mutations currently performed by DEXFUZZ.

Mutation	Verify Fail	Mutate Fail	Timeout	Success	Crashes	Divergent
BranchShifter	1110	37	167	137	549	0
ComparisonBiasShifter	0	572	0	1428	0	0
ConstantValueChanger	0	13	459	1528	0	0
InstructionDeleter	1880	8	32	60	20	0
InstructionDuplicator	558	5	0	728	709	0
InstructionSwapper	1811	7	52	79	50	1
OperationChanger	636	2	571	791	0	0
PoolIndexChanger	1753	169	7	71	0	0
RandomInstructionGenerator	1979	10	2	5	2	2
TryBlockShifter	279	1098	0	444	179	0
VirtualRegisterChanger	1944	2	18	25	0	11

Table 6.2

Absolute counts of execution outcomes for each mutation shown in Figure 6.5.

Execution Outcome	Observed
Verification Failure	9249
Timeout	129
Mutation Failure	28
Crashing	314
Successful	257
Divergent	23
Iteration Total	10000

Table 6.3

Breakdown of execution outcomes when running DEXFUZZ with only the top five most successful mutations enabled: VirtualRegisterChanger, RandomInstructionGenerator, InstructionSwapper, InstructionDuplicator, and BranchShifter.

6.3.3 Effect of multiple mutations

The previous section shows that some mutations are capable of finding divergent behaviour on their own, and indeed some have been individually responsible for many of the bugs that have been discovered in ART using dexfuzz so far. With the use of these small mutations there is a hope that a combination of these could perhaps find other bugs that individual mutations could not. The mutations tested in the previous experiment were ranked first by their ability to find divergent behaviour, and then by their ability to produce crashing programs. The top five of these mutations—*VirtualRegisterChanger*, *RandomInstructionGenerator*, *InstructionSwapper*, *InstructionDuplicator*, and *BranchShifter*—were selected, DEXFUZZ was run with all five enabled, each with equal chance of being selected. This time DEXFUZZ was run for 10,000 iterations, and each divergence found was checked to see if a combination of mutations was responsible for its existence.

Table 6.3 presents the breakdown of execution outcomes when running the top 5 most successful mutations. As would be expected, the largest percentage of programs produced were invalid, since three of the top five mutations had extremely high invalid production rates. Like those mutations, divergences were also found. Each divergence found was checked to see if it was produced by a combination of mutations. This was found to not be the case - in all divergent programs, a single instance of mutation created the divergent behaviour. It is easy to imagine a set of mutations that may

result in divergence together - for example, creating a random legal `array-length` instruction, and then swapping it to a location where it becomes illegal and finds the bug mentioned in Section 6.1, where the swap without the `array-length` instruction would have also been legal. However, these results have failed to find such occurrences in practice thus far, and so further investigation is required.

6.3.4 Comparison with other mutative fuzzing strategies

While Section 6.2 showed that a naive fuzzing scheme was not ideal for fuzzing a strictly verified bytecode format, and while there exist no available fuzzing systems that specifically fuzz DEX programs for ART or Dalvik, there do exist target-program-agnostic fuzzers that attempt to intelligently perform mutative fuzzing. One such fuzzer is American Fuzzy Lop (AFL) [AFL], a system that fuzzes inputs for an instrumented binary program, observes the execution paths of the instrumented binary, and attempts to intelligently fuzz new inputs using basic bit-level flipping, such that it will aim to fuzz interesting programs further to explore new execution paths. It aims to maximise the number of unique execution paths it finds, as well as the number of crashes.

To compare this fuzzer against DEXFUZZ, a host (x86-64) version of ART was built with the compiler wrapper that AFL provides. This wrapper inserts the instrumentation required for any program to be to fuzzed properly with AFL. The header verification of ART was disabled when being used with AFL for this experiment as well, in order to give AFL a chance to explore more than the checksum calculation code of ART, as this is a known limitation of AFL. The AFL fuzzer was run in its instrumented fuzzing mode for 24 hours, with a selection of 16 ART test programs as seeds, to see how many crashes and hangs it found, as well as programs with unique paths. All of these programs with unique paths were run through DEXFUZZ's standalone differential tester to see if any of the programs produced divergence between the interpreter's and the *quick* compiler's output. DEXFUZZ was then run for 24 hours with the same seed programs, to see how many crashes, hangs, and divergent programs it produced. In this experiment, DEXFUZZ's mutations have all been set to have an equal chance of being triggered.

Figure 6.6 shows how DEXFUZZ's ability to find divergent programs and crashes compares with AFL's. While it is clear that AFL is capable of producing a significantly larger number of programs within 24 hours, this is explained by choice of implemen-

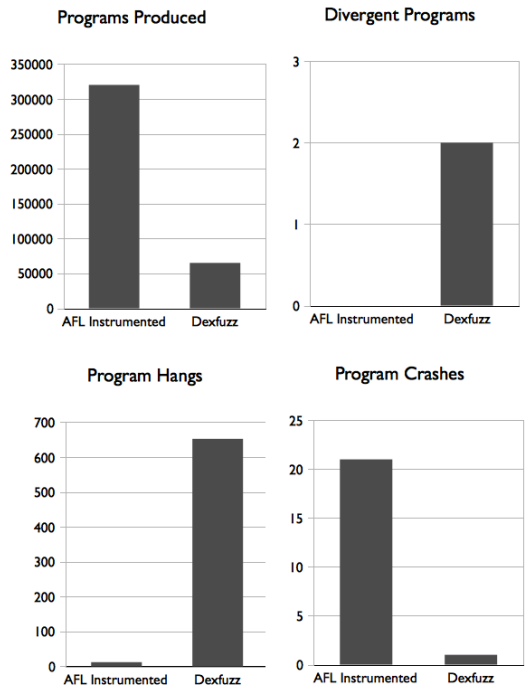


Figure 6.6

Comparison of success rates of AFL and DEXFUZZ after a 24-hour period of fuzzing.

tation language (DEXFUZZ is written in Java, while AFL is written in C), and that very little time has currently been spent optimising DEXFUZZ, as most of the testing time in the tool is dominated by transferring programs to the test platform, and waiting for them to finish executing. Despite a comparatively small quantity of tests with respect to AFL, DEXFUZZ managed to find two programs that demonstrate divergent behaviour in ART while none of the programs that AFL marked as having unique execution paths demonstrated divergent behaviour. Indeed, of all the programs that AFL produced, none of them passed the verification stage of ART either. For all the hanging programs that AFL reported, none of these passed verification, and may have been an artefact of AFL's default timeout value. Despite having a more lenient timeout value, DEXFUZZ was able to find many more hanging programs in comparison to AFL, due to it actually producing programs that pass verification. Where AFL does excel in comparison to DEXFUZZ however, is its ability to find crashing behaviour. AFL has probably thoroughly tested the structural verification of ART, while DEXFUZZ's fuzzing is weaker in this regard. In general, however, these results demonstrate that the use of domain-aware mutation is useful in discovering divergent behaviour.

6.4 Impact

This chapter has presented DEXFUZZ, a differential mutative fuzz testing tool for finding bugs in ART, and presented a number of bugs discovered in the AOSP version of ART. While characterising the mutations of this tool, information has also been shown about divergent programs that were found in the experimental version of ART released in Android 4.4, that would have proven useful had DEXFUZZ been available at the time. Furthermore, this section will discuss some of the impact on ART from the development of this testing tool.

6.4.1 Improvements to ART Test Suite

Previously, the ART test suite consisted of a number of Java programs, that were compiled to DEX bytecode and executed. The mutative differential fuzz testing shown in this chapter has uncovered bugs that are revealed using programs that cannot be produced directly from Java code, but actually require manipulation of already existing DEX bytecode. Since the point where some of these bugs uncovered by DEXFUZZ were fixed by submitting patches to AOSP, the developers of ART have added a new suite of tests to the existing ART suite, that allows for tests using mutated DEX bytecode to be checked. Now tests can be described using the *smali* DEX bytecode assembly format [Sma], which allows developers to test for regressions in bugs found by DEXFUZZ. These patches to AOSP were evidently the catalyst for the creation of these tests.

6.4.2 Potential for future backends

Currently DEXFUZZ has only been put into practice with the *quick* compiler and interpreter of ART, as these are the only mature backends currently available. From looking at the AOSP code base, it is clear that significant work is going towards producing a new backend, the *optimizing* compiler. The *quick* backend treats DEX bytecode compilation in a simple manner, mapping directly from bytecodes to sequences of native code, and then performing a few simple code optimisations on this native code. By comparison, the *optimizing* compiler focuses on much more significant optimisation, of the DEX bytecode itself before lowering it to native code, using more sophisticated techniques found in modern JIT compilers. With such complexity comes the potential for bugs however, and as DEXFUZZ has demonstrated its ability to find bugs in older

versions of ART, it is expected that use of this tool will find bugs in the *optimizing* compiler at an early stage.

6.4.3 Contribution to AOSP

Proof of the quality of any bug-hunting tool is naturally in how many bugs it finds. Since development of DEXFUZZ has begun, a number of bugs have been found in ART, and patches for these bugs have been submitted to the AOSP tree. Most of these patches have been accepted, showing that DEXFUZZ finds bugs that the developers are interested in fixing. DEXFUZZ itself has also been successfully submitted as a patch and merged into the AOSP code base, clearly indicating that DEXFUZZ is considered valuable to the future development of ART. DEXFUZZ is currently being integrated into the continuous testing system for ART, and test cases produced by DEXFUZZ have been credited for a number of patches to the ART code base.[1] [2] [3]

6.5 Conclusion

This chapter has presented a tool to augment the testing of the new ART HLLVM in Android, called DEXFUZZ, primarily aimed for testing the code generators of ART, but consequently also thoroughly testing the bytecode verifier. This is done by exploiting the fact that multiple methods of execution in a HLLVM follow the same VM specification on how an input program might be executed in most cases. This chapter has described a number of bugs that were found and fixed using this tool in the in-development AOSP version of ART, to demonstrate the utility of this tool to prevent bugs being released in future versions of the software. DEXFUZZ is built upon the combination of domain-aware binary fuzzing and differential testing, and evaluation has been made of what sorts of binary mutations that can typically be applied to DEX files lead to divergent behaviour in the experimental version of ART found in Android KitKat, to confirm which mutations were more useful for finding bugs. This found that mutating specified virtual registers, insertion of random instructions, and swapping instructions, while increasing the amount of invalid programs produced, are the mutations most likely to lead to divergent behaviour, and should be useful initial

[1] `https://android-review.googlesource.com/#/c/146807/`
[2] `https://android-review.googlesource.com/#/c/109814/`
[3] `https://android-review.googlesource.com/#/c/148526/`

mutations to use if future VM designers wish to use this testing approach for testing their code generators.

Chapter 7

Conclusion

This thesis has investigated how wasted effort in the code generation that powers process virtual machines can be mitigated through the exploitation of information sharing. Applications are investigated where code regions are being JIT compiled, profiling to direct code generation is taking place, and code generators are being tested. These applications are shown to be useful in improving the performance and correctness of pervasive virtual machines that are found in mobile, personal desktop, and enterprise environments today. Chapter 4 has presented a novel approach to code generation performed within a multithreaded dynamic binary translator, allowing generated code regions to be shared between multiple execution threads. Chapter 5 has discussed the application of profiling to direct selective compilation and focussed optimisation within the VM used on the Android mobile OS, indicating that new users can be spared the burden of profiling once a generic profile has been formed by the information sharing efforts of others. Finally, Chapter 6 has presented a technique for exploiting the shared semantics of multiple execution engines in a VM in order to improve the testing of the VM used in Android.

7.1 Contributions

The goal of this thesis was to reduce the undertaking of redundant work and exploit information sharing in order to improve the performance and correctness of process virtual machines that perform code generation. This exploitation took place in the generation of code itself, but also in a number of areas surrounding code generation, namely profiling and testing. This section summarises the main contributions of this thesis in all of these areas.

7.1.1 Sharing Code Regions in a Multithreaded DBT System

In Chapter 4, an approach to generating JIT-compiled code regions that can be used within a dynamic binary translator by multiple threads wishing to execute the same region of a binary was presented. This *thread-agnostic* code generation technique was shown to perform as well as *thread-specific* code generation on x86 platforms, despite intuition that would perhaps suggest otherwise. Additionally, a technique for sharing these translated regions between threads within the ARCSIM dynamic binary translator was shown. This improves over prior approaches that either globally record code flow for compilation or produce translations usable by only a single thread. Analysis showed that an average of 76% of all regions can be shared in the Splash-2 benchmark suite when 128 threads are used for simulation. Evaluation was made of the performance improvements obtained when the proposed system was used to execute these benchmarks with 128 threads, showing an overall performance improvement of 1.4x.

7.1.2 Sharing Profiles for Selective and Focussed Code Generation for PVMs

In Chapter 5, an investigation was made into how multiple users within the Android ecosystem can share profiling information to intelligently compile Android applications to native code on their devices, removing the burden of profiling for many users. First, an evaluation of how profile-based selective compilation can reduce the code-size and installation times for standard Android benchmarks without unduly sacrificing performance was presented, showing that for a performance loss of 9%, code-size can be reduced by 52% and installation time by 25.4%. Preliminary investigations into the use of a more powerful code generator for focussed optimisation of these benchmarks found speedups between 1.46x and 10.83x.

This was followed by an investigation into into how different profile merging schemes can affect the estimated performance for Android users using popular applications when they selectively compile an application with a merged profile that they did not participate in the creation of, removing the burden of profiling for many users. It showed that if a personalised selective compilation profile provided a reduction in performance of 7%, a profile produced by the best merging scheme only increases this to 10% at most. The analysis was performed using profiles gathered from real users' use of these applications. Finally, an evaluation of how these merged profiles can affect the installation time and code-size for popular Android applications was presented,

showing a reduction of 31.8% and 49.9% respectively, which should be beneficial to low-end devices where compilation of the whole application is infeasible.

7.1.3 Differential Fuzzing for PVM Code Generator Testing

In Chapter 6, an approach was presented where binary fuzzing is combined with differential testing to exploit the presence of multiple VM code execution engines in order to improve the testing of the VM used in Android. This approach relies on discrepancies between the way these engines execute a program to identify bugs. The use of domain-aware binary fuzzing to generate new test cases was also described, a comparison of this approach to both naive and intelligent mutative fuzzing systems was also made. The binary mutations that were used to find bugs in Android's VM were characterised, with identification of which mutations could be beneficial when applying this technique to the testing of other VMs.

7.2 Critical Analysis and Future Work

This thesis has investigated various applications of information sharing to improve code generation in process virtual machines. This section provides a critical analysis of this work, as well as potential research directions to take it into in the future.

7.2.1 Sharing Code Regions in a Multithreaded DBT System

This thesis has presented an extension to the ARCSIM instruction set simulator using dynamic binary translation that supports the generation of code that can be shared by parallel threads of execution, where each is responsible for a single core that is being simulated by the DBT. It is unfortunately the only research simulator that is capable of executing the ARCompact instruction set architecture, only the commercial xISS simulator [Syn] also supports it. This has made comparison of this technique to the execution of multithreaded programs in other systems difficult. The single-core version of ARCSIM has since been extended to support ARM execution, and has shown to be competitive in performance with other ARM simulators, suggesting that a multi-core version that supports region sharing may also perform well. Nevertheless, it has been shown that this approach can improve the performance of the simulator over its default non-sharing operation. It has also not been possible to directly compare how the use of region-sized instead of trace-sized compilation units affects the ability of the simulator

to perform this sharing, although the region-forming techniques enhanced by this work have yet to have been directly compared against trace-based JIT compilation as well.

Future work could investigate the potential for *similar* regions to be shared in the system. Currently only regions that are exactly identical in structure can be shared between threads. However, with the fact that regions can support multiple entry points for execution, there exists the potential for a thread to generate a region that is highly similar to another region, when it may be able to simply execute that region instead. For example, a thread generates a region with a large CFG, and then multiple other regions produce regions that are strictly subgraphs of this CFG. Additionally, it may be possible to identify when multiple regions sitting in the translation queue share a high degree of similarity. In this case, it may be possible to form a super-region from these regions, and then share this region with all interested execution threads. This could be done to further reduce the wasted effort of code generation in this system, both in terms of code size and time spent performing JIT compilation. Finally, future work could investigate the applicability of region-generation and sharing in other process virtual machines.

7.2.2 Sharing Profiles for Selective and Focussed Code Generation for PVMs

This thesis has investigated a system for reducing the burden of profiling on new users of applications in Android's ecosystem by sharing and using the results of profiling from the initial users. This can be used to direct selective compilation and focussed optimisation, improving code quality and therefore performance of these applications on Android devices. Limitations of this study are that while the selective compilation has been shown to reduce code size and installation time without an undue effect on performance for common Android benchmarks, this has not been empirically proven for real-world Android applications - nor has it been shown for focussed optimisation. The thesis attempts to reconcile this by comparing its performance estimation methodology against the actual observed performance of the benchmarks. It then presents this estimation of performance impact on real-world applications, hoping that the actual impact on performance will be similar. Regardless, one of the questions the thesis attempts to answer is: "do different users of the same application tend to produce similar profiles in terms of hot methods?" This study adequately answers this question, as it shows that when an independent group of users were asked to use the same appli-

cations, merged profiles that were produced from their usage patterns did not unduly affect the users' experience in the majority of cases.

Future work in this area may wish to develop a system for empirically measuring the impact of selective compilation or focussed optimisation on real-world Android applications. Furthermore, investigation could be directed more towards using an off-the-shelf compiler to perform a limit study of the potential of dynamic code generation on mobile devices, if it could be feasible to use this system to compile a small amount of extremely hot methods.

While the profiling techniques used to select hot code regions are fairly heavy-weight by virtue of being counter-based, there exist several optimised approaches which reduce the overhead of application profiling [BL94, AH02, DEC05]. Through application of these methods to the proposed profiling system, the burden on the initial users of the profiled application could be reduced, although it is suggested that these users will commonly be comprised of the developers and beta-testers. To further reduce this burden, future work may wish to investigate the possibility of having users profile only subsections of the program, and thereby crowd-source a profile that applies to the program as whole.

Finally, work appears to be ongoing in the Android Open Source Project to add a JIT compiler back into ART. If this compiler does reach maturity, it may be sufficient to use a crowd-sourced selective compilation profile for the install-time code generation as proposed in this work, and leave any unexpectedly hot code regions (as seen in two outliers in the study) to be handled by this JIT compiler.

7.2.3 Differential Fuzzing for PVM Code Generator Testing

Finally, this thesis has presented a system, DexFuzz, that combines binary fuzzing and differential testing to improve the testing of code generators within process virtual machines. This is achieved by exploitation of the fact that code generators – as well as interpreters – must share a set of semantics about the functional requirements of how a programs are executed, particularly in the case of VMs that offer the ability to "write once, run anywhere". The thesis presents domain-aware binary fuzzing, comparing it to naive and intelligent prior art for fuzzing.

In the case of naive fuzzing, domain aware binary fuzzing is able to produce programs that indicate divergent behaviour (and therefore VM bugs) much more frequently than naive fuzzing. Additionally, this is done with reduced generation of ob-

viously structurally invalid programs, wasting less time. It does this by focussing mutations on the bytecode of the binary, while the naive approach is not aware of which bytes in a binary are actually instructions. It is hoped that this focussed mutation will find more bugs in the code generators of ART, although admittedly many of the bugs thus far have been verification bugs that lead to illegal assumptions being made in the code generator. In the case of intelligent fuzzing, DEXFUZZ was compared to a tool that uses state-of-the-art mutative fuzzing techniques, called AFL. It was found that while AFL can produce test cases at a much higher rate, DEXFUZZ was able to find more programs that indicated divergent behaviour. AFL's main strengths lie in the finding of crashes and maximising the number of unique paths of execution taken in the program, qualities that DEXFUZZ does not share. Ultimately, these techniques should form part of a toolkit of fuzzing technologies, that can be used in concert to find VM bugs. Finally, the characterisation of DEXFUZZ's mutations was somewhat limited, particularly in the section trying to finding combinations of mutations that lead to divergent behaviour.

Cases of multiple mutations being responsible for bugs were found during the development of DEXFUZZ, so future extensions to DEXFUZZ could automatically search for the minimal set of mutations that result in execution divergence. This could be done by, for example, finding if the mutations can be limited to a single method, and then performing a binary search from there. This will aid users of the tool in finding the source of the bug, if they know exactly which methods and mutations are responsible for the issue.

In the case of multi-threaded programs, it is possible that while a test suite seed program produces deterministic output during normal execution, mutation could cause the program to produce non-deterministic output, either during normal execution, or during the process of error reporting. Currently DEXFUZZ filters out any found divergences between backends[1] due to this problem, by checking if one of the backends produces divergent output with itself when tested multiple times with the same program. However, better solutions to solving this problem may exist, such as running a backend multiple times to find the range of divergent outputs it produces, and then checking that the other backends' outputs fall within this set of divergent outputs, similar to a technique performed in QuickCheck [CH00].

DEXFUZZ's testing strategy is currently predicated on the belief there are no bugs in the VM such that all backends of the VM will produce exactly the same incorrect

[1]Chapter 6 refers to interpreters and code generators used by ART as backends.

output. In this case, there would be no divergence to indicate the presence of a bug. This is a general flaw with the differential testing methodology, and the only possible mitigation is to use more backends to perform testing. Alternatively, another VM that obeys the same specification could be used. This is easily possible with a VM like the Java Virtual Machine, which has a well-defined specification and a myriad of implementations available. However, the only feasible execution alternative to ART on Android is the previous VM, Dalvik, although there exist alternative Dalvik implementations such as Myriad's Alien Dalvik [Ali] and BlueStacks AppPlayer [Blu]. It would be useful to execute programs with both the ART and Dalvik VMs, provided that the compared output is filtered to remove known divergences between the two implementations - namely the way that they report errors.

Finally, the application of techniques such as genetic algorithms (as used by AFL and other mutative fuzzers) could intelligently guide mutations based on the fitness of the resulting program. Additionally, adding more mutations such as changing the access flags of classes, methods, and fields, injecting the printing of virtual register values at random locations, or calls to specific system methods such as the `System.gc()` method could increase the ability of DEXFUZZ to find VM bugs.

This thesis has presented cases where information sharing can be exploited to improve code generation as performed by process virtual machines. This has been done in the areas of JIT compilation, profiling to direct AOT compilation for a mobile VM, and testing of code generators used by virtual machines. As process virtual machines form a major part of many aspects of computing today, it is important to improve both their performance and robustness, and the techniques presented in this thesis help with this.

List of Figures

List of Tables

Listings

Bibliography

[AA12] Susanne Albers and Antonios Antoniadis. Race to Idle: New Algorithms for Speed Scaling with a Sleep State. In *Proceedings of the Twenty-third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 1266–1285. SIAM, 2012.

[AAB+00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Syst. J.*, 39(1):211–238, January 2000.

[ABdH+14] Rafael Auler, Edson Borin, Peli de Halleux, Micha Moskal, and Nikolai Tillmann. Addressing JavaScript JIT Engines Performance Quirks: A Crowdsourced Adaptive Compiler. In *Compiler Construction*, volume 8409 of *Lecture Notes in Computer Science*, pages 218–237. Springer Berlin Heidelberg, 2014.

[ABF12] Håkan Ardö, Carl Friedrich Bolz, and Maciej Fijałkowski. Loop-aware Optimizations in PyPy's Tracing JIT. *SIGPLAN Not.*, 48(2):63–72, October 2012.

[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.

[AFL] american-fuzzy-lop on Google Code. `https://code.google.com/p/american-fuzzy-lop/`. Accessed: 2014-11-12.

[AH02] Taweesup Apiwattanapong and Mary Jean Harrold. Selective Path Profiling. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '02, pages 35–42, New York, NY, USA, 2002. ACM.

[Ali] Alien Dalvik. `http://www.myriadgroup.com/products/device-solutions/mobile-software/alien-dalvik/`. Accessed: 2014-11-25.

[And] EEMBC - AndEBench. `https://www.eembc.org/andebench/`. Accessed: 2015-02-18.

[AOS] AOSP Development Team. Dalvik Bytecode — Android Open Source Project. `https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html`.

[AvRAF09] Wolfram Amme, Jeffery von Ronne, Philipp Adler, and Michael Franz. The Effectiveness of Producer-side Machine-independent Optimizations for Mobile Code. *Softw. Pract. Exper.*, 39(10):923–946, July 2009.

[Ayc03] John Aycock. A Brief History of Just-in-Time. *ACM Comput. Surv.*, 35:97–113, June 2003.

[BBF+10] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. SPUR: a trace-based JIT compiler for CIL. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 708–725, New York, NY, USA, 2010. ACM.

[BCFR09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, Genova, Italy, 2009. ACM.

[BCW+10] Michael Bebenita, Mason Chang, Gregor Wagner, Andreas Gal, Christian Wimmer, and Michael Franz. Trace-based Compilation in Execu-

tion Environments Without Interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 59–68, New York, NY, USA, 2010. ACM.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM.

[Bel73] James R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, June 1973.

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[BEvKK+11] Igor Böhm, Tobias J.K. Edler von Koch, Stephen Kyle, Björn Franke, and Nigel Topham. Generalized Just-In-Time Trace Compilation using a Parallel Task Farm in a Dynamic Binary Translator. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'11. ACM, 2011.

[BFKR09] F. Brandner, A. Fellnhofer, A. Krall, and D. Riegler. Fast and Accurate Simulation Using the LLVM Compiler Framework. In *Proceedings of the 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO'09, pages 1–6, 2009.

[BKGB06] Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji. Thread-Shared Software Code Caches. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 28–38, Washington, DC, USA, 2006. IEEE Computer Society.

[BL94] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, July 1994.

[BL96] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[Blu] BlueStacks AppPlayer. `http://www.bluestacks.com/app-player.html`. Accessed: 2014-11-25.

[Caf] CaffeineMark - Android Apps on Google Play. `https://play.google.com/store/apps/details?id=com.android.cm3`. Accessed: 2015-02-18.

[CAR08] Simone Campanoni, Giovanni Agosta, and Stefano Crespi Reghizzi. A Parallel Dynamic Compiler for CIL Bytecode. *SIGPLAN Not.*, 43:11–20, April 2008.

[CH00] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[CK94] Bob Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. In *Proceedings of the ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, SIGMETRICS'94, pages 128–137, 1994.

[CLCG00] W.-K. Chen, S. Lerner, R. Chaiken, and D.M. Gillies. Mojo: A Dynamic Optimization System. In *Proceedings of the Third ACM Workshop on Feedback-Directed and Dynamic Optimization*, FDDO'00, 2000.

[CMS+11] Mason Chang, Bernd Mathiske, Edwin Smith, Avik Chaudhuri, Andreas Gal, Michael Bebenita, Christian Wimmer, and Michael Franz. The Impact of Optional Type Information on JIT Compilation of Dynamically Typed Languages. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 13–24, New York, NY, USA, 2011. ACM.

[DEC05] Madeline Diep, Sebastian Elbaum, and Myra Cohen. Profiling Deployed Software: Strategic Probe Placement. *CSE Technical reports*, page 17, 2005.

[DRH14] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Language Fuzzing Using Constraint Logic Programming. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 725–730, New York, NY, USA, 2014. ACM.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[EA97] Kemal Ebcioğlu and Erik R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility, 1997.

[FMT+08] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Phil Barnard, Elton Ashton, Eric Courtois, et al. MILEPOST GCC: Machine Learning-based Research Compiler. 2008.

[GAS+00] Michael Gschwind, Erik R. Altman, Sumedh Sathaye, Paul Ledak, and David Appenzeller. Dynamic and Transparent Binary Translation. *Computer*, 33:54–59, March 2000.

[GBCF07] Andreas Gal, Michael Bebenita, Mason Chang, and Michael Franz. Making the Compilation "Pipeline" Explicit: Dynamic Compilation Using Trace Tree Serialization. Technical Report 07-12, University of California, Irvine, 2007.

[GES+09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based Just-in-Time Type Specialization for Dynamic Languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[Gooa] Google. Activity — Android Developers. `http://developer.android.com/reference/android/app/Activity.html`. Accessed: 2014-02-18.

[Goob] Google. Android One. `http://www.android.com/one/`. Accessed: 2014-02-14.

[Gooc] Google. Profiling with Traceview and dmtracedump. `http://developer.android.com/tools/debugging/debugging-tracing.html`. Accessed: 2014-02-22.

[GPF06] Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: An Fffective JIT Compiler for Resource-constrained Devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE'06, pages 144–153. ACM, 2006.

[GRS+13] Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter Mössenböck. An Efficient Native Function Interface for Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 35–44, New York, NY, USA, 2013. ACM.

[HHCM09] Jungwoo Ha, Mohammad R. Haghighat, Shengnan Cong, and Kathryn S. McKinley. A Concurrent Trace-based Just-In-Time Compiler for Single-threaded JavaScript. In *Workshop on Parallel Execution of Sequential Programs on Multicore Architectures*, PESPMA'09, June 2009. in conjunction with ISCA 09.

[HHY+12] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: A Multi-threaded and Retargetable Dynamic Binary Translator on Multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 104–113, New York, NY, USA, 2012. ACM.

[HHZ12] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Secu-*

rity Symposium, Security'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[HIT] HITB2014AMS - Day 1 - State of the ART: Exploring the new Android KitKat Runtime. `https://www.corelan.be/index.php/2014/05/29/hitb2014ams-day-1-state-of-the-art-exploring-the-new-android-kitkat-runtime/`. Accessed: 2014-11-20.

[HLW+13] Chun-Chen Hsu, Pangfeng Liu, Jan-Jan Wu, Pen-Chung Yew, Ding-Yong Hong, Wei-Chung Hsu, and Chien-Min Wang. Improving Dynamic Binary Optimization Through Early-exit Guided Code Region Formation. *SIGPLAN Not.*, 48(7):23–32, March 2013.

[HM11] Christian Häubl and Hanspeter Mössenböck. Trace-based Compilation for the Java HotSpot Virtual Machine. In *Proceedings of the International Conference on Principles and Practice of Programming in Java*, PPPJ'11, Kongens Lyngby, Denmark, August 2011.

[HU94] Urs Hölzle and David Ungar. A Third-generation Self Implementation: Reconciling Responsiveness with Performance. In , pages 229–243. ACM Press, 1994.

[HWM14] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Trace Transitioning and Exception Handling in a Trace-based JIT Compiler for Java. *ACM Trans. Archit. Code Optim.*, 11(1):6:1–6:26, February 2014.

[IHWN11] H. Inoue, H. Hayashizaki, Peng Wu, and T. Nakatani. A Trace-based Java JIT Compiler Retrofitted from a Method-based Compiler. In *Code Generation and Optimization, 2011 9th Annual IEEE/ACM International Symposium on*, CGO'11, pages 246 –256, April 2011.

[JHGK09] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. jFuzz: A Concolic Whitebox Fuzzer for Java. In *NASA Formal Methods*, pages 121–125, 2009.

[JMO14] Dong-Heon Jung, Soo-Mook Moon, and Hyeong-Seok Oh. Hybrid Compilation and Optimization for Java-based Digital TV Platforms. *ACM Trans. Embed. Comput. Syst.*, 13(2s):62:1–62:27, January 2014.

[KAH07] Prasad Kulkarni, Matthew Arnold, and Michael Hind. Dynamic Compilation: The Benefits of Early Investing. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 94–104, New York, NY, USA, 2007. ACM.

[KF11] Prasad Kulkarni and Jay Fuller. JIT Compilation Policy on Single-Core and Multi-core Machines. In *Interaction between Compilers and Computer Architectures, 2011 15th Workshop on*, INTERACT'11, pages 54 –62, feb. 2011.

[KGL+00] Chandra Krintz, David Grove, Derek Lieber, Vivek Sarkar, and Brad Calder. Reducing the Overhead of Dynamic Compilation. *Software: Practice And Experience*, 31:200–1, 2000.

[Kla00] Alexander Klaiber. The Technology Behind Crusoe Processors. Technical report, Transmeta Corporation, January 2000.

[LAS14] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 216–226, New York, NY, USA, 2014. ACM.

[LIB15] Derek Lockhart, Berkin Ilbeyi, and Christopher Batten. Pydgin: Generating Fast Instruction Set Simulators from Simple Architecture Descriptions with Meta-tracing JIT Compilers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2015, Philadelphia, PA, USA, March 29-31, 2015*, pages 256–267, 2015.

[Lim07] ARM Limited. *ARM1156T2F-S Architecture with Thumb-2 Core Technology*, 2007.

[Lin] Linpack for Android. `http://www.greenecomputing.com/apps/linpack/`. Accessed: 2014-02-07.

[LMMP07] Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, and Roberto Paleari. A Smart Fuzzer for x86 Executables. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, SESS '07, pages 7–, Washington, DC, USA, 2007. IEEE Computer Society.

[LPKL12] Yeong-Kyu Lim, S. Parambil, Cheong-Ghil Kim, and See-Hyung Lee. A Selective Ahead-Of-Time Compiler on Android Device. In *Information Science and Applications (ICISA), 2012 International Conference on*, pages 1–6, May 2012.

[LYBB] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. Verification of Class Files - Java Virtual Machine Specification Java SE 8 Edition. `https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.10`. Accessed: 2015-05-25.

[Mac93] Stavros Macrakis. The structure of ANDF: Principles and examples. *Open Systems Foundation*, 1993.

[May87] C. May. Mimic: A Fast System/370 Simulator. In *Papers of the Symposium on Interpreters and Interpretive Techniques*, SIGPLAN '87, pages 1–13, New York, NY, USA, 1987. ACM.

[McC60] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM*, 3(4):184–195, April 1960.

[MCHJ15] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. A Survey of the Use of Crowdsourcing in Software Engineering. Technical Report RN/14/01, Department of Computer Science, University College London, 2015.

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[MM11] M. Mehrara and S. Mahlke. Dynamically Accelerating Client-side Web Applications Through Decoupled Execution. In *Code Generation and Optimization, 2011 9th Annual IEEE/ACM International Symposium on*, CGO'09, pages 74 –84, April 2011.

[MS] Kevin Millikin and Florian Schneider. A New Crankshaft for V8. `http://blog.chromium.org/2010/12/new-crankshaft-for-v8.html`. Accessed: 2015-05-27.

[NBS+02] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A Universal Technique for Fast and Flexible Instruction-set Architecture Simulation. In *Proceedings of the 39th annual Design Automation Conference*, DAC '02, pages 22–27, New York, NY, USA, 2002. ACM.

[Ora] Oracle. Learn about Java Technology. `https://www.java.com/en/about/`.

[Pag] Larry Page. Google July 2013 Earnings Call. `https://plus.google.com/+LarryPage/posts/QVbFayWxfrm`. Accessed: 2014-03-10.

[PC97] Michael Plezbert and Ron K. Cytron. Does "Just in Time" = "Better Late than Never"? In *In Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'97, pages 120–131. ACM Press, 1997.

[Pea] Peachfuzzer. `http://peachfuzzer.com/`. Accessed: 2014-11-20.

[Pho] Spice Smart Phones. Spice Dream Uno 498. `http://www.spicesmartphones.com/Dream-UNO-Mi-498`. Accessed: 2014-02-14.

[Pic] Sundar Pichai. Android Device Activations Announcement. `https://plus.google.com/+SundarPichai/posts/NeBW7AjT1QM`. Accessed: 2014-03-10.

[PM10] M. Papadakis and N. Malevris. Automatic Mutation Test Case Generation via Dynamic Symbolic Execution. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 121–130, Nov 2010.

[Pol] Konstantinos Polychronis. BenchmarkPi - Android Benchmarking Tool. `http://androidbenchmark.com/`. Accessed: 2014-01-27.

[PR98] Ian Piumarta and Fabio Riccardi. Optimizing Direct Threaded Code by Selective Inlining. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 291–300, New York, NY, USA, 1998. ACM.

[PTB+97] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for Applications a Way Ahead of Time (WAT) Compiler. In *Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3*, COOTS'97, pages 3–3, Berkeley, CA, USA, 1997. USENIX Association.

[Pur72] Paul Purdom. A Sentence Generator for Testing Parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, USENIX-JVM'01, pages 1–12, 2001.

[QDZ06] Wei Qin, Joseph D'Errico, and Xinping Zhu. A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-compiled Instruction-set Simulation. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '06, pages 193–198, New York, NY, USA, 2006. ACM.

[Qua] Aurora Softworks - Quadrant. `http://www.aurorasoftworks.com/products/quadrant`. Accessed: 2015-02-18.

[Ray] RayTracer Benchmark - Android Apps on Google Play. `https://play.google.com/store/apps/details?id=org.noote.RayTracer`. Accessed: 2015-02-18.

[RBJ01] Ramesh Radhakrishnan, Ravi Bhargava, and Lizy K. John. Improving Java Performance Using Hardware Translation. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 427–439, New York, NY, USA, 2001. ACM.

[RCCS07] Vijay Janapa Reddi, Dan Connors, Robert Cohn, and Michael D. Smith. Persistent Code Caching: Exploiting Code Reuse Across Executions and Applications. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 74–88, Washington, DC, USA, 2007. IEEE Computer Society.

[REP+11] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. Scaling Up Automated Test Generation: Automatically Generating Maintainable Regression Unit Tests for Programs. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 23–32, Washington, DC, USA, 2011. IEEE Computer Society.

[RMD03] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 758–763, New York, NY, USA, 2003. ACM.

[RRJ+99] Ramesh Radhakrishnan, R. Radhakrishnany, Lizy K. John, Juan Rubio, L. K. Johny, and N. Vijaykrishnan. Execution Characteristics of Just-In-Time Compilers, 1999.

[San06] Graham Sanderson. High Performance: Writing a Sony PlayStation Emulator Using Java Technology, 2006.

[SCEG08] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual Machine Showdown: Stack Versus Registers. *ACM Trans. Archit. Code Optim.*, 4(4):2:1–2:36, January 2008.

[Sci] Scimark - Android Apps on Google Play. `https://play.google.com/store/apps/details?id=net.danielroggen.scimark`. Accessed: 2015-02-18.

[Sma] smali - an assembler/disassembler for Android's DEX format on Google Code. `https://code.google.com/p/smali/`. Accessed: 2014-11-19.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[Str] StrategyAnalytics. GitHut - Programming Languages and GitHub. `http://githut.info/`.

[Str15] StrategyAnalytics. Android Shipped 1 Billion Smartphones Worldwide in 2014. `http://www.strategyanalytics.com/default.aspx?mod=reportabstractviewer&a0=10539`, 29 January 2015.

[SWFT14] Tom Spink, Harry Wagstaff, Björn Franke, and Nigel Topham. Efficient Code Generation in a Region-based Dynamic Binary Translator. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 3–12, New York, NY, USA, 2014. ACM.

[Syn] Synopsys. DesignWare ARC xISS. `http://www.synopsys.com/dw/ipdir.php?ds=sim_xiss`. Accessed: 2015-05-27.

[SYN06] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A Region-based Compilation Technique for Dynamic Compilers. *ACM Trans. Program. Lang. Syst.*, 28:134–174, January 2006.

[Tea] V8 Development Team. V8 JavaScript Engine. `https://code.google.com/p/v8/`. Accessed: 2015-05-27.

[Tho68] Ken Thompson. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[UKL06] P. Unnikrishnan, M. Kandemir, and F. Li. Reducing Dynamic Compilation Overhead by Overlapping Compilation and Execution. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, pages 929–934, Piscataway, NJ, USA, 2006. IEEE Press.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[WCB+09] Christian Wimmer, Marcelo S. Cintra, Michael Bebenita, Mason Chang, Andreas Gal, and Michael Franz. Phase Detection using Trace Compilation. In *Proceedings of the 7th International Conference on*

Principles and Practice of Programming in Java, PPPJ '09, pages 172–181, New York, NY, USA, 2009. ACM.

[Wha01] John Whaley. Partial Method Compilation using Dynamic Profile Information. In *Proceedings of the 16th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 166–179, New York, NY, USA, 2001. ACM.

[WLC+11] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. COREMU: A Scalable and Portable Parallel Full-system Emulator. *SIGPLAN Not.*, 46(8):213–222, February 2011.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd annual International Symposium on Computer Architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[WPC+11] Chih-Sheng Wang, Guillermo Perez, Yeh-Ching Chung, Wei-Chung Hsu, Wei-Kuan Shih, and Hong-Rong Hsu. A Method-based Ahead-of-time Compiler for Android Applications. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '11, pages 15–24, New York, NY, USA, 2011. ACM.

[WR96] Emmett Witchel and Mendel Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Measurement and Modeling of Computer Systems*, SIGMETRICS'96, pages 68–79, 1996.

[WZLY13] Guanxing Wen, Yuqing Zhang, Qixu Liu, and Dingning Yang. Fuzzing the ActionScript Virtual Machine. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 457–468, New York, NY, USA, 2013. ACM.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

[ZBS07] Mathew Zaleski, Angela Demke Brown, and Kevin Stoodley. YETI: a graduallY Extensible Trace Interpreter. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 83–93, New York, NY, USA, 2007. ACM.

[ZGC+15] Xiaochun Zhang, Qi Guo, Yunji Chen, Tianshi Chen, and Weiwu Hu. HERMES: A Fast cross-ISA Binary Translator with Post-optimization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 246–256, Washington, DC, USA, 2015. IEEE Computer Society.