THE UNIVERSITY
of EDINBURGH

Enabling aggressive compiler optimization for the mobile environment

Paschalis Mpeis

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2021

Abstract

Aggressive code optimization on the mobile environment is a difficult endeavor. Billions of users rely on mobile devices for their daily computing tasks. Yet, they mostly run poorly optimized code, under-utilizing their already limited processing and energy resources. Existing optimization approaches, like iterative compilation, perform well in other domains but fall short on the mobile environment. They either rely on representative inputs that are hard to reconstruct, or expose users to slowdowns and crashes.

An ideal solution must be able to perform an optimization search by repeatedly evaluating different optimization decisions on the same input. That input should be representative of actual user usage without requiring developers to artificially create it. Finally, users should never be exposed to slow or crashing evaluations, a quite common side-effect of iterative compilation. This thesis presents a novel approach with all above in mind, bringing aggressive code optimization to the mobile environment.

With a transparent capture mechanism, real user inputs can be stored. This mechanism is infrequently invoked and remains unnoticeable from the users. A single capture is enough to enable offline, input-driven code optimization. It supports C functions as well as code regions of interactive Android applications. It allows controlling the timing and frequency of captures, it bails out on imminent high-impact runtime events, and excludes from captures some immutable data.

A replay-based evaluation mechanism is able to repeatedly restore a captured input while changing the underlying code. For C programs, it employs compile and link-time strategies to consistently work despite code transformations. For Android apps, a novel mechanism was developed, able to replay using different code types. These are the original Android-compiled code, interpretation, and LLVM-generated code. Additionally, it works well even in the presence of memory-shuffling security mechanisms.

Capture and replay is fused into an iterative compilation system that uses offline, replay-based evaluations. Initially, real inputs are captured online, without noticeably affecting the users. For C and interactive apps, captures required on average 2ms and 15ms respectively. Then, an optimization search is performed by repeatedly replaying the inputs using different code transformations. As this happens offline, any crashing or erroneous executions are not affecting the users. C programs became 29% faster using a random search, while interactive apps became 44% faster using a genetic algorithm and a novel Android backend based on LLVM. Finally, with crowd-sourcing, the offline evaluation effort was significantly accelerated. Specifically, for the user with the highest workload the search accelerated by 7 times.

Lay Summary

Mobile devices are not only limited in terms of processing power and battery capacity, but they also run mostly poorly optimized code. By repeatedly evaluating different code generation strategies, one can discover faster code than what compilers generate out of the box. This simple technique, called iterative compilation, despite being successful in other domains, it struggles to adapt on the mobile environment.

Any iterative compilation approaches that evaluate code when a device is not being used rely on pre-generated inputs. However, to prepare any realistic inputs in advance is a non-trivial issue. The approaches that run while a device is actively being used, operate directly on real inputs. However, the input can arbitrarily change, making the comparisons between different compilation strategies difficult. On top of that, some strategies can introduce slowdowns or crashes, affecting the user experience. This thesis proposes a novel technique that enjoys the benefits of both approaches. When a device is being used, it stores real inputs without noticeable overheads. And when it becomes idle, it repeatedly uses the stored inputs for evaluating different compilation strategies.

Initially, real inputs are captured infrequently. Three different mechanisms have been developed. The first, minimizes the amount of data to store by saving only the chunks of memory that are needed by the program. The second improves upon the first one by keeping within those memory chunks only the bits that contain relevant data. The third one runs a bit faster, as it does not attempt to minimize the stored data at all. All three approaches remain unnoticeable from the users as they incur overheads between 2ms and 15ms. Once an input is captured, it can be replayed several times, as a means of evaluating different compilation strategies. As this happens at idle times, any slow or crashing evaluations are not affecting the users. As the same input is repeatedly used, the comparisons between different strategies are sound.

Systems for C and Android applications were developed. As Android had a few conservative optimization strategies, a novel aggressive code generation add-on was implemented. By randomly searching between different compilation strategies, the C code became 29% faster. With an algorithm based on genetic evolution, the Android code became 44% faster. Finally, by leveraging multiple users, the time needed to run iterative compilation was significantly accelerated. New users can utilize the information accumulated by the previous ones to find better code in less time. Specifically, the most busy user was able to finish code evaluations 7x faster.

Acknowledgements

I would like to express my sincere gratitude to my advisers Dr. Hugh Leather and Dr. Pavlos Petoumenos. Without their guidance, motivation, and of course ample patience throughout the years I would not have been able to materialize the ideas that are presented in this thesis. I simply could not have asked for better advisors.

Next I would like to thank my family for their emotional support throughout the years. A special mention to my grandfather, with whom I share my name and surname. I gladly explain to him each time we meet what my research is about, despite knowing that I will have to do it again the next time.

I would also like to thank my friends for supporting me, and grab the opportunity to apologize for the late replies to those text messages. Last but not least, I would like to thank Nicole. She has helped me more than I could have described.

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:

- **Iterative compilation on mobile devices.**
  Paschalis Mpeis, Pavlos Petoumenos, and Hugh Leather.
  ADAPT, HiPEAC (2016), [MPE+16]

- **Developer and user-transparent compiler optimization for interactive applications.**
  Paschalis Mpeis, Pavlos Petoumenos, Kim Hazelwood, and Hugh Leather.
  PLDI (2021) [MPE+21]

- **Object Intersection Captures on Interactive Apps to Drive a Crowd-Sourced Replay-Based Compiler Optimization.**
  Paschalis Mpeis, Pavlos Petoumenos, Kim Hazelwood, and Hugh Leather.
  {under submission to TACO Journal}

(*Paschalis Mpeis*)

Table of Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

ANN Artificial Neural Network. 33
AOSP Android Open Source Project. 18, 73, 77
AOT Ahead Of Time. 19, 37, 61, 62, 94
API Application Programming Interface. 13, 20
APK Android Application Package. 18, 19, 37
ART Android RunTime. 18, 20, 21, 37, 62, 65–67, 75, 76, 94, 125, 139
ASLR Address Space Layout Randomization. 17, 138
AST Abstract Syntax Tree. 33

CFG Control Flow Graph. 35, 38
CPU Central Processing Unit. 15, 22, 142, 143
CR Capture and Replay. 21, 35, 36, 39–41

DAG Directed Acyclic Graph. 30
DNN Deep Neural Network. 33
DSP Digital Processing Unit. 143
DVM Dalvik Virtual Machine. 18, 19

ELF Executable and Linkable Format. 19, 76

FOSS Free and Open Source Software. 17, 18

GA Genetic Algorithm. xi, xii, 30, 32, 90, 107, 108, 110–115, 117, 120, 127, 130, 139, 144
GC Garbage Collection. ix, 79, 83, 125
GOT Global Offset Table. 50
GPU Graphics Processing Unit. 143
GUI Graphical User Interface. 37

HPC High-Performance Computing. 35

IOMMU Input/Output Memory Management Unit. 143
IR Intermediate Representation. 13, 20, 62, 67

JIT Just In Time. 19, 20
JNI Java Native Interface. 20, 58, 68, 72, 80, 81, 93
JNI Java Native Interface. 18, 19, 65
JVM Java Virtual Machine. 38

MMU Memory Management Unit. 143

NDK Native Development Kit. 18

OS Operating System. 1, 17, 18, 96, 139, 142, 143
OTA Over The Air. 19

PLT Procedure Linkage Table. 50

UI User Interface. 62

VMA Virtual Memory Area. 17, 18, 45, 139, 140, 142

Chapter 1

Introduction

More and more of our computing tasks rely on smart mobile devices. With active users now more than 5.2 billion [GSM20], smartphones are the defining computing medium of our era. Nevertheless, mobile devices are severely limited, both in terms of processing power and battery life. Aggressive performance and energy optimizations are not only welcome, they are necessary for maintaining the user's Quality of Experience, supporting novel capabilities, or providing reasonable levels of autonomy. What we get instead is poorly optimized software. Even the preeminent mobile platform, Android, relies on a compiler with just 18 distinct optimizations, an order of magnitude less than what traditional optimizing compilers offer [GOO20a]. As a result, immense amounts of performance and energy are wasted impacting the smartphone user's experience.

Iterative compilation, despite being successfully employed for aggressive optimization in several domains, has never been widely adopted in the mobile environment due to several obstacles. Existing offline approaches rely on representative hardware, software, and inputs, making them unsuitable for mobile applications. And online approaches inevitably expose users to sub-optimal, erroneous, or crashing executions. Despite ultimately resulting in faster code, it is rather unlikely that these side-effects will be tolerated by the end users, the application developers, or the device and Operating System (OS) manufacturers. This thesis presents a novel fusion of iterative compilation with a capture and replay system able to enjoy the best of both worlds.

The remainder of this chapter is organized as follows. Section 1.1 demonstrates a strong motivating example for pursuing iterative compilation on the mobile environment. Section 1.2 briefly discusses challenges for an input capture and replay mechanism suitable for mobile devices. Section 1.3 presents the main contributions of this work, followed by some publications in Section 1.4. The overall structure of this thesis

is presented in Section 1.5. Finally, Section 1.6 concludes this chapter.

1.1 Motivation for iterative compilation on mobile systems

Iterative compilation [KIS+99] is an aggressive code optimization technique that readily outperforms a compiler's standard optimization levels. It searches through different combinations of code transformations and transformation parameters, evaluates their effect on performance, and at the end keeps the best performing set. Multiple approaches exist to select optimization sequences: randomly, through genetic search, statistical models [PAR+11], or with the help of a machine learned model [LEA+09a]. Despite having the benefits of iterative compilation clearly demonstrated by the research community, the technique has not been applied in a general way on mobile systems. The remainder of this section explores why.

1.1.1 Offline compiler optimization

Regardless of the methodology used to optimize an application offline, there is in general a reliance on some kind of a representative evaluation system (both hardware and software), and inputs that are both representative and deterministic. By repeatedly executing the application with a deterministic input under different code transformations, the optimization decisions can be compared directly and the best performing one will be selected. By using a representative system and representative inputs, there can be confidence that the chosen optimization strategies will work well under most scenarios.

This simple methodology has proved hard to adapt to mobile systems. There is no such thing as a representative system where an application can be optimized once for every other system. Instead, it has to be optimized for each system individually. And even then, it is hard to create inputs without any involvement from the application developer. Mobile applications (and interactive applications in general) tend to have complex inputs, including configuration files, system state, user events, and network data. Packaging all these in a neat deterministic input that introduces no undesirable side-effects is a non-trivial problem. Making sure that they are representative is even harder. Expecting mobile developers to put the effort required to do all these with little incentive is unrealistic.

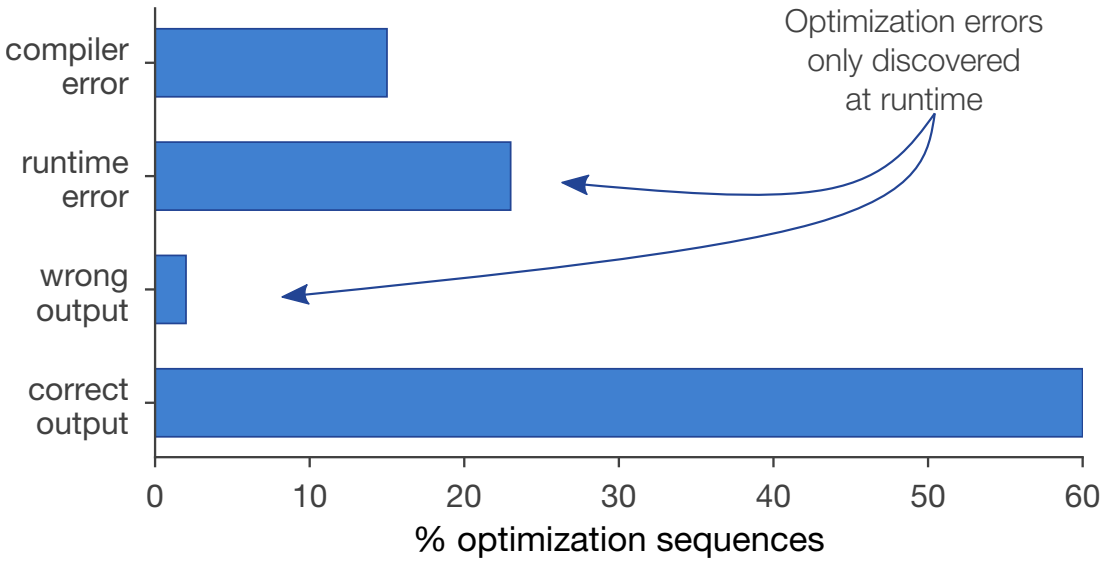

Figure 1.1: Compilation outcome with LLVM version 10 when using 100 randomly generated optimization sequences for the `FFT` kernel from the *Scimark* benchmark suite. A Google Pixel 4 device was used that ran Android 10. 25% of the sequences result in a binary that does not behave as expected at runtime. In an online optimization system, this would directly affect the user experience and might lead to data corruption with long-term consequences for the user.

1.1.2 Online optimization is risky

An alternative approach is to evaluate compiler optimization decisions online. The application is compiled using different code transformations and evaluations happen when the user uses that application by interacting with their device. Evaluating each decision is just a matter of profiling the application while it is being used. This removes the problem of identifying representative systems and inputs: the system and the input are by definition the ones that it is desired to optimize the application for.

While a workable solution in some cases, in the general case online optimization creates a whole new set of problems. The first is that there is no hard guarantee that compiler code transformations will not introduce errors. Figure 1.1 shows what happens when LLVM optimization passes are randomly chosen and subsequently applied on a benchmark, `FFT` from *Scimark*. Only 60 out of the 100 different code transformation sequences lead to a binary that behaves the way it was supposed to. In 15 cases the optimizations cause the compiler to crash or timeout. This is a manageable problem. What is not manageable is the other 25 optimization sequences that lead to compilation errors that only become apparent at runtime, either with a program crash,

a program timeout, or a wrong output. When encountered in offline optimization, such code transformation sequences are simply rejected with no other side-effects. But in an online optimization setup, broken optimizations are visible to the user and affect the user experience. Even worse, silent errors that cause the program behavior to change can lead to unwanted changes to permanent state, either local or remote, with long-term consequences for the user. This is an unacceptable risk.

1.1.3 Online optimization affects the user experience

Broken optimizations are just one part of the wider problem of code transformation choices being visible to the user. Even if the chosen compiler optimizations lead to valid binaries, there is a good chance that the new binary will be slower than what the user expects. Figure 1.2 shows the performance of 50 `FFT` binaries that were generated by applying a random sequence of LLVM optimization passes. Performance is relative to the version produced by the Android compiler. All binary versions are slower than it, from 15% to as much as 8x slower. Even if this is an initial exploration of the optimization space and later optimization choices improve performance, the user will still have experienced unacceptable levels of slowdown. In practice, as it is showcased in Chapter 6, suboptimal binaries are common even in later stages of an otherwise profitable optimization process. Such behavior might lead to the user removing the application or disabling the optimizer, negating the benefit of using iterative compilation.

1.1.4 Online optimization is slow

Even if these limitations are somehow worked around, the fundamental problem remains: there is no control over the context in which optimizations are being evaluated. The most important component of this context is the input. In offline search, every optimization is evaluated on the exact same set of inputs. Program versions that take less time to perform the same amount of useful work are better, versions that take longer are worse.

With online search, on the other hand, code transformations are evaluated on whatever input happens to be fed to the program. This is not necessarily a problem. If the program is performing more or less the same amount of work every time and this is known, using the execution runtime to evaluate different code transformations is a sound approach. If the amount of useful work performed can be estimated, the work per unit of time can be used for comparing optimizations evaluated on different inputs.

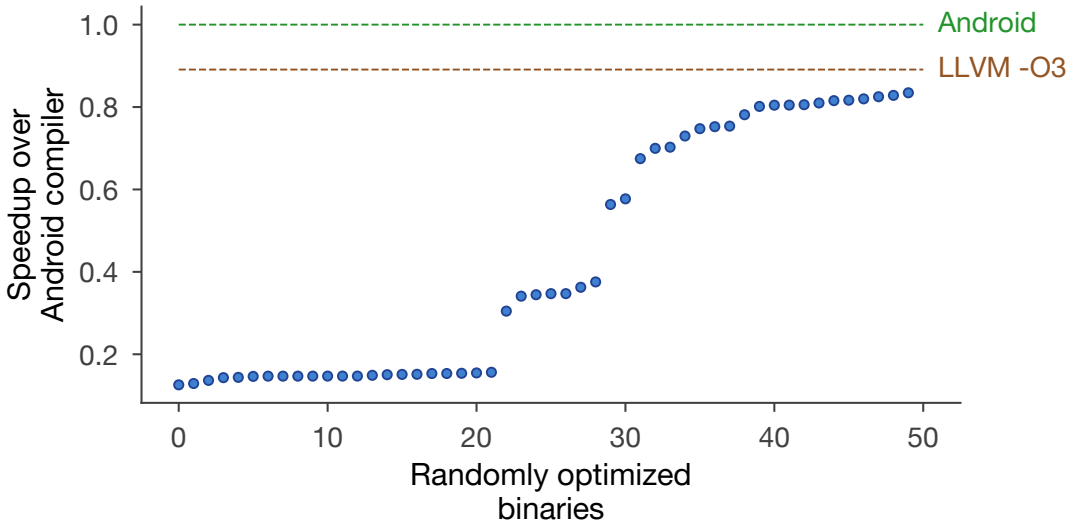

Figure 1.2: Speedup over the Android compiler for 50 randomly generated LLVM code transformation sequences applied on the `FFT` kernel. Sequences that crash the compiler or the execution are discarded. All of them slow down the application relative to both the Android compiler and LLVM `-O3`. In the worst case, the program runs 8x slower. Evaluating these optimization sequences online would have an unacceptable impact on the user experience.

But in the general case, there is no information about what the program does. The only way to make online optimization work is to evaluate each optimization a large number of times with different inputs. If all optimizations are evaluated on a similar sample of inputs, direct comparisons of execution time will be statistically meaningful. The problem is this can be a very lengthy process.

Figure 1.3 shows such an approach. It is attempted to estimate the speedup of LLVM `-O1` over `-O0` for the `FFT` kernel using multiple evaluations with different inputs drawn from a uniform distribution. This experiment is repeated 10000 times to capture a wide range of outcomes. The evolution of the speedup estimation for a single experiment is shown as a line and the range of likely outcomes as areas. The same information is provided for the offline case for comparison. While `-O1` is clearly better than `-O0`, almost doubling performance, the online estimation varies wildly: from almost 2x slowdown to 8x speedup. It is not until the 22nd evaluation when the estimated speedup stops going below 1 and the online approach can reliably decide that `-O1` is better than `-O0`. It takes another 20 evaluations for the estimation to start stabilizing. This behavior is not an outlier. In 25% of the repeated experiments, after 25 evaluations

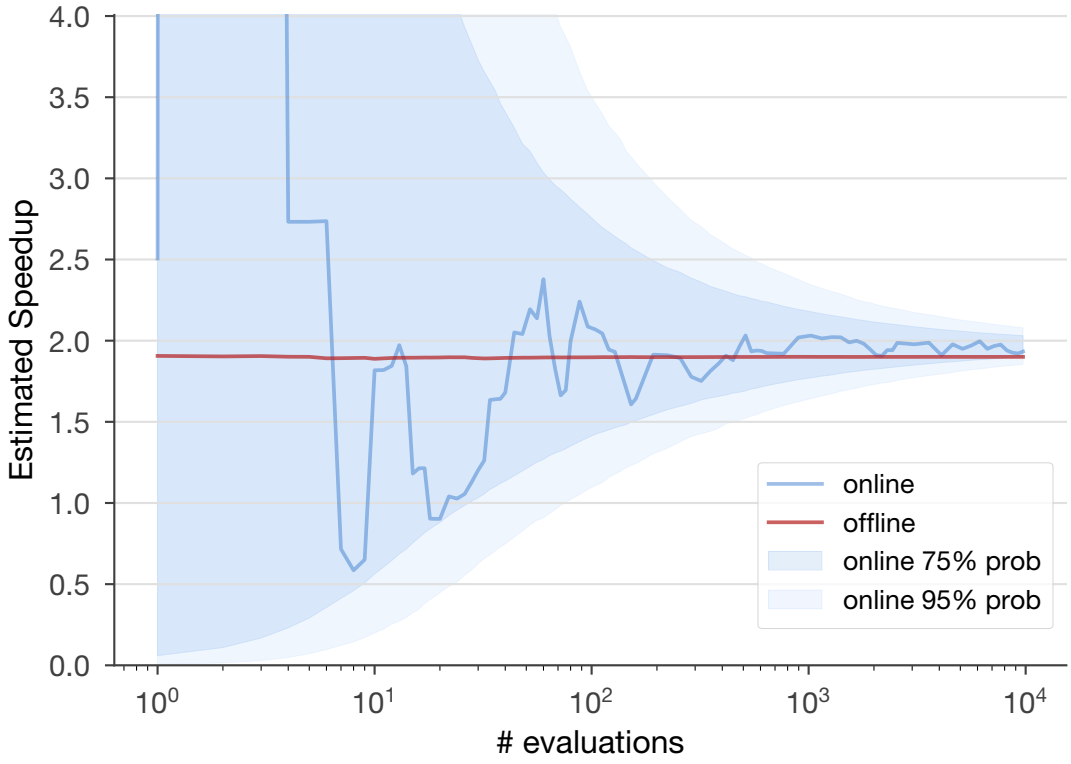

Figure 1.3: Estimation of the speedup of LLVM `-O1` over `-O0` for the *SciMark FFT* benchmark as the number of evaluations increases. Offline search always uses a `FFT_SIZE_LARGE` input. Online search performs each evaluation with a different randomly selected input between `FFT_SIZE` and `FFT_SIZE_LARGE`. Lines represent single experiments. Areas represent 75% and 95% confidence intervals over 10000 repeated experiments.

a speedup below 1 was still being estimated. In 5% of the repeated experiments, more than 70 evaluations are needed just to determine that `-O1` is better and more than 1000 evaluations are needed to reduce its speedup estimation error below 10%. In contrast, there is very little variation across the 10000 offline evaluation experiments. In 95% of them, a single measurement is enough (though not statistically safe) to estimate the speedup within 2% of the real value.

This scenario points towards a 100-1000x increase in evaluation time compared to an offline approach in order to get a comparable level of confidence in the optimization decisions. This is very likely an underestimation. Other sources of experimental noise that cannot be controlled online, such as frequency scaling, thermal throttling, and contention for resources affect the confidence levels. Skewed distributions of input sizes and processing times may reduce the confidence even further. Seen in the context of an iterative compilation system like the ones presented in Chapters 4 and 6, this would translate into 100k to 1m+ evaluations for each optimized program. For `FFT` this means tens of hours of repeated experiments. This is unfeasible. Mobile applications are typically active for only a few tens of minutes every day and are updated every few weeks. There is just not enough evaluation time for an online approach.

1.1.5 Beyond online and offline optimization

An ideal optimization approach should combine the best of both worlds: online and offline iterative compilation. Both a system that can repeatedly use the same inputs to quickly search the optimization space without affecting the user experience and a system where developers do not need to manually build and maintain sets of representative inputs. With existing approaches optimization decisions will either be suboptimal or require a level of engineering effort beyond the capabilities of most developers.

This thesis proposes a novel fusion of iterative compilation with a capture and replay system to realize an aggressive code optimization approach that is practicable for the mobile environment. With online transparent input captures, real user inputs are stored. And with offline replay-based iterative compilation, aggressive optimization is performed without ever negatively affecting the users. Finally, with crowdsourcing, these offline evaluation efforts are split amongst different users. The next section presents the challenges for a transparent input capture and replay mechanism.

1.2 Obstacles for transparent input-capture mechanisms

There needs to be a mechanism able to capture real user inputs online and then replay them offline under different optimization decisions. Not all of the existing approaches capture enough information for reconstructing a process in a robust manner. This is necessary for building an offline evaluation mechanism that uses the same input while allowing the selection of different code to carry out the execution. And from the approaches that do capture enough information, none is designed around a low-latency execution environment.

Such approaches are mainly divided into two categories. The first excessively intercepts the execution as a means of capturing the input at a fine-grain level. This significantly reduces the capture sizes but it causes unbearable overheads. The second captures input at a coarser-grain level to minimize the online overheads, however, the required storage explodes in size. Neither is good. At no time should a user experience slowdowns from a system whose ultimate goal is to improve performance. In addition, several captures of different code regions must easily fit on mobile devices, requiring little amounts of storage.

1.3 Contributions

This thesis enables aggressive code optimization on the highly-restricted mobile environment. It is input-driven, user-transparent, requires no developer effort, and can split the offline optimization search efforts amongst several users. These are accomplished through the below key contributions:

- Input capture mechanisms for C functions and Android application code regions. The proposed mechanisms capture inputs of targeted code at different granularity to address the challenges described in Section 1.2. The proposed approaches are lightweight enough to remain unnoticeable from the users. Captures happen infrequently and a single one is enough to drive code optimization. On average, the presented approaches that minimize the capture sizes require 2ms for C programs and 15ms for Android applications (Chapters 4 and 6 respectively), while the full capture Android approach requires less than 5ms. Regarding storage, capturing just the input pages on Android requires 5.06MB of storage on average. This amounts to only 6% of the application's total runtime heap memory. For the more lightweight C runtime, programs required storage between 100KB

and 200KB, which is at least two orders of magnitude less of the program's total virtual memory. The reachable object intersection captures of Android hot regions decreases the storage requirements by an additional 64%. Once optimization is finished the capture is discarded. Given such low online overheads that require just a small amount of transient storage, the proposed approach is suitable even for low-end devices.

- A replay-based evaluation mechanism of different code types and code versions that gracefully handles the issues of both offline and online iterative compilation approaches, as described in Section 1.1. Each replay operates in the same environment using the same input from an originally captured execution. This makes comparisons between different code transformations both sound and representative of real user usage. As they happen offline, when the device is idle and charged, any suboptimal or erroneously optimized evaluations are discarded without ever causing an inconvenience to the user. With dynamic profiling information, extracted offline, each evaluation is verified for correctness. Replaying evaluations offline allows setting a tighter control over the device environment. This, combined with robust statistical methodologies, makes execution noise manageable in the inherently noisy mobile environment.

- A novel iterative compilation system that aggressively optimizes programs for specific devices, without requiring any developer effort or having a negative impact on the user experience. It uses lightweight online input captures to perform a replay-based offline iterative compilation, which avoids the shortcomings and enjoys the benefits of the purely offline and online approaches, presented in Section 1.1. With a random search, C programs were improved by 29%. Android applications, however, had a very limited transformation space to begin with. A novel LLVM backend for Dalvik code was implemented to that end. On its own, the backend improved applications by 7%. When combined with a genetic search and a custom replay-based iterative compilation, applications were improved by 44%. Just 6% less of that was achieved with crowd-sourcing, which required just a fraction of the time as the search was performed collaboratively between several users.

1.4 Publications

Some of the methodologies and the findings in this thesis have been published to the below conferences:

- **Iterative compilation on mobile devices.**
  Paschalis Mpeis, Pavlos Petoumenos, and Hugh Leather.
  ADAPT, HiPEAC (2016), [MPE+16]

- **Developer and user-transparent compiler optimization for interactive applications.**
  Paschalis Mpeis, Pavlos Petoumenos, Kim Hazelwood, and Hugh Leather.
  PLDI (2021) [MPE+21]

- **Object Intersection Captures on Interactive Apps to Drive a Crowd-Sourced Replay-Based Compiler Optimization.**
  Paschalis Mpeis, Pavlos Petoumenos, Kim Hazelwood, and Hugh Leather.
  {under submission to TACO Journal}

The source code of the LLVM backend was published as open source software:

- **Experimental LLVM backend for Android applications.**
  GitHub, Apache2.0 license. [MPE21]

1.5 Thesis Structure

This section outlines the structure of the remainder chapters of this thesis.

Chapter 2 provides the required technical background and some fundamental concepts that have been employed throughout this thesis, for either developing solutions or evaluating them.

Chapter 3 surveys the relevant literature around the topics of iterative compilation, compiler optimization, as well capture and replay approaches.

Chapter 4 presents a novel approach that makes iterative compilation practical on mobile devices. Initially, it infrequently captures real user inputs online, without causing noticeable overheads to the users. Then, with a replay-based offline iterative compilation it searches for better code transformations through random search. It can readily optimize the most time-consuming function of a C program.

Chapter 5 presents the first backend alternative to the default Android one that is able to generate LLVM bitcode from Dalvik code. It extends the severely limited

optimization space of Android to unlock aggressive code optimization. Additionally, it automatically detects code regions that are worth optimizing. It supports multiple code regions as well multiple methods within those regions. Finally, it implements some Android-specific optimization passes.

Chapter 6 presents two input capture mechanisms that operate at a different granularity and support the more complex Android runtime. Captures are postponed when high-impact events are imminent. Code regions that can be accurately replayed are automatically detected. A novel replay mechanism was also developed, able to replay different code types. It also works well alongside memory-shuffling security mechanisms. It utilizes the LLVM backend (Chapter 5) for iterative compilation driven by a genetic search. With dynamic profiling data, extracted offline, it further optimizes the code and verifies its correctness. Finally, with crowd-sourcing, it splits the offline evaluation efforts among different users.

Chapter 7 summarizes the findings of this thesis, provides a critical review of the presented work, and outlines interesting future directions.

1.6 Summary

To enable aggressive compiler optimization for mobile devices any approach must evaluate different optimization strategies. Both offline and online approaches are unsuitable for the task. Offline approaches require representative user inputs, applications, and system/hardware configurations. Online approaches casually introduce significant overheads, faulty executions, or require a ridiculous amount of evaluations due to changing inputs. Existing approaches are either impracticable or significantly affect the user experience. The next chapter presents the relevant technical background, followed by a literature review. The following three chapters present methodologies developed to address the above problems.

Chapter 2

Background

2.1 Introduction

This chapter presents some fundamental concepts and details the relevant technical background to this thesis. Section 2.2 describes the compilation technologies and methodologies used by the rest of the chapters. Sections 2.3 and 2.4 present relevant information for the Linux and Android operating systems respectively. Section 2.5 describes the basic mechanisms behind capture and replay frameworks, and Section 2.6 explains the statistical methodology that was used by the proposed systems or by their experimental evaluations. Finally, Section 2.7 concludes this chapter.

2.2 Compilers and optimization

Compiler is a program able to transform the source code of another program to a new format. In many cases that format is a binary representation. Modern compilers consist of front-ends, middle-ends, and back-ends. A *front-end* performs syntax and semantic verification and creates an Intermediate Representation (IR) which is then passed to the middle-end. The *middle-end* applies optimizations on the IR and then passes its output to the back-end. The *back-end* lowers the code while applying architectural or processor specific optimizations and finally outputs code for a *target machine*. A compiler infrastructure might incorporate multiple front-ends and back-ends for supporting different programming languages and architectures respectively. A modern compiler might also provide a comprehensive Application Programming Interface (API) to facilitate third-party development of additional front-ends and back-ends, or the creation of custom passes used for analysis and optimization.

An *optimization pass* is a code transformation that is applied while the source code is being transformed from one format to another, aiming to improve particular aspects of the new format. It might require multiple passes over the source code, some of which could be just for collecting auxiliary information through code analysis. Transformation passes are most commonly applied on a program's source code as a means of improving its performance or for reducing its overall size. Due to the diversity in programs, a transformation can very well have a neutral or a negative effect on such aspects, which is why compilers provide a way to enable or disable particular passes through a known set of *optimization flags*. For a greater control over the transformations, these flags can accept parameters for fine tuning the internal compiler heuristics, e.g. by explicitly instructing the compiler to limit the maximum number of times that a loop is allowed to be unrolled. Well established compilers have a plethora of such optimizations.

Optimization flags are often clustered to different *optimization levels*. Usually there are 4 levels, ranging from `-O0` to `-O3`. Level `-O0` does not perform any code optimizations. Level `-O1` applies optimizations that commonly increase performance and reduce the binary size. Level `-O2` is a super-set of the first level as it applies some additional, more sophisticated code transformations. The outcome can be more effective at the cost of higher compilation times. Level `-O3` performs even more aggressive optimizations. There is a chance, however, that some of those passes might have a negative impact instead. There can be additional levels specific for code size reductions, or for faster mathematical operations (e.g., by trading-off floating-point accuracy).

The *optimization space* of a compiler comprises all the valid sequences that can be generated by combining flags that trigger analysis or transformation passes. A valid sequence generates a binary that once executed produces a correct outcome. A sequence may contain an arbitrary number of optimization combinations and their tuning parameters. The order within a sequence can be significant, which is known as the phase ordering problem [ALM+03; COO+02a]. Searching such enormous spaces is not something trivial. For example, the GCC compiler [GCC20a] has more than 200 optimizations, excluding the architecture specific ones. If assumed that the ordering is insignificant and that no additional parameters are accepted, such space has a staggering 2^{200} points. A compiler expert that has studied well the sources of a specific version of a program might be able to hand-pick from such space. This, however, can be quite expensive in both terms of money and time.

A multitude of techniques were developed that leverage a compiler's capabilities

to ultimately generate optimized code. *Profiling* is one such technique that gathers runtime information of a program. One way to extract profiling data of a program is to use code instrumentation. Such data might contain the number of times a basic block is executed, or measurements from *hardware performance counters* of the Central Processing Unit (CPU). These counters are built into the hardware and describe various events like the instructions executed or the cache misses. *Tracing* is another technique that gathers profiling information regarding the execution path that was followed in a program. It can be implemented either by code instrumentation or by sampling the call stack. The former leads to accurate findings, i.e., can calculate the exact number of times a method was invoked. As this comes with high overheads, the placement of instrumentation is quite important [BAL+94]. The latter incurs less overhead at the cost of decreased accuracy [WHA00], i.e., can estimate which methods the program spends most of its time executing.

2.2.1 Iterative compilation

Iterative compilation [AAR+97; COO+05; KNI+01; LIM+13] is an optimization technique that undertakes the task of exploring the code optimization space of a compiler. Its goal is to find the best optimization sequence for a particular application. The technique can focus on improving particular aspects of a program like the performance, the code size, or the energy consumption.

A search algorithm explores the *optimization space* of a compiler to discover better optimization strategies by constructing flag sequences. Those sequences are used to compile an application. Afterwards, the application runs and some execution data are logged. This procedure is repeated and the execution logs are compared between them so the technique can deduce the best optimization strategy discovered through search. Iterative compilation outperforms static optimization approaches as it benefits from dynamic information. To provide any guarantees for its findings though, it might have to exhaustively search the *optimization space*. Additionally, it incurs high overheads that might be prohibitively expensive, depending on the application domain. Last but not least, the noise in the timings can further complicate evaluations and comparisons.

The number of times the technique has to iterate depends on the search space. As it is quite large many approaches have manually pruned it [FUR+02; KIS+99]. Various search algorithms have been developed, from naive randomized approaches to sophisticated machine learning algorithms [AGA+06; LEA+09a; OGI+17]. Other

approaches have used profiling information to guide the search algorithms [BOD+98; SAM16].

Despite the efforts of the research community there are still many unresolved issues when it comes to the applicability and effectiveness of iterative compilation. The optimization search has to consider several factors. Some of which include the different hardware architectures, changes in the inputs of a program, or even changes in the program itself. A literature overview of iterative compilation frameworks can be found in Chapter 3. While the described frameworks tackle some of the known issues, none of them is applicable for the domain of mobile devices. In such an environment the programs are frequently updated, the user input can change frequently, and the high overheads of iterative compilation must not hog the limited processing and power resources of the device.

2.2.2 LLVM compiler infrastructure

LLVM is a compiler infrastructure that was originally developed at the University of Illinois [LLV20a]. It allows performing optimizations at various phases of a program, like during execution, idling, compilation, or linking. The optimizations are applied to a program through a set of passes [LLV21b] over its source code. Passes are classified into two main categories. The first one contains the *analysis passes* that gather information about the program. The second one, contains *transformation passes* that perform code optimization. Most of the transformation passes rely on information from a previously executed *analysis pass*.

LLVM supports a variety of front-end programming languages and back-end machine architectures. It also provides a rich set of compilation libraries, tools, and toolchains that allow the development of additional back-ends or front-ends. The tools used for the implementation of this thesis are described below.

Clang compiler: an LLVM front-end [CLA20] that supports the C programming languages family (C/C++, Objective-C/C++). *clang* is mostly compatible [CLA21] with the GCC compiler driver [GCC20a].

llvm-link tool: a linker at the LLVM bitcode level. It takes as an input several bitcode files and links them to a single output bitcode file.

opt tool: an interface to the LLVM analysis and optimization passes. It takes LLVM

bitcode as an input, runs one or more passes, and finally emits either the analysis results or the optimized bitcode.

llc tool: an assembler that translates the LLVM bitcode to a binary object file. This contains machine language for a particular architecture and (optionally) a processor. `llc` can apply architectural or CPU specific optimizations.

lld tool: a faster drop-in replacement linker for GCC's linker that can also be useful for tool-chain developers. It takes as input machine code from one or more binary object files, links them together, and outputs either a library (shared object) or an executable. It may optionally link the output object file against other libraries.

2.3 Linux kernel

Linux is a Unix-like OS that is Free and Open Source Software (FOSS). It is the leading OS on servers, runs on most mobile devices, and has the largest overall installed base. The remainder of this section describes some of its components that are relevant to this thesis.

`fork` and Copy-on-Write: `fork` is a system call that duplicates an original process. The original process is called the parent and the duplicated one the child. Both processes have separate Virtual Memory Areas (VMAs), which initially point to the same set of physical pages. A page is duplicated only when either of the processes wants to modify a common physical page. This is done through the Copy-on-Write mechanism, transparently and efficiently in the kernel space.

/proc interface: is a pseudo-filesystem that provides an interface (through common files) to internal data structures of the kernel. Some of these files are writable to allow controlling or tuning several of the kernel parameters. A process can access its own VMA mappings and their access permissions by reading the `/proc/self/maps`, where `self` points to the current process id.

Address Space Layout Randomization (ASLR): is a security mechanism that aims to mitigate exploitation against memory-corruption vulnerabilities. It was originally developed by Linux, but since then it was implemented in all major OSes. When loading a program for execution, ASLR ensures that its binary segments (including

heap, stack, and libraries) are placed into random VMAs, making them hard to predict during attacks.

2.4 Android mobile OS

Android is a FOSS mobile OS that runs on top of the Linux kernel. Its code base is known as the Android Open Source Project (AOSP). It consists of several modules that are all compiled together using the Soong Build System [GOO20f]. Android supports a variety of devices, mostly mobile, that come with limited processing and battery resources. The remainder of this section describes the parts of Android that were used or extended during the implementation of the systems described in this thesis.

2.4.1 Android Applications

Android applications are typically written using Java [ORA20] or the more concise Kotlin [JET20] programming languages. The original source code is compiled into Dalvik bytecode and stored following the DEX file format. DEX files can be executed on the register-based Dalvik Virtual Machine (DVM). They are compressed and archived into zip files named Android Application Packages (APKs), along with additional multimedia and metadata resources. An APK can then be distributed through online application stores, like the Google Play Store [GOO21b].

APK files might also contain pre-compiled shared libraries for particular architectures, written using the C/C++ native languages. Mobile developers can utilize the Native Development Kit (NDK) to create such libraries. The interaction between the native code and the Android RunTime (ART) is done through the Java Native Interface (JNI). The CPU intensive methods of an application are typically written into JNI as the default Android code is not as optimized. The NDK code is linked against the *bionic* library [GOO21a], which is an optimized version of the C runtime for Android. While it is recommended to use the bionic library with the NDK and Android applications, it is also possible to use it for external, standalone C programs as well (like the benchmarks used in Chapter 4).

2.4.2 Android Runtime (ART) and compiler

The Android RunTime (ART) is the environment that executes Android applications. During execution, each application is sand-boxed to ensure that it operates within its

requested permissions and also that it does not strain the limited computing resources. The Android compiler is responsible for verifying an application's bytecode and for translating parts of it to native code. It also generates any scaffolding code required for JNI calls. The remainder of this subsection goes into more detail on the internals of the runtime and the compiler.

2.4.2.1 Compilation strategy adaptations

During the course of this research the Android compilation strategy has undergone several dramatic shifts. The current version of the compiler is in-line with the proposed approaches: it may consult profiling data, which are incrementally generated, to restrict the amount of code that will be compiled. It supports both Just In Time (JIT) and Ahead Of Time (AOT) compilation.

In the early days, all of the code was interpreted by a DVM. A dex-to-dex compilation pass was applying a few pattern-matching improvements. Then, a JIT compiler was introduced [AND21a] that transformed the frequently executed code traces from bytecode to architectural-specific code. Essentially, it was doing line-to-line translation to machine code instructions for select code traces. Those were limited within basic blocks, which is a significant limitation in code optimization [COO+86b].

Quick backend.
Full AOT compilation support was introduced with the *quick* backend. The process was initiated with the `dex2oat` compiler driver. On application installs or updates, `dex2oat` would AOT compile all of the bytecode found in its APK. The resulting machine code would be packed in a file, called *OAT* [AND21c], which adheres to the Executable and Linkable Format (ELF) file structure. This has increased performance due to native execution instead of interpretation. However, by keeping the original bytecode and also translating all of it down to architectural specific machine code, it required significantly more storage space. Keeping the bytecode was necessary for debug metadata, as well interpretation in special cases. Another disadvantage was the extra time needed for AOT compiling everything. This was especially noticeable during the Over The Air (OTA) software updates, where a full recompilation of all device applications was triggered. This operation casually needed several minutes, close to an hour for some cases. The default compilation strategy was to compile everything. On low-end devices, the compiler had to restrict the amount of the generated native

code in order to keep supporting them. This was reducing the high compilation times and conserving internal flash memory space. The quick backend did not apply any sophisticated transformations. It was simply expanding the line-to-line translation of the JIT compiler beyond basic blocks.

Optimizing backend.
Then a more sophisticated backend was introduced, named *optimizing*. It uses the same calling convention with the *quick* backend. As it required more time for compilations, *optimizing* was eventually restricted to methods that were found through a profiler. The intermediate representation of this backend is named `HGraph` and there is no an API for it. An example IR is shown at Listing 2.1.

This approach became in line with the compilation approaches presented in Chapters 4 through 6. That is, to put more compilation effort to the methods that dominate execution time.

```
1 BasicBlock 0, succ: 1
2   0: ParameterValue [3]
3   1: SuspendCheck
4   2: Goto 1
5 BasicBlock 1, pred: 0, succ: 2
6   3: InstanceFieldGet(0) [4]
7   4: Return(3)
8 BasicBlock 2, pred: 1
9   5: Exit
```

Listing 2.1: The HGraph IR of an instance method that returns a field of its class. The field *getter* is invoked at line 6. Some additional code is generated in this case, like the exit block (line 8-9) that never gets reached, and a check call for operations that require suspending the execution, like the garbage collection (line 3).

2.4.2.2 Method execution

ART may execute a method of an application using interpretation or native execution. For the latter, the code could either be Java Native Interface (JNI) code, or code that was generated either by the quick or optimizing backends. How a method is executed depends on several factors, some set ahead of time and some at run time. These include compilation settings, debug configuration, or runtime information. In ART, the class representing an Android method object is named `ArtMethod`.

2.4.2.3 Runtime optimizations

ART pre-initializes a set of relevant libraries and a heap of objects that are extensively used by most Android applications. Those are stored in special binary files named `boot.oat` and `boot.art` respectively. Android leverages memory optimizations on Linux to share common resources to multiple applications. When a device boots, a special process named `zygote` initializes the commonly used objects or classes. Afterwards, when applications are launched, `zygote` is forked and specialized. This saves both memory and initialization time.

2.5 Capture and replay

Capture and Replay (CR) frameworks have two fundamental phases. During the first phase, any relevant state of a program is captured to permanent storage. A *snapshot* is another term for a *capture*. It contains any process state that is needed for accurately replaying the original process. On the program's original execution, that process state resided on the main and processor memory. Regarding the main memory, the state could either be in the user or kernel level spaces. Regarding the processor memory, that state is the register values. The second phase, called a *replay* or a *restore*, loads the stored state of a previously captured execution. This is done by filling the main memory areas, as well the processor's registers with the captured data. Then it proceeds by re-executing the code of the captured program.

There are many use cases for a CR system. One is for process migration [JAN+05]. First, the state of a process is stored on disk before it terminates. Then, the captured state is transferred to another machine and is loaded into a new process. The ability to replay a process is also widely used in debugging and testing systems. If a bug can be reproduced by a replay, then the error's source can readily be spotted. Some systems employ CR to increase fault tolerance by preventing bugs from reoccurring. Other systems that automate testing use replaying to stress test programs [LEI+09; ORS+05]. Chapter 3 describes several CR systems proposed in the literature. Nonetheless, none have tried to fuse CR with aggressive compiler optimization, in a way that works well on mobile environments.

2.6 Statistical methodology

This thesis uses various statistical techniques to cope with the inherently noisy environment of mobile devices. Background services, caching mechanisms, CPU scheduling, are some of the factors that contribute to a noisy environment. Such statistical techniques have not been employed only during the experimental evaluation, but were also used during the optimization search of the approaches presented in Chapters 4 and 6.

When a program is executed the proposed approaches log an *observation*. Many of such *observations* constitute a *sample*. Many samples, that cover every characteristic that one would like to understand, constitute a *population*. The population size is commonly denoted with an N.

The *mean* (μ) of the sample is the sum of all the sample's observations divided by the sample's size. The *true mean* is the mean of the *distribution*. The *variance* of a sample is the average of the observations' square differences from the mean. It shows how spread, from the mean, the observations are. The *standard deviation* (σ) is the square root of the variance and it is shown in Equation 2.1. The population size is N and x_i are the individual values from the population. If a sample follows a *normal distribution* it means that most of its observations are around the mean, while the rest of the observations follow a symmetrical fashion. The *histogram* of a normally distributed sample is known as a *bell curve* 2.1.

$$\sigma = \sqrt{\frac{\sum_{(} x_i - \mu)^2}{N}} \tag{2.1}$$

2.6.1 Outliers

An *outlier*, is an observation that it is widely separated from the main body of the distribution. In a mobile environment several factors might cause an *outlier*. Such observations can significantly skew the *mean* of the sample from the *true mean*. To remove any outliers the *interquartile range* method was used, which is shown on the Equation 2.2 and explained below.

$$\begin{aligned} Lowerfence &: Q1 - c * IQR \\ Upperfence &: Q3 + c * IQR \end{aligned} \tag{2.2}$$

The observations of a distribution can be grouped into 4 equal sections, called *quartiles*. The *lower quartile* (denoted as $Q1$) is the point where it is greater than

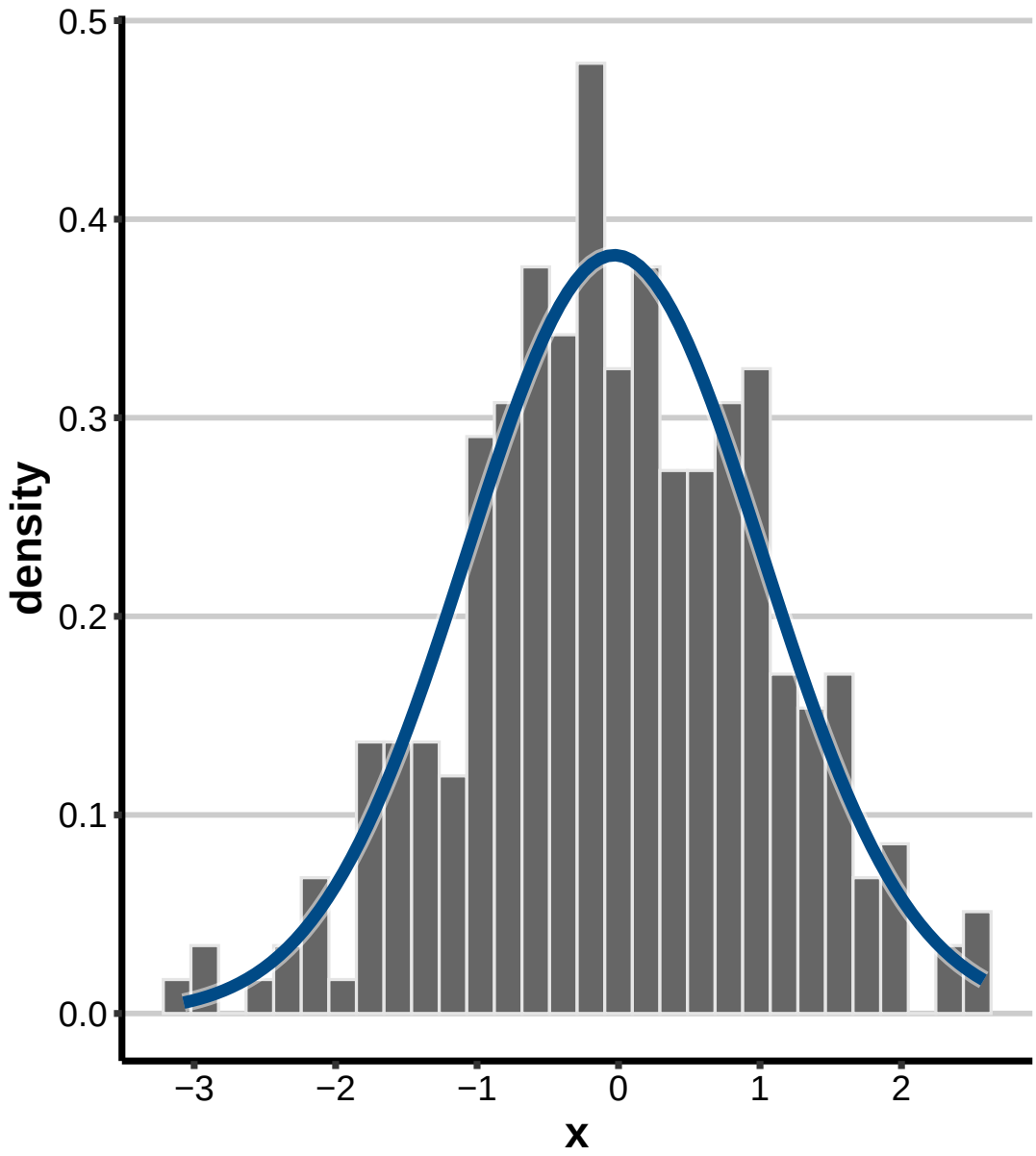

Figure 2.1: The histogram of a normally distributed sample with a density curve drawn on top of it. For normal distributions the density curve resembles a bell, as most of the observations lie around the mean.

the 25% of the sample's observations. Similarly, the *upper quartile* (denoted as *Q3*) is the point that it is smaller than the 25% of the sample's observations. Some of the observations that are smaller than the *Q1* or greater than the *Q3* can potentially be outliers. To detect outliers, this technique sets two fences: upper and lower. Any observations that exceed them are considered as outliers. The *interquartile range* value (denoted as *IQR*) can be calculated by subtracting the *Q3* with the *Q1*. The constant value, denoted as c, was set to 1.5.

2.6.2 Confidence intervals

Confidence intervals are calculated from the observations of a sample and they try to give a reliable estimate for a population's parameter. The occurrence frequency of the parameter within the confidence intervals can be modified by setting a *confidence level*. On applicable experiments, two-sided confidence intervals were used. These are lower and upper fences to the mean of samples, as shown by Equation 2.3. $\bar{X}$ denotes the sample mean, and it adds and subtracts the *margin of error*. The critical value (ζ) is set according to the desired confidence level, the *degrees of freedom* ($D_f = N - 1$), and alpha (α) that is computed by subtracting the confidence level from 1.

$$\bar{X} \pm z_{\alpha/2} \frac{\sigma}{\sqrt{n}} \tag{2.3}$$

2.6.3 Statistical tests

For statistically sound comparisons the *two-sided Student's t-test* was used. It is a statistical test that decides whether the means of two different samples are significantly different. A *null* and an *alternative* hypothesis have to be set. The *null hypothesis* is the one that it is desired to be rejected, i.e. the means of the two samples are the same. Rejecting the *null hypothesis* means that the *alternative hypothesis* is accepted, i.e. the means of the samples are significantly different. Also a *confidence level* has to be set. This test requires the samples to follow a *normal distribution*. The minimum sample size required by the test is 2 observations. The tests that were used in the proposed approaches did not assume equal variances of the samples.

Rejecting the *null hypothesis* does not necessarily mean that the two means are equal. To argue equivalency two *one-sided Student's t-tests* have to be conducted. This answers whether the difference of the two means is smaller than a constant value. That value is set by the user and it is called the *indifference region*. The *null hypothesis* of the first test claims that the difference of the two means is smaller than the *indifference region*, while the *alternative hypothesis* supports the opposite. The *null hypothesis* of the second test, claims that the difference of the two means is greater than the *indifference region*, while the *alternative hypothesis* supports the opposite. If both *null hypotheses* are rejected, then there is statistical evidence that the two means are equal. This is because the accepted *alternative hypotheses* will support that the difference of the two means falls within the *indifference region*. The confidence level of the *tost* test was set to 90%, since two *one-sided t-tests* with 95% *confidence level* were used.

2.7 Summary

This chapter provided the technical background and some fundamental concepts that have been employed throughout this thesis. Those relate to compiler optimization techniques, the LLVM compiler infrastructure, relevant areas of Linux and Android OSes, to capture and replay basic concepts, and finally to the employed statistical methodologies. The following chapter presents a literature review that is relevant to this thesis.

Chapter 3

Related work

3.1 Introduction

This chapter presents a literature review that is relevant to this thesis. It is focused on two distinct research areas: iterative compilation, and capture and replay systems. Section 3.2 surveys the research field of iterative compilation and code optimization approaches. Section 3.3 reviews research on capture and replay systems. Section 3.4 provides the concluding remarks of this chapter.

3.2 Iterative compilation and optimization systems

This section briefly describes several iterative compilation and code optimization approaches. These include exhaustive search, space-pruning techniques, simple random approaches, genetic algorithms, online approaches, and machine-learning approaches.

3.2.1 Early approaches

Early iterative compilation systems [GOR+02; LEE+99; MAS87] targeted specific embedded applications. For those systems, the target architecture, the program's source code, and the input are all fixed. Better code is discovered with either an one-time exhaustive optimization search or with a manual hand-tuning performed by a compiler expert. These are quite costly approaches by today's standards, in terms of both time and money. Modern compilers come with a myriad of optimizations and heuristics, while there are several target architectures and processor designs.

3.2.2 Accelerating search and evaluation through space pruning

Then, several space-pruning approaches were developed to tackle the ever-growing code transformation space. Some approaches focused only to particular optimization categories (e.g., loops), while others tried to exclude some of the areas speculated as non-beneficial.

An approach by Fursin et al. [FUR+02] performs an isolated search for 3 distinct loop optimizations: padding, tiling, and unrolling. Once the search concludes, the best findings for each optimization return. Additionally, it generates optimization sequences from the findings, as a means of investigating any possible inter-dependencies between the 3 optimization passes.

Some other approaches rely on static analysis data to reduce the required time for searching. Such data allow pruning the less rewarding areas of the transformation space, risking however the exclusion of the optimal solution [SAM16]. An approach by Bodin et al. [BOD+98] visits only particular points in space as it favors the ones with higher performance. Kisuki et al. [KIS+99] have presented three search algorithms, the first of which is similar to the previous approach [BOD+98]. It applies a coarse grid over the space and iteratively focuses on the rewarding transformations. The second one chooses a random point in the space and samples the neighboring points. The last one is a purely random search. In later works [KIS+00], the authors have improved the grid-based algorithm by focusing only on areas that are within acceptable distances from the best findings. Still, the search concerns no more than 3 loop optimizations. Quoting the authors, for embedded programs the offline compilation and evaluation overheads will *"be amortized over the number of systems shipped and the lifetime of the application"*. On mobile devices, however, offline evaluation is not an option unless the application developers provide a set of neatly packed representative inputs. In an online scenario, even a single slow or erroneous evaluation can be detrimental to the user experience. To make matters worse, mobile application code is updated quite frequently. Given such a short lifetime of application code versions, these side-effects will have to be repeatedly endured by the users.

COSPpp [LIM+13] is another space pruning technique that searches for optimizations based on the similarities between programs. Initially, a subset of the optimization space and a set of programs is used to generate some profiling data. These consist of hardware performance metrics of two versions of each program. One that is not optimized and another that is optimized with randomly chosen transformation sequences.

These metrics have proved in the past that they can accurately identify the performance bottlenecks of programs [UHS+08]. When a new program is encountered, a similarity algorithm compares their performance metrics with the profiling data to drive the selection of transformations. However, if the initial set of programs or the subset of the optimization space are not representative in regards to the newly encountered programs, the selection process would be flawed.

Other approaches have tried to quickly achieve decent but not optimal performance. The work of Parello et al [PAR+04] aims at steady performance gains across iterations. A decision tree is built, to which runtime statistics are then fed to drive a transformation search. Hardware performance counters have been used to extract some statistics, in a similar way with other systems [LIM+13; LU+04; PAR+11]. On each iteration the extracted data improves. However, building a decision tree for a particular processor requires significant effort. The authors have been hand-tuning a set of benchmarks for several months.

The approach by Purini et al. [PUR+13] aims to create more compiler optimizations classes than the predefined ones. For example, some compilers have a handful of standard optimization levels, like the `-O1` to `-O3` found in GCC. The authors argue that having a bigger set of classes would yield better results. The number of classes in the set should be large enough so that every newly encountered program will have at least one good sequence. Similarly with other works [LIM+13], an initial set of programs and transformations is picked. Then, by using a similarity metric and sequence clustering algorithms, new transformation-sequence classes are generated. Those can then be searched without a significant cost. While the findings outperform the predefined optimization levels, they still lag behind fine-tuning approaches.

Cohen et at. [COH+05] argues that significant performance gains are the result of complex transformation sequences. To discover more sophisticated sequences they render the optimization space using a *polyhedral representation*. This simplifies the traversal, as the polyhedron encodes the order of code transformations in the representation. Any invalid sequences are ruled out. The authors also try to untangle the interference between different transformations by dividing them into categories based on the components they modify. These components include: the iteration domain, the access functions, the data layout, and the scheduling of individual instructions. On the down side, the proposed approach relies heavily on manual intervention. For example, the detection of the targeted code regions, or the correctness evaluation of the tested transformations are performed manually.

Other approaches [CHE+12; FAN+15] have exploited the Map-Reduce paradigm to split the evaluation workload in order to accelerate the optimization search in data centers. A main node instructs several working nodes to evaluate different transformations. The findings are reported back to the main node in order to advance the search process. One problem is that the message interception between the main and the worker nodes can incur high storage overheads for mobile devices. Another, is that both approaches are applicable only to workloads that are known to operate well in a Map-Reduce environment.

3.2.3 Genetic Algorithm approaches

To progressively avoid slow code transformations several self-adapting algorithms have been developed. Almagor et al. [ALM+03] utilized data after exhaustively searching the space as a means of understanding its properties. During the analysis they discovered several *local minima*, with most of those falling within a 20% of the *true optimum*. Therefore, they developed a GA, which they improved in their later works [ALM+04] by removing a handful of weak genomes. They also employ a *hill climbing* algorithm, which performs shallower local exploration but with more random points in an attempt to escape the local minimas. Finally, they have explored a *greedy algorithm* that views the transformation space as a Directed Acyclic Graph (DAG) and descended from the root towards nodes with higher fitness values. From the same research group, Cooper et al. [COO+99] have also employed a GA for reducing the generated code size. In later works [COO+02b], the authors have improved upon their hill climbing approach by applying an *adaptive sampling* algorithm. This approach takes random samples from the space, evaluates them, and keeps the statistics. Apart from code size, they also consider performance and power consumption.

VISTA [KUL+03] is a system that provides an interactive interface for improving embedded systems using different exploration algorithms. The first is an exhaustive search, which is applicable only with sufficiently small search spaces. The second is a GA that uses the number of executed instructions as its *fitness function*. As it is explained later in this section, this is not always indicative of better performance. The last is a permutation search that keeps the length of a sequence static and rearranges the different flags.

Any approach described in this chapter so far, requires representative inputs in order to be performed offline. Online evaluation approaches on the other hand, utilize

real inputs that are user representative by definition. Nevertheless, they bring a whole new set of problems as explained in the next subsection.

3.2.4 Online approaches

Some online approaches were also developed to tackle some of the unresolved issues. Fursin et al. [FUR+05] utilizes online phases for speeding up the evaluation of different transformations. A phase is a time period where the performance for a particular trace of code remains stable. Therefore, it is both feasible and fair to compare multiple transformations within a single phase. Initially, a simple algorithm captures the phase patterns of a program. Then a 3-stage procedure predicts the start of a phase, evaluates some transformations, and finally verifies that the evaluations happened within the predicted phase. The last 2 stages can be repeated as long as the program remains at a particular phase. While such an online system would naturally tackle the input variance, several points of the space can severely degrade the performance or even crash the execution. This is why a severely limited subset of the space was used in this system. Lastly, the main focus of this approach is to accelerate the optimization search. In contrast, the ideas described in this thesis focus on the performance improvement of the input programs.

ADORE [LU+04] is another system that exploits a program's online phases. In contrast with the previous approaches, ADORE does not iteratively compile. Instead, it utilizes hardware performance counters to apply its own optimizations: dynamic register allocation and cache pre-fetching. Its phase detection algorithm continuously reads measurements from various performance counters in a separate thread from the main one. When major phase changes are detected, *dynamic optimization* is applied to the relevant code traces. As both the phase detection and the dynamic optimization happen online, this approach incurs considerable overheads for a mobile environment. In any case, several factors can arbitrarily affect the performance stability in a low-latency interactive environment, making both phase-prediction approaches [FUR+05; LU+04] unsuitable for mobile devices.

Dynamo [BAL+00] is another dynamic optimization approach that operates at runtime. It interprets binaries to dynamically generate optimized native streams. Due to its online operation, any incurred overheads should not overshadow its potential performance gains. Therefore, the optimization effort as well the amount of code it processes is severely limited. ADAPT [VOS+01] is a competitive approach that tries to decouple

the overheads from the critical execution path by offloading optimization to idling processors or to background threads. In a mobile environment this still consumes precious processing and energy resources. On top of that, ADAPT relies on manual heuristic tuning.

3.2.5 Machine learning approaches

Several researchers [AGA+06; CAV+07; CUM+18; FUR+09; KUL+12; LEA+09a] have employed machine learning to speedup the space exploration, improve the findings, or both. These approaches improve over the GA approaches, as those have to start afresh for each new program or even segments within a program. The acquired knowledge is encoded into a model that is inexpensive to operate on newly encountered programs, while achieving good results. On the other hand, machine learning approaches require lengthy offline training phases over a set of representative applications and inputs. Milepost GCC [FUR+08] is a system that integrates machine learning capabilities to the well-established GCC compiler infrastructure. It aims to automate the construction of optimizing compilers on general-purpose programming. It focuses on performance, code size, and energy efficiency.

Cavazos et al. [CAV+07] used *logistic regression* for predictive modeling. Their model maps hardware performance counters to beneficial code transformations. It was found to be particularly effective for applications that had a complicated control flow. As other researchers noted [ALM+03; COO+02a], the order as well the repetition of particular flags or flag sequences can alter the quality of any subsequently applied transformations.

Park et at [PAR+11] have employed *linear regression* and *supervised learning* to predict good optimizations for programs. During training, multiple evaluations are performed on a set of different input programs to generate three models. On each evaluation, a random optimization sequence is used and its performance statistics are collected. Those come from around 30 hardware performance counters. The first model outputs a prediction regarding the performance of each transformation. The second one also outputs a prediction given an input transformation sequence complementary to the performance counters. The third model considers two transformations and predicts the best performing one. Furthermore, it operates on a restricted transformation space (e.g., 45 boolean flags) when compared to highly optimizing compilers.

Agakov et al. [AGA+06] have attempted to accelerate the optimization search by

focusing on areas that are more likely to be beneficial. An initial exhaustive search is performed on two different platforms as a means of characterizing the space. The proposed approach uses two models. The first is a simple model that considers the product of the probabilities of all individual transformations in a given sequence. The second and more sophisticated model utilizes a *stationary Markov chain*. This considers any side-effects or inter-dependencies between different optimizations, as the probability of applying a transformation depends on the previously applied ones.

Ashouri et al. [ASH+17] have tried to mitigate the phase ordering problem by organizing passes into multiple clusters, instead of trying to find their optimal order by considering them individually. This significantly alleviates complexity. Applications are characterized with a vector of dynamic features and are independent from target architectures. This vector is encoded into a conventional fixed-length space which allows traditional machine learning algorithms to be applied.

Kulkarni et al. [KUL+12] presented an approach more directly focused on solving the phase ordering problem. It employs *neuro-evolution* to construct an Artificial Neural Network (ANN) from features found in a program's methods, and uses it to predict the most efficient optimization. Subsequently, the features are updated and fed again to the ANN in order to predict the next optimization point.

Leather et at. [LEA+09a] argue that hand-crafted features are problematic. Some features might be irrelevant while some others can be effective regardless of whether they are combined with others. Additionally, the effectiveness of a particular set of features can vary between different machine learning algorithms. Their solution uses a hybrid *grammatical evolution* and *genetic search* to generate a set of features from the Abstract Syntax Tree (AST) of a program. The system iteratively builds a list of good features and utilizes machine learning for their evaluation. The training was focused on loop unrolling and lasted 2 days.

Machine learning approaches generally require big training data-sets. *Predictive modeling* in compiler optimization commonly used only a few well-known benchmarks. To significantly increase the amount of input programs, Cummins et al. [CUM+17b] have mined open source repositories and trained a Deep Neural Network (DNN) for constructing artificial benchmarks. Then, the authors have trained a model over the artificial benchmarks, which outperforms approaches that use smaller but real data-sets. In later works, Cummins et al. [CUM+17a] proposed DeepTune. It is an optimization framework that employs DNNs to completely bypass a feature extraction phase. This manual phase required human experts to select and tune relevant features. DeepTune

instead, learns directly from raw source code. Therefore, it is agnostic to compiler, platform, or optimization problems. It was applied for heterogeneous device mapping and GPU thread coarsening.

While machine learning approaches significantly improve certain aspects of previous approaches, they require lengthy training phases that span from days to months. And this, while most of them consider a fraction of the transformation space. In addition, offline training phases assume representative inputs.

3.2.6 Other approaches

Any approach that is based on *iterative compilation* (presented in Section 2.2.1) needs to repeatedly compile, execute, and time code under different optimizations. Some optimizations, however, might have substantially different runtime performance than others. For such cases, a smaller number of evaluations suffices without compromising statistical soundness. For the rest of the cases, where two code versions might exhibit similar performance, an increased number of evaluations might be required. Raced Profiles [LEA+09b] is a statistically robust approach that leverages this property and utilizes dynamic sample sizes. The benefits against approaches that use fixed execution samples are twofold. First, sub-optimal code versions are quickly identified and eliminated. As shown in Chapters 4 and 6, such versions may have a detrimental effect on performance. Second, the sample sizes of fast and similarly performing versions are increased, which improves the statistical confidence.

cTuning [FUR+09] is a framework with the aim of re-using auto-tuning tools and crowd-sourcing of several code sources and meta-information. Its main objective is to enable different researcher groups to re-use existing optimization knowledge in a unified and verifiable format. This will ultimately help them in understanding correlations between similar applications, inputs, and hardware.

ACME [COO+05] have employed *virtual executions* as a means of reducing the overheads of iterative compilation. A virtual execution runs the program once and then makes performance predictions of different transformations without re-executing the code. The initial run stores a profile of the program that contains the number of instructions of each basic block, as well its execution frequency. Then, under the assumption that differently compiled programs with the same input will follow the same trace of basic blocks, it calculates the total number of instructions that will be executed. The transformations that are predicted to perform best are the ones that

result in the lowest number of instructions. However, this is not the case for many optimizations. As an example, loop unrolling increases the number of instructions in an effort to reduce loop-related overheads. Another drawback is that ACME cannot make reliable predictions for transformations that modify the Control Flow Graph (CFG).

3.3 Capture and replay systems

Capture and replay systems is another well studied area with highly active research and industrial communities. There is a wide variety of applications including automatic test-case extraction, facilitating reproducible and cyclic debugging, process migration, speculative parallelization, and optimization. This section briefly summarizes several systems from different domains and emphasizes on the components that are most relevant to this thesis.

3.3.1 Approaches leveraging additional develop-time context

Some approaches leverage the additional data and flexibility available at the development and debugging stages of an application. Leitner et al. [LEI+09] proposed a technique to automate test case extraction through a CR system. In contrast with competitive approaches [JHA+13; ORS+05; PAT+10; STE+00; XIA+12; XU+07a], an application's state is captured only at the point of a crash. During normal execution, a relevant stack frame is copied on each call-site. If the execution is successful, this shadow copy of the stack is discarded. Otherwise, any heap elements referenced by the stack frame are deep-copied and finally all duplicated data are serialized to permanent storage. This approach minimizes online overheads as the bulk of the capture operations only happen after a crash has occurred. The captured data can be restored afterwards to reproduce the software crash as a means of accelerating the debugging efforts.

Xia et al. [XIA+12] have proposed a technique that replays a program at an intermediate stage. This enables correlations of program executions at different stages of the compilation process. It facilitates the compiler writer in detecting software bugs introduced by any potentially unsafe parallelization transformations. An alternative approach by Hursey et al. [HUR+10] utilizes thread suspension to accelerate iterative debugging of High-Performance Computing (HPC) applications. It provides the user with consistent debugging data over several intermediate states through checkpoint and

restart operations.

A major obstacle of any development time approach is their requirement to execute the input application under a special developing or debugging environment. This additional context is not always available when optimizing applications released by third party developers to online software stores. When it is available, it can severely interfere with several code optimizations, which is the use case of the CR approaches described in this thesis.

3.3.2 Intercepting code at the variable-level

PIN [LUK+05] is a dynamic instrumentation framework that is portable, transparent, and architecture-independent. It copes well with mixed code or data, variable-length instructions, and dynamically loaded or generated code. Several systems have been developed since that are based on PIN. One such system is PinPlay [PAT+10], which enables input capturing. It was designed for selecting and analyzing simulation checkpoints and for facilitating reproducible debugging. By operating at the variable-level, it minimizes the information that is captured to the absolute minimum that is necessary for reproducing any non-deterministic events. This comes at a high cost. Every single memory access needs to be intercepted. Afterwards, a replayer tool recreates the previously captured program state to enable offline analysis of large program executions and repeatable analysis of parallel programs. In later works, Patil et al. have proposed ELFie [PAT+21], a framework that precisely captures code regions and uses them to create standalone executables. This accelerates analysis over long-running programs, as the efforts can be focused only to specific portions during the replayed execution. Once generated, the analysis can be used for simulation, native performance analysis, and dynamic program analysis. ELFie also attempts to replay filesystem I/O, by allowing write operations on file replicas without modifying the original files.

SCARPE [JOS+07] is a system that uses CR to automate extraction of real-user test cases. It requires explicit annotation of an application's subset that is desired to be captured. During captures, the annotated code will be instrumented to log its interactions with the rest of the code. Four types of method invocations are logged: when external or internal functions are called or returned. Additionally, four data interactions are logged: read or write on external or internal data structures. During replay, the event log is re-executed in the original order. For any external code there is scaffolding code that mimics any relevant behavior. Guoqing Xu et al. [XU+07a] proposed another

variable-granularity capture approach that operates purely at the language level and was built specifically for Java. It improves over SCARPE as it employs static bytecode analysis to identify and select application code subsets for capture and replay.

Approaches that rely on variable-level instrumentation can significantly decrease the amount of captured data. Unfortunately, these savings trade off performance as they commonly introduce considerable overheads. Such approaches are rendered impracticable under several scenarios, including ours that aims to be transparent on the already limited, in terms of processing and battery sources, mobile environment.

3.3.3 Intercepting framework or peripheral I/O events

VALERA [HU+15] is a system targeting input streams on Android that does not require kernel modifications. It operates on the Dalvik bytecode that is available through the APK files of applications, therefore it does not require access to the original sources. It supports several I/O devices such as the GPS sensor, the camera, or the network. It relies on manual instrumentation and it is based on an outdated version of the Android runtime that interprets all of the bytecode. This conceals to some extent the high instrumentation overheads. In more recent versions of ART, where AOT compilation realizes native executions, VALERA would incur significant slowdowns.

RERUN [GOM+13] is a time-sensitive record and replay system on Android that does not modify the Android platform. It focuses on the peculiarities of touch-based interactions. For example, it captures a touch gesture as a single contiguous action, rather than as several distinct touch events. It can also replay low-level sensor events. While it can be used to reproduce bugs or fast-forward executions, it does not store enough information to recreate a process execution environment, even a partial one. This makes RERUN unsuitable for standalone replayed executions and unfit as an evaluation mechanism of different code transformations.

jRapture [STE+00] is another system that aims to improve software quality by extracting real-user test cases. It captures several interactions between a Java program and the system including inputs from the Graphical User Interface (GUI), files, and the console. Then, it replays those interactions on each captured thread in a time-sensitive manner. It is heavily coupled to a particular Java implementation as it modifies the Java API. This modification allows the authors to log certain actions during capture and replay them afterwards. Similarly with the approach presented in Chapter 6, jRapture additionally allows an application to be specially instrumented for offline, dynamic

profiling during replays.

Jha et al. [JHA+13] has proposed another instrumentation-based approach for reproducing crashes on Android applications. It is focused around the main architectural blocks that are part of an application's life-cycle. Those are: Activities, Services, Content Providers, and Broadcasts. This event-based approach serializes any in-memory recorded events to permanent storage when a crash occurs.

By intercepting framework or peripheral I/O events, the above approaches can support specific cases of non-determinism that the proposed solutions in this thesis have chosen not to. While such capabilities sound compelling, I/O bound code generally does not respond well to compiler optimization, which is the goal of this thesis. In addition, such event recording requires a significant amount of specialized instrumentation, which can affect the users' interactions in the low-latency mobile environment.

3.3.4 Using code-slicing to accelerate replays

Zhang et al. [ZHA+06] have employed dynamic slicing of long running programs to accelerate the discovery of runtime bugs. The proposed approach prunes irrelevant captured events, which speeds up replaying without affecting the accuracy of reproducing a failure. The dynamic slice of a variable is computed by finding all the executed instructions that have contributed to its value. DrDebug [WAN+14] is another PIN-based framework. It enables interactive debugging on multi-threaded programs and it improves upon competitive approaches as it automatically identifies slices within captured code regions. It initially performs a static code analysis to generate the CFG and finally extracts a slice by computing the immediate post dominators.

Xu et al. [XU+07b] proposed another technique that operates purely at the language level through instrumentation, requiring no Java Virtual Machine (JVM) or OS modifications. With static bytecode analysis, regions that can be accurately replayed are initially identified. Then, a user manually configures a checkpoint for capturing. Two bytecode versions of the program are generated: the checkpointing version that stores relevant runtime information, and the replay version that restores the captured information. To set up the input of the replay, this approach follows the control flow from the beginning of the program, eliminating any statements outside the execution path. Additionally, it replaces predicates at *control-decision making points* with boolean values of the expected outcome. Despite such slicing, there could be scenarios where this approach can suffer from increased setup overheads.

Slicing approaches focus only on how to accelerate replays without affecting the final execution outcome. Therefore, instructions are being removed or replaced to achieve that. A replay-based evaluation mechanism for different optimization decisions should reflect, as accurately as possible, the efficacy of decisions when those are applied outside of a replay sandbox. The mechanisms presented in this thesis do not alter the input code, since such level of intervention would certainly affect the replayed evaluation accuracy. Nevertheless, evaluations are still accelerated as replays start and end at distinct code region boundaries.

3.3.5 Specialized hardware and kernel-space approaches

RASP [HER+11] is a system that focuses on speculative parallelization. In contrast with competitive systems [DIN+07; KEL+09; TIA+08], it requires a specialized hardware for input capturing that is capable of identifying operation violations. When this happens, a rollback operation to a valid state is performed.

CRAK [ZHO+01] is a CR system that has access to low-level OS operations without requiring application or runtime modifications. It relies, however, on support for dynamic loading of kernel modules. It can work with parallel programs, as it can restore the child and parent relations between several processes. It can also restore open network connections.

Zap [OSM+02] is another approach based on loadable kernel modules that offers transparent process migration with low-overheads. It provides a thin virtualization layer on top of the underlying OS. This enables a mapping between the participating processes and the utilized OS resources. The migration is realized with a checkpoint/restore mechanism that intercepts system calls at the virtualization layer. Zap additionally allows a complete suspension of a process and its resumption at a later stage. Cruz [JAN+05] extends Zap by adding replay capabilities for shared memory, threads, inter-process communication, and sockets. It additionally offers advanced network capabilities, such as restoring the full network state, replicating processes while modifying IP addresses, and maintaining connection states for several remote clients.

While the ability to replay low-level I/O is impressive and quite useful under several scenarios, when it comes to real inputs in interactive mobile applications it can cause catastrophic consequences to the users. One such example is replaying sensitive operations, like bank transactions, or tunneling private information through the network.

3.3.6 Using speculative threads or processes

Several capture and replay systems were developed for solving problems related to execution parallelism. Ding et al. [DIN+07] have proposed a system that aims at improving coarse-grain parallelism with correctness and efficiency guarantees. It relies on a manual phase where code regions that can benefit from parallelism are explicitly marked by the developer. Then, a lead thread executes the program without performing any speculations. There are also additional speculative threads that execute the code under potentially unsafe parallelization. The outcome of the speculative execution has to pass through a verification process. When this step fails, a complete rollback to a correct, known thread state is performed. The applicability of this approach depends on the size of the accessed data, instead of the amount of code that is parallelizable. The mechanism for storing such data is similar to the one presented in this thesis. It is facilitated by revoking the permissions of the memory pages used by the lead thread. When the thread wants to access the memory, a page fault is raised. It is immediately handled by storing at the call-site the relevant pages and restoring their permissions, so the thread can successfully continue its execution.

Kelsey et al. [KEL+09] has also proposed a similar system for enabling unsafe parallel optimizations. It slightly improves upon the previous approach as it supports implicit marking of the parallel regions. It still requires custom code to be injected to perform correctness verification. The input storing mechanism operates at the process level. Other than that, it is similar to the approach by Ding et al. [DIN+07].

Copy or Discard (CorD) [TIA+08] is yet another CR framework aimed at speculative parallelization. It uses two distinct thread groups, one for a non-speculative thread, and another for some speculative threads. It divides the shared memory space between the threads into 3 partitions, in an attempt to reduce the captured memory state. The first, is the non-speculative execution memory state that is always correct. The second, contains the state of the speculative execution. And the third, is a coordination partition that is used for the synchronization between the two thread groups. Initially, all the variables (heap or stack) reside in the non-speculative partition. Once a speculative thread attempts to modify them, they are copied to the second partition.

RecReplay [RON+99] is a system aimed at facilitating cyclic debugging in parallel programs by detecting data races. Similarly with the approaches presented in this thesis, it uses replaying to execute heavy-weight operations without affecting the original process. Storing all the non-deterministic choices made during a parallel execution

would allow an accurate replaying, however, it would cause significant execution overheads. Since those overheads are harmless on a separate process, the race detection algorithm runs during replay.

The approaches developed during this thesis for implementing a CR system have similarities with the above approaches. However, spawning several additional execution contexts and freezing executions to capture the inputs on-the-spot incurs additional online overheads that could be avoided.

3.3.7 Other Unix-based user-space approaches

Jockey [SAI05] has implemented a user-space CR library that is linked to a target Linux process without requiring kernel modifications or patching. It takes periodic checkpoints for diagnosing long-running programs. Any non-deterministic system calls and CPU instructions are re-written and recorded, so they can be replayed afterwards. As its intended usage is during testing and debugging, it does not prioritize efficiency. As a result, it introduces noticeable online overheads.

Libckpt [PLA+94] is a Unix CR library that also aims at taking snapshots of long running programs. In the case of a failure, the program could simply resume from a stored checkpoint instead of rerunning afresh. It is based on fork checkpointing, a technique that other CR systems are also using. Additionally, it supports incremental checkpointing, which omits data in late snapshots if those can be reused from previous ones. Finally, it allows the users to pass extra information for tuning the capture operations, like what data can be omitted from a capture or at which point in time it is preferred to take a snapshot.

CRIU [VIR20] is a CR system based on Linux that does not depend on particular source languages or runtime environments. Its main drawback is that it captures all of the application state, dramatically increasing the storage requirements. While this is necessary for CRIU's intended usage, which is process migration in data centers, it is extremely wasteful for mobile devices.

CERE [CAS+15] is another Linux-based approach that identifies *codelets*, distinct computationally important areas of code. It uses a page-based mechanism to capture each codelet's working set, so that it can replay it in isolation. It differs from the presented approaches in that it aims to accelerate benchmarking large scientific applications by breaking them down into a set of very short regions. As such, increased latency during capture is not a significant problem for CERE. Runtime overheads are

typically over 20% and as high as 250%. Part of this is due to CERE not using Copy-on-Write. When a page is first accessed and a page fault is triggered, execution freezes until CERE copies the page to a temporary buffer, regardless of whether the page will be modified or not. The approach presented in this thesis leverages Copy-on-Write to efficiently offload the page duplication to the kernel, which will copy only the modified pages.

3.4 Summary

This chapter presented a relevant literature review around the topics of iterative compilation, compiler optimization, as well capture and replay systems. The following three chapters present novel approaches that fill in the gaps of existing approaches, for creating an optimization approach that is fit for mobile applications.

Chapter 4

Using Capture and Replay to optimize C programs

4.1 Introduction

Mobile devices are an indispensable part of our daily routines, yet they predominantly run poorly optimized code. On one hand, their limited computing and energy resources reinforce the necessity of an optimization framework. On the other hand, it is those very restrictions that hinder the adoption of code optimization techniques that are well-established on server and desktop environments.

In this chapter it is investigated whether iterative compilation can be applied on the highly restricted environment of mobile devices.

The idea of a representative hardware, operating environment, and software stack on a mobile system to compare different optimizations on application code is unrealistic, while creating offline inputs is non-trivial [SEE+14]. Therefore, any offline approaches are impracticable. While online approaches inherently operate on real inputs they introduce unbearable side-effects. Compilation strategies that lead to runtime crashes or yield wrong computation outputs are not something uncommon when applying aggressive code optimizations [CUM+18]. In a real setting those will never be tolerated by mobile device users. Mixing multiple optimizations can also be counter-productive, introducing prohibitively expensive execution and energy overheads. Even self-adapting algorithms, which gradually evolve and reduce such overheads, will inevitably evaluate significantly slower code during their early learning curves.

This chapter describes an approach that enables iterative compilation for C programs on mobile devices by overcoming most of the obstacles of existing approaches.

With online input captures, the proposed approach is able to perform offline mass-evaluations of different code transformations. For each evaluation a targeted function is replayed using the same input but different compilation strategies. Evaluations happen at idle times, therefore users do not have to suffer the side-effects of iterative compilation, which include dramatically slower or even crashing executions.

Initially, the time consuming and computationally intensive functions of C programs are identified and extracted from the rest of the sources. A single function is targeted per program and in this chapter it will be either referred to as the hot function or the hot region. The input to the hot region is then captured by storing only a minimal set of pages that are actively used while the function runs. Captures happen transparently to the program without requiring any kernel modifications. They also remain unnoticed by the users as they incur on average a slowdown of less than 2ms. When compared to traditional capture approaches, which take a snapshot of the whole memory, the proposed mechanism achieves space savings between 2 and 3 orders of magnitude.

Once a capture is taken, the replay mechanism can repeatedly restore the same input and execute the hot function, which may use differently compiled code. For compilation, the code transformation space of a compiler is randomly probed. This is a practicable evaluation mechanism for the mobile environment as executions, including crashed or significantly slower ones, are performed while a device is idle and charged. Therefore, users will not have to endure any side-effects of iterative compilation in order to benefit from its findings.

The effectiveness of the proposed approach is evaluated with an iterative compilation system for C programs. By targeting and replaying only the computationally intensive functions, the evaluations of multiple code transformations are accelerated. The presented system is able to outperform the highest optimization level of Clang [CLA20] by up to 57%.

This chapter is organized as follows. Section 4.2 presents a capture mechanism for C programs on a mobile architecture. Section 4.3 outlines a replay mechanism for C functions, able to work in spite of code transformations. Section 4.4 describes the experimental setup, along with a system implementation for C programs. Section 4.5 evaluates the proposed approach on C programs executed on a mobile device. Finally, the concluding remarks of this chapter can be found in Section 4.6.

4.2 Capturing C programs

The main idea behind this thesis is to capture the behavior of a program online, during a regular invocation, and then accurately replicate it offline, under different compiler optimization strategies. Any methods that perform I/O or have a non-deterministic behavior are excluded by the proposed approach. Therefore, capturing a program's behavior is the same as capturing its main memory state right before a targeted hot function is executed. Simple approaches could save all processor registers and everything that resides in the main memory space of a program [VIR20]. This can incur noticeable overheads for mobile users. Firstly, it affects performance as the low-latency execution flow on a mobile system freezes until everything on the memory space is dumped to a disk. And secondly, as Chapter 6 will show, this requires an increased amount of storage as it writes out the entire memory space instead of only the memory pages that contain the input of a targeted hot function.

The approach introduced by this section re-purposes two kernel mechanisms to capture only a subset of the memory space with the least amount of overhead. With a `fork` right before the invocation of a targeted hot region, a copy of the VMA is created. The kernel's efficient Copy-On-Write mechanism will duplicate any pages that are modified by the hot region, leaving a copy of the memory space in its original state. The program's memory pages are also read protected before invoking the hot region, which will offload the identification of accessed memory pages to the kernel. When the execution of the hot region has finished, the proposed mechanism simply has to store the original state of the pages that were accessed. This approach is transparent to the user and to the program, without requiring any kernel modifications. As shown in Figure 4.1, it consists of 6 stages and it is described in detail in the remainder of this section.

❶ Initiating a capture:

Code is added to the entry point of a hot region that checks whether a capture should occur. In that case some preparatory steps are performed, necessary for the identifying and preserving the original input to the hot region. Since only a single capture is required to perform an offline iterative compilation search, no additional invocations of the function are captured during the same run of the program. Chapter 6 describes how this step is revised for interactive Android applications to allow a greater control over the capture frequency.

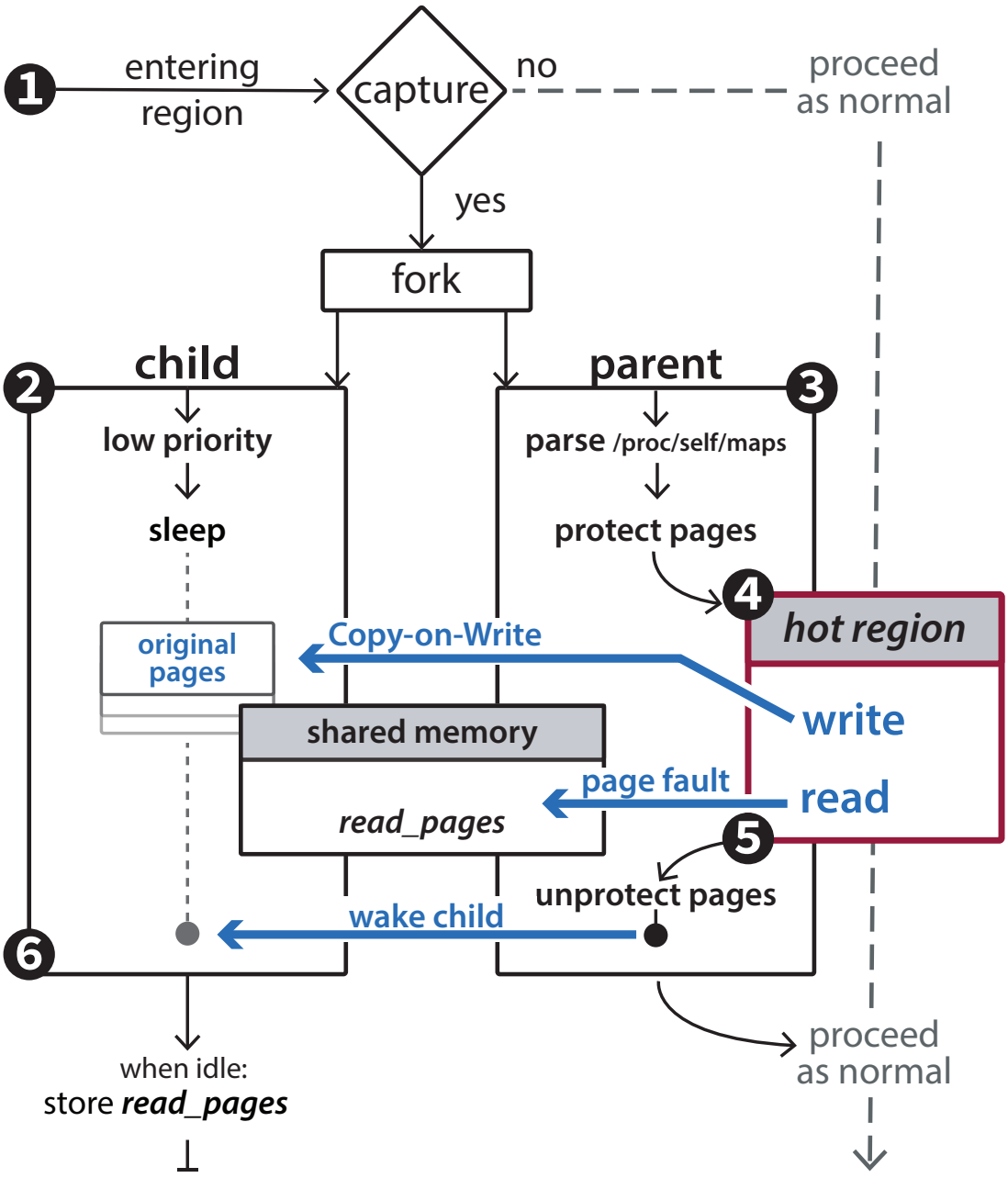

Figure 4.1: A lightweight capture mechanism stores a partial state of a C process that enables re-executing a hot region of code at any future point in time. It is transparent from the users and the program developers. By leveraging the fork and Copy-On-Write mechanisms, a copy of the original state is preserved in a child process. And with page read protection and fault-handling the set of read pages can be identified. This approach minimizes the performance overheads, the storage requirements, as well the time required to restore the input on each replayed execution.

❷ Forking a child process:

Freezing the execution flow before invoking the hot region until all memory is dumped to a disk would have wasted precious performance cycles or the limited storage resources of a mobile device. Instead, the proposed approach uses `fork` to spawn a child process, which transparently keeps the original input in the child with minimal overheads. The virtual memory table is then duplicated for the child, with both child and parent processes initially pointing to the same set of physical pages. It is only when the parent attempts to modify a page that a physical page is copied. The copy is assigned to the parent with the write permissions restored, leaving the child process with the original content. This process is carried out efficiently by the kernel as a part of the Copy-On-Write mechanism. This is significantly faster when compared to full capture approaches, or approaches that manually copy each memory page right before it is used by the region [CAS+15]. At this step, the child's priority is set to the minimum possible value and the process is sent to sleep.

❸ Protecting memory pages:

Saving all of the memory pages to disk is quite wasteful and even prohibitive in practice as several captures of different programs could be simultaneously kept on a mobile device. Therefore, by identifying and capturing only the set of pages that are used while the hot region executes, the snapshot sizes can be significantly reduced. The `/proc/self/maps` file, which is part of a pseudo-filesystem on Linux machines [LIN20], is parsed to get the full list of memory pages that lie in the parent process's address space. Then, most of these pages are read-protected, which will cause deliberate page faults once the process attempts to read from them. Those will trigger a fault handler that is also installed at this point and will perform two actions. Firstly, it will store the address of the offending memory page in a shared buffer between the parent and child processes. Secondly, it will restore the relevant access permissions to that page so that any future read attempts will not trigger another page fault.

❹ Execution of the hot region:

At this point, the parent process can proceed with the execution of the hot region as it normally would have done. Other than fault handling when the memory pages are first read, and Copy-On-Write when the memory pages are first written, there is no further

overhead. Given that most programs have a decent amount of spatial locality [DEN06], the storage requirements for the accessed pages should be in the same order of magnitude of the memory space that is actually required by the hot function of the program. This causes the amount of state that is saved to remain low. Similarly, the amount of page faults and the associated fault handling overheads should also remain low.

❺ Ending the hot region:
Once the parent process has finished with the execution of the hot region, it wakes up the child process. It also restores the access permissions to any memory pages that are still read-protected and then uninstalls the fault handling mechanism. Finally, the parent process continues execution, as it normally would have done, to code that comes after the hot region. If a hot region is invoked multiple times, only the first input will be stored. In other words, there will not be another capture on any subsequent invocations of the hot region. Improvements in Chapter 6, make the capture point configurable, allowing more captures in the same run.

❻ Saving the memory state:
Once the child process is awoken it starts spooling memory pages that were marked as read by the parent to the disk. Since it has the lowest possible priority it will store pages to disk only when the system has unused processing and I/O capacity. As a result, there will be no inconvenience for a mobile device user.

Apart from the used memory pages there are some additional data that need to be captured. This mainly concerns the architectural state of the processor. In particular, the non-volatile registers and some special registers must be preserved. This is done through inline assembly, while ensuring that the assembly itself is not interfering with the register values to be retrieved. This value retrieval happens as the very initial step of the capture process, with the exception of the Program Counter (PC) register. PC is instead retrieved later on by the child process, once it is awoken by the parent. It points to code that lies right before the hot region. That point will serve as the entrypoint of resuming execution for any future replayed processes.

4.3 Replaying a C function

For the mass-evaluation of different code transformations of a hot region there needs to be a mechanism that can create a process with an identical input state with the captured one. This mechanism, apart from fixing the input, must also work well even when the underlying code of a hot function is modified by iterative compilation. To replay a captured hot region, the program's binary and the saved state must be loaded into memory. Then, the captured architectural state of the processor is restored, with the PC register being the last one. Restoring the PC is effectively a *jump* into the point in code right before the invocation of the hot region. The function is then re-executed until completion. The performance statistics can be collected once the replay process terminates the hot region execution.

The replay mechanism internals are relatively straightforward as it supports only a single function of a C program. Replaying several methods of interactive applications, based on the Android runtime, is a more involved process and is described in detail in Chapter 6. The remainder of this section describes the specialization that is required for replaying a hot function that can be differently compiled through iterative compilation.

4.3.1 Replaying differently compiled C functions

To compare between different optimization strategies the replay mechanism must successfully execute regardless of any code transformations to the underlying binary. Producing a valid executable under any code transformation is a delicate process. Most of the code is untouched, however, changes to the hot function can easily break replaying. To consistently replay different code, while fixing the input, requires some low-level tinkering at the assembly and link stages during the generation of the binary.

The system, described in Section 4.4.1, repeatedly builds binaries that contain differently transformed code for the hot function. Those binaries, however, easily become inconsistent with the originally captured input data, crashing the replay process. A first issue concerns code transformations that result in modifications to function pointers and addresses of globally accessible variables. A second issue arises when code transformations either increase or decrease the size of the hot function. With some link-time tinkering and code or binary segment alignment it is ensured that the binaries produced, despite having different underlying code, will be consistent with the captured state at all times. The remainder of this section describes how this is achieved.

Handling function or data pointer changes. For each program, a single hot function can be modified from code transformations found by the system described in Section 4.4.1. This function is compiled into position independent code, a commonly used option that makes a program's start-up faster as the dynamic linker has to apply a minimal set of relocations. Those are applied to the binary sections named Global Offset Table (GOT) and Procedure Linkage Table (PLT). GOT has entries with offsets to global or static variables. Each entry is initialized by the dynamic linker. PLT is used for invoking non-static functions and has entries only for the ones that are called by the program. These entries are initialized with PC-relative offsets to the function's code in the text segment of the binary. Lazy initialization can be performed.

Code transformations can potentially replace particular generic method calls to faster architectural-specific ones. When this happens, PLT segment entries change, which causes cascading changes to the GOT segment and any other binary segments that follow. As a result, the global or static variables and pointers to non-static functions can have different relative addresses to the ones they had during the original capture. This will crash the replay process, as the restored data will be placed at different addresses in memory than the ones the new binary expects.

To address this issue an auxiliary object file is built and linked against the hot function. The object contains a function that is never actually called but it is marked as being called. In its body it contains calls, with dummy arguments, to any function that might be introduced by some lowering or architectural-specific code optimizations. This causes the assembler to include additional entries for these calls in the PLT. As a result, this section of the binary becomes immutable for any given program. This workaround introduces no additional overheads during the program's bootstrap, as the additional PLT entries will never be lazily initialized.

Handling code size changes of the hot function. Several code transformations might affect the size of the hot function, either by decreasing or increasing it. This will cause inconsistencies with the restored state, as the relative offsets to the pointers of any functions that follow will be shifted. Additionally, and in similarity with the previous issue, altering the size of the hot function will cause cascading changes to any segments of the binary that follow the `.text` segment, which is the one that contains the program's code.

With custom address aligning it is ensured that the variable hot function size will not affect the offsets to any function pointers. This is achieved with padding before

and after some relevant binary segments, as well right after the hot function, which is within the `.text` segment. For the former, a linker script aligns to a two-page boundary some segments, like the `.text`, `.data`, `.rodata`, and `.bss`. For the latter, a dummy method was introduced after the hot function to align the code that comes after it, also to a two-page boundary. The added method contains a single `nop` instruction and requests from the assembler and linker (using the pre-processor) to be aligned, not to be inlined, and not to be optimized away. This adds enough empty space after the hot function, and when combined with the link-time padding it allows code expansion without causing modifications to any function pointers or binary segments that follow. Additional size checks in the system ensure that this boundary is not violated with a relevant warning during the optimization search. If this ever happens the system will not proceed with replaying. The alignment boundary was empirically chosen and no violations were observed during evaluation. It was set to a 2 page boundary.

4.4 Experimental Setup

The proposed approach was evaluated with a series of experiments on a *Motorola Nexus 6* mobile device that ran Android version 5.1. The device has a *Qualcomm Snapdragon 805* processor that is powered by four 2.7 GHz *Krait 405* cores. The capture and replay mechanism was specific to C programs on the ARM architecture and was targeting a single hot function. To minimize the performance noise the following execution environment was imposed. The mobile device was idle. All four cores were kept online and their execution frequency was hardwired to the maximum available value. This allowed more precise timings while minimizing the overall time of any evaluation. Additionally, the maximum setting ensures that no hardware optimizations are disabled due to lower frequency or voltage scaling [BAO+16]. Finally, the device temperature was checked before performing any evaluations. If it was above a threshold then the device entered a cool-down period. This minimized the chance of any interference from thermal throttling. We noticed that this could also be achieved when the device is placed on a cooling pad, in an air-conditioned room.

The open source BEEBS benchmark suite [BRI20] was used to evaluate the proposed approach. The suite contains programs written in the C language, focused on embedded systems. Table 4.1 shows the used benchmarks, which contained only deterministic calls and were able to successfully link and operate with the bionic library [GOO21a], which is a more lightweight implementation of the standard C library

Name	Description
ADPCM	Adaptive Differential Pulse-Code Modulation decoder
Blowfish	Symmetric-key block cipher algorithm
Bubblesort	Simple sorting algorithm
Dijkstra	Finds the shortest paths between nodes in a graph
FFT	Fast Fourier Transform
FIR	Filter int data array based on passed int coefficients
Huffbench	A data compression benchmark

Table 4.1: C programs from the BEEBS benchmark suite [BRI20] that were linked against the Bionic library and were used for the experimental evaluation. For each benchmark a single hot function was identified using Callgrind [VAL20]. Subsequently, it was extracted from the remainder of the source code. Those functions were then targeted for optimization by a prototype iterative compilation system.

on Android. A hot function of each benchmark was extracted from the remaining of its sources, as visualized in 4.2, to allow an integration with the presented iterative compilation system that is presented in the next section.

With an initial profiling stage using the Callgrind tool [VAL20], the *call graph* as well the call frequencies of each program's function were generated. Using this information, the function with the highest execution runtime was identified as the hot function. Since a single hot function was supported, any callee functions whose source code was available (i.e, not a library call) were inlined. Finally, the resulting hot function was extracted from the remaining of the sources in a specific format that the presented system was expecting.

4.4.1 Iterative compilation system for C programs

For the evaluation of the proposed approach a system for C programs was developed. As shown in Figure 4.2, a random transformation sequence is extracted from the compiler's space and is used to compile only the hot function of the input program. The hot function is compiled into its own binary object. The rest of the sources are compiled using a standard optimization flag of the compiler (i.e, `-O3`) and stored in a separate object file. The two object files are linked together using the assembly and link time strategies as described in Section 4.3.1.

All produced binaries are evaluated using the replay mechanism. Compilations

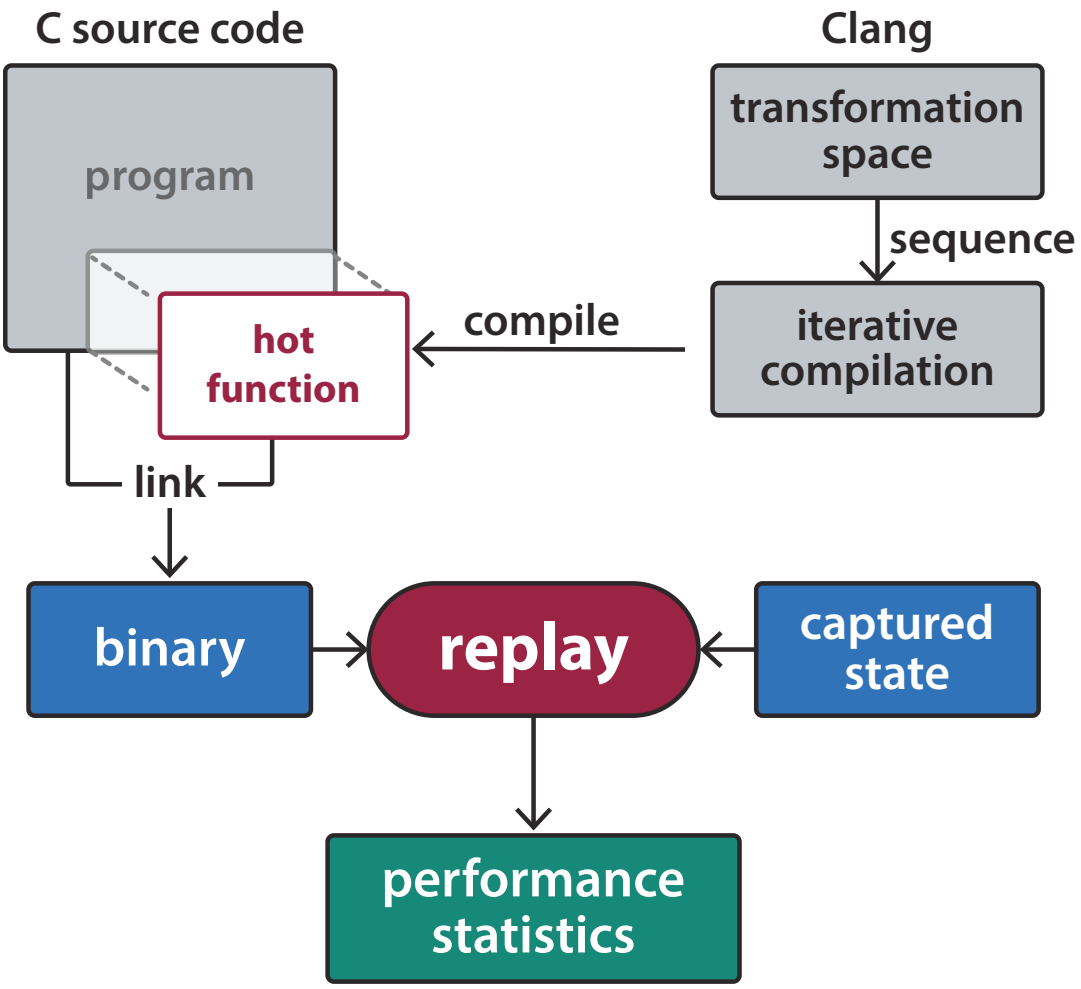

Figure 4.2: An iterative compilation system for C programs. To evaluate multiple code transformations, a single hot function is repeatedly compiled and subsequently linked to a new binary. The transformations are randomly extracted from the compiler's space. Each generated binary will operate over the same input, as the execution of the hot function will be replayed by restoring previously captured data. Once the hot function execution finishes the performance statistics are collected as a means of evaluating different code transformations.

and executions are performed on a mobile device, which ensures that the produced binary will be optimized for the underlying architecture. The evaluations happen when a mobile device is idle which minimizes the performance noise. Finally, by focusing only on a hot region of a program, instead of the entirety of its code, the whole iterative compilation process is accelerated.

The iterative compiler uses Clang [CLA20] version 3.6.1. It randomly constructs code transformations from a list of 60 flags. If the flag accepts any parameters then those are also randomly selected. In the experiments presented in the next section, 2000 points of the transformation space were randomly probed. Each binary version was replayed 10 times and outlier removal was performed using the *interquartile range* method with a k set to 1.5. The computed final outcome of the tested code transformations was manually verified for correctness. The performance results between different transformation sets were compared using a *two-side student's t-test*. Where applicable the 95% confidence intervals were calculated and presented.

4.5 Experimental Results

Three sets of experiments were used to evaluate the proposed approach. The first set showcases the performance gains that the iterative compilation system can achieve. The second set examines the one-time execution overheads that are introduced by the capture mechanism. The third compares the space overheads of the capture mechanism against traditional approaches.

4.5.1 Optimizing C programs with iterative compilation

The goal of the proposed system was not to improve iterative compilation per se, but to demonstrate that it now becomes applicable on the restricted mobile device environment. Nevertheless, it is important to show that this approach is able to optimize C programs.

Figure 4.3 shows the speedups of the best binaries that were found by the developed system for the hot functions of the used benchmarks. All findings were compared against the `-O3` optimization level of Clang, which was used as a baseline. Despite visiting only a small fraction of the enormous compilation space of Clang, the proposed approach was able to increase the performance of each benchmark. The highest achieved speedup was 57%, while the lowest was 16%.

4.5.2 Capture overheads of C programs

To push all overheads associated with iterative compilation offline, through replay-based evaluations, an online captured execution of the C program must precede. There-

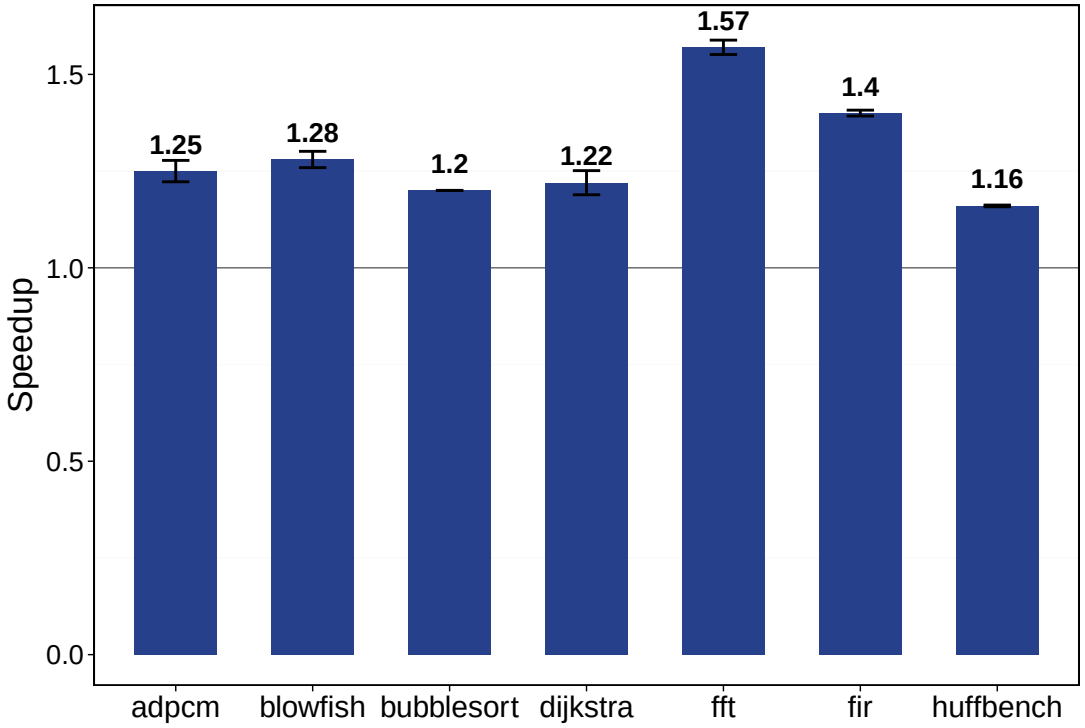

Figure 4.3: Speedup obtained with by the proposed iterative compilation system. The baseline was the `-O3` optimization level of Clang. The findings were discovered by extracting code transformations sequences from the compiler's optimization space, using 2000 random probes. The error bars represent 95% confidence intervals.

fore, it is important to establish that capturing the input of a hot function introduces as little overhead as possible and remains in all cases transparent to the users.

Figure 4.4 shows the execution runtime of the C benchmarks that were executed normally versus when they had their inputs captured. The overheads introduced by the proposed mechanism were negligible when compared to the total execution runtime. The average slowdown was less than 2ms. This amounts to less than 1% of the total runtime, while the highest was just 1.7%. Most of the overheads come from parsing the `/proc/self/maps` file to get the virtual memory areas, and from forking a child process before executing the hot function. The overheads related to fault-handling were small, as the total number of page faults were kept low even for memory-intensive programs. In the worse-case scenario, the number of page faults can be as high as the number of pages owned by the program. Similarly, the Copy-On-Write mechanism also had low overheads. Utilizing this mechanism is quite efficient as it is engaged only when the parent process modifies a page. And even then, the copy operation happens in the kernel space, transparently to the program making the proposed mechanism fast.

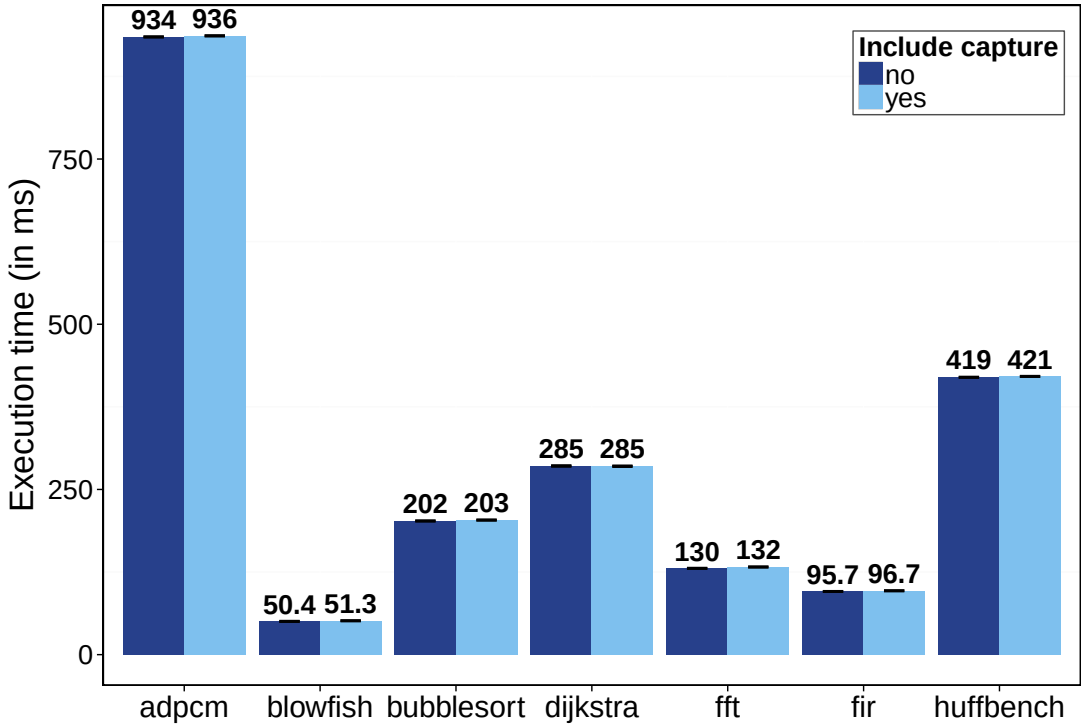

Figure 4.4: Snapshot overheads that occurred during the one-time capture operation for the C benchmarks. The darker color shows the execution time of a hot function that was normally executed and the lighter color shows the execution time when the input to the function was captured. In any case the introduced slowdown never surpasses 2% of the total execution runtime, therefore it is unlikely to noticeably affect a mobile device user in a real life scenario.

Given the negligible overheads introduced by the capture mechanism, it is unlikely to noticeably affect real mobile device users. This makes the proposed optimization approach applicable for the mobile environment.

4.5.3 Space savings against full capture approaches

Most of the overheads introduced by the capture mechanism, as explained in the previous section, were related to operations performed right before the invocation of a hot function. Those operations were necessary for identifying a minimal set of pages that are actively used while the hot function runs. For that almost imperceptible overhead, the proposed capture approach is able to significantly decrease the amount of storage needed for each input snapshot.

Figure 4.5 visualizes the space savings of the capture mechanism. It stores a min-

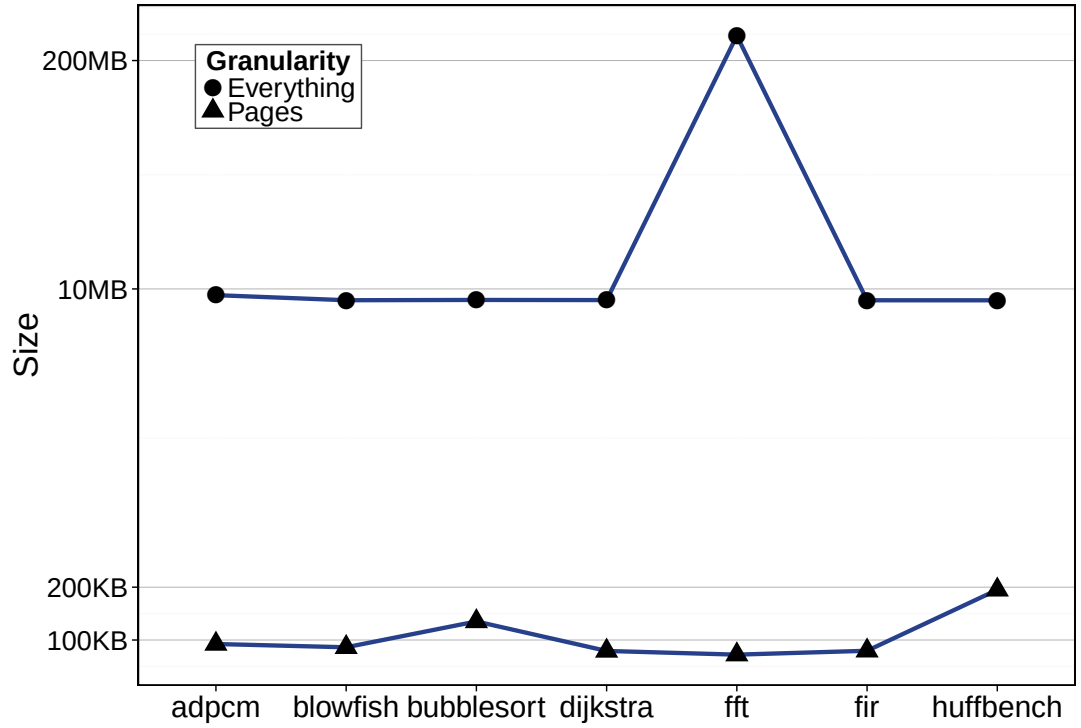

Figure 4.5: Comparing snapshot sizes between full capture approaches against page-granularity captures for the C benchmarks. The captured information concerns a particular hot function of a C program. Less than 200KB of data were captured for each benchmark. Full snapshots require at least two orders of magnitude of additional storage for any of the used benchmarks.

imal set of memory pages that contain all the input the hot function needs. It is compared against a full capture approach that stores the entire state of a C process. The highest amount of storage needed for replaying a hot function was less than 200KB of data. Full snapshots contained at least two orders of magnitude of additional information. *FFT* was an extreme case where more than 200MB were captured in the full snapshot. The proposed mechanism had reached space savings of three orders of magnitude for that particular case.

Minimizing the amount of captured information is important as mobile devices come with limited internal storage. Easily fitting data from multiple captures to a mobile device without significantly reducing the available storage of the user means that potentially sensitive data will never need to be transferred to external or cloud storage devices.

4.6 Summary

This chapter investigates whether iterative compilation can be applied to the highly restricted mobile device environment. It proposes a hybrid approach that combines the benefits of offline and online approaches to solve several issues that have previously rendered iterative compilation infeasible for mobile devices. With a single online capture it is able to perform at a later stage mass-evaluations of code transformations. Evaluations are done through replaying, which keeps the same input on each replayed execution. Every evaluation is performed when the device is idle and charged, which hides any overheads associated with iterative compilation, making the proposed approach transparent to the users and practicable for the mobile environment.

The capture mechanism minimizes the execution and storage overheads by limiting the snapshot data to the set of memory pages that contain the input to the hot function. It does not require any kernel modifications and it is transparent to the users, incurring less than 2ms slowdown per capture. It conserves two to three orders of magnitude of storage when compared to full capture approaches [VIR20]. With a single capture, the replay mechanism can then re-execute a C function, even if its underlying code was compiled with different code transformations.

A system was implemented to evaluate this approach. The system is able to outperform the `-O3` optimization level of Clang by 29% on average. This was achieved with a random-based search over a small fraction of the compiler's transformation space.

While it is established that iterative compilation can indeed become a practicable optimization approach for mobile devices, many hurdles remain when considering a real life scenario. Android device users interact with real Android applications instead of C programs. As such, the capture mechanism requires adaptations to support such a complex runtime environment. As described in Chapter 6, it should be able to execute different types of code, including interpretation, JNI code, and dynamically linked code. At the same time it should be able to reliably operate in the presence of memory-related security mechanisms. A scalable approach should automatically identify, recompile, and link a hot region, which may contain complex method call nesting. Its code should be statically analyzed a priori to decide whether it contains any sources of non-determinism. Validating the correctness of any applied code transformations must also be automated, without requiring any effort from the application developers. This is a non-trivial problem. Finally, the optimization space of the Android compiler should be expanded, as it currently comes with just a handful of code transformations

that yield no benefits when combined with iterative compilation.

Chapter 5 describes a considerable implementation effort to develop an LLVM backend for the Android compiler, which tremendously increases the code optimization space for Android applications. Chapter 6 presents a system that utilizes this LLVM backend. It is able to automatically identify replayable and computationally intensive code regions of interactive Android applications and capture real user inputs to them. Two additional capture mechanisms are also introduced. Then, using a novel replay mechanism, the system can restore the input to create a partial Android application process, which has all state required for evaluating several code transformations. It also automatically performs code correctness verification and it can employ crowdsourcing to significantly accelerate the offline optimization search. This approach can potentially scale to any Android application, without requiring any user or application developer effort.

Chapter 5

Enabling aggressive code optimizations for Android applications

5.1 Introduction

With more than 5.2 billion of active users [GSM20], smartphones are definitely the computing medium of our era. Yet, there is no framework able to aggressively optimize interactive mobile applications at scale. Even in the preeminent mobile platform, Android, the compiler backend relies on just 18 distinct optimization passes [GOO20b]. When compared to well-established compilers, this transformation space is an order of magnitude smaller [LAT+04]. As a result, immense amounts of optimization potential is wasted, under-utilizing the already limited computing and energy resources on the mobile environment.

This chapter describes the implementation of a full compilation toolchain, based on LLVM, able to AOT compile interactive Android applications. Prior to this backend, the only complete, alternative toolchain for Dalvik code was the default Android backend, called *optimizing*. Despite its naming, the default backend is designed to be conservative rather than highly optimizing. It only applies a handful of optimizations that are guaranteed to either have a positive impact or no impact at all on any encountered code. Exhaustively searching such a small space of conservative optimizations yields no better binaries. A more sophisticated and mature optimization infrastructure could unlock a tremendous optimization potential for mobile applications.

The system presented in this chapter is the first that enables passing code from the Android compiler to the LLVM compiler. It focuses optimization to the time consuming and computationally intensive code regions that may contain complex method

nesting. The code regions are automatically identified with a sample-based profiler. The region's methods are then passed to the LLVM backend to AOT compile them. Initially the LLVM backend invokes the default Android compiler facilities to generate its IR, called the *HGraph*. It then applies to the *HGraph* the default optimization passes as Android normally would have done. Despite being conservative, these passes are specific to ART and therefore can be beneficial. Then, it performs an IR-to-IR translation to output LLVM bitcode, which exposes Android applications to a significantly larger code transformation space. In addition to that, a few Android-specific optimization passes were developed.

The LLVM backend represents a significant engineering effort with more than 25,000 lines of code. It is released as open source software [MPE21] to allow researchers or compiler enthusiasts to use it and further improve it. Despite being a work in progress, the backend is able to outperform the Android compiler for most tested applications. By applying the aggressive `-O3` optimization level, the LLVM backend outperforms Android by 7% on average. Also, some Android-specific optimizations have improved over generic ones by squeezing as much as 38% of additional speedup. As the IR-to-IR translation matures, the performance gains are expected to grow even further. Nevertheless, even a highly optimizing compiler is tuned to be conservative by default. A more aggressive optimization framework, like the one presented in Chapter 6, could exploit the capabilities of this backend to a much greater extent.

This chapter is organized as follows. Section 5.2 describes how a hot region of an application is automatically detected. Section 5.3 presents the Android LLVM backend that performs an IR-to-IR translation, implements Android-specific optimization passes, and finally assembles and links the generated code to an Android application. Section 5.4 describes the experimental setup, followed by Section 5.5 where the implemented backend is evaluated against the Android compiler. Section 5.6 provides the concluding remarks of this chapter.

5.2 Detecting computationally intensive code regions

Interactive applications consist of multiple asynchronous tasks whose outputs, once calculated, are presented to the users. Due to the low-latency nature of mobile devices, these tasks are executed in background threads and occasionally synchronize to a User Interface (UI) thread to provide a visual update. The computational intensity and the execution frequency of individual tasks depend on the user's input and the underlying

code. Typically, a fraction of the application's code dominates execution runtime, which will be referred to from now on as the *hot region*. In contrast to Chapter 4, a hot region may consist of multiple method calls that may or may not be inlined. By focusing exclusively on hot code regions, the overall execution times can be improved while the compilation times remain within reasonable levels.

The remainder of this section details how a hot region of an Android application is automatically detected. Subsection 5.2.1 describes a static bytecode analysis pass that identifies compilable methods. Subsection 5.2.2 describes a one-time profiling process for detecting optimization worthy methods. Finally, Subsection 5.2.3 describes how the profiling and static analysis data are combined for identifying the hottest code regions.

5.2.1 Static bytecode analysis

Initially, with static bytecode analysis the *compilable* methods are detected. These are methods that can be successfully compiled and optimized by the LLVM backend. Any methods that are classified as pathological cases by the Android compiler [AND20] are excluded from this list. These are methods that the Android compiler itself cannot process. Therefore, the LLVM backend will not be able to process them either as it relies on Android compiler's IR as its input to its translation pass. This is not a major limitation. As it is shown in Section 5.5, pathological cases usually take a very small percentage of the execution runtime. Most of the native methods are also excluded since they are already in a binary format. There are a few exceptions that are converted into intrinsics, as detailed in Section 5.3.2. Finally, any methods that are not supported due to current limitations of the LLVM backend are similarly excluded at this stage.

5.2.2 Sample-based profiling

The analysis phase is followed by a lightweight online profiling phase. An application is invoked using Android's *sample-based profiler* in order to generate a *trace* file of the execution. By sampling the *active stack frames* at fixed intervals, the profiler can estimate the execution time spent on each method. This happens once per application. A sample-taking thread is executed every 15 seconds and each time it takes samples for 10 seconds. The thread may use a start-up delay, configurable per input application. A new sample is recorded at a millisecond granularity since only an approximation of the runtime is actually sufficient. This whole configuration keeps the profiling overheads

Algorithm 1: Detecting computationally intensive code, named hot regions, of interactive applications by utilizing static analysis data and runtime execution traces.

```
input: methods from a sample-based profile
output: method with biggest compilable region

def estimateRegionRuntime(method):
    sum ← 0
    compilable ← compilableRegion(method)
    foreach c ∈ compilable do
        sum += runtimeExclusive(c)
    end
    return sum
def compilableRegion(method):
    def inner(m, l):
        if m ∉ l and IsCompilable(m) then
            add(l, m)
            foreach c ∈ callees(m) do
                inner(c, l)
            end
        end
    list ← ∅
    inner(method, list)
    return list
def IsCompilable(method):
    if IsPathologicalAndroid(method) then
        return false
    else
        return IsSupportedByLLVM(method)
    end

sort(methods, estimateRegionRuntime)
return methods.first
```

to a minimum, making this one-time online phase unnoticeable to the device users. The outcome is a list of the executed methods, their execution time estimates, as well their execution frequencies.

5.2.3 Combining data to extract hot code regions

Once the profiling data are generated, the computationally intensive hot regions can be detected. This process is outlined by Algorithm 1. It first computes the compilable code regions for each method in the *trace* file. It excludes from these regions any methods that were identified as non-compilable by consulting the static analysis data. Then, the total execution runtime of each region is estimated. This is the sum of the execution runtime of each *reachable* and *compilable* method, including the outermost caller. Each execution runtime is *exclusive*, meaning it concerns only the operations that belong directly to the method itself. In other words, any computing cycles spent during external method calls are excluded.

Once the execution times of compilable regions of each method are estimated, the most significant one becomes the hot region. This region will be targeted for aggressive optimization. Despite optimizing only a fraction of an application's code, this targeted approach decreases the compilation times and binary sizes, while still allowing noticeable performance improvements.

5.3 LLVM backend for Android

This section describes the LLVM backend for Android applications. Prior to this implementation, the only complete compilation toolchain for Dalvik code was the Android compiler's default backend, which is used throughout the experiments as the *baseline* for comparisons. The default backend applies only 18 distinct code transformations [GOO20b] that are considered safe, meaning they might have positive or no impact at all on any encountered code. Searching exhaustively through this small transformation space yields no better binaries than the baseline, as it lacks aggressive code transformations found in well established compilers like LLVM [LAT+04]. The only alternative approach to fine-tune specific code regions is to rewrite them using JNI, requiring additional effort from the application developers. Even then, there can be high overheads proportional to the amount of interactions between the ART and JNI environments [KUR+01]. With the LLVM compiler backend, the optimization

strategies for Android applications are significantly increased.

5.3.1 IR-to-IR translation for the generation of LLVM bitcode

LLVM is a mature compiler infrastructure that supports many language front-ends and machine back-ends. It was lacking however, an Android front-end along with a complete compilation toolchain that supports ART. Such capability could bring tremendous optimization potential to the massive Android ecosystem [GSM20]. Two main approaches could be followed to implement an LLVM backend for Android. The first one, is to create a front-end at the Dalvik bytecode level and provide for it full runtime support. The second is to perform an IR-to-IR translation from Android compiler's IR to LLVM bitcode. This was the pursued approach and is explained in the remainder of this section.

5.3.1.1 Overview of the IR translation

The default compiler backend performs a handful of optimization passes [GOO20b] that are tailored for ART. As they can improve the effectiveness of any subsequent code transformations, it is wise to apply those first before applying more sophisticated optimizations. The proposed approach leverages the Android compiler to achieve that. Initially, it invokes the default backend to generate its IR, called the `HGraph`. Then, it applies the passes as Android normally would have done, and finally performs an IR-to-IR translation to generate the LLVM bitcode. This approach comes with some additional benefits. One, is that the runtime support for the backend is simplified. A subset of the required runtime entrypoints for the LLVM compiled code could utilize the existing ones for the Android compiled code. For the remaining entrypoints, a similar convention with the Android-compiled code can be followed. That is, to create assembly *stubs* that are exposed to LLVM as function pointers with statically known offsets, which start from the special *Thread register*. Another benefit is that any changes that might be introduced from the high-level source code languages (i.e., Java or Kotlin) all the way down to the *HGraph* generation would be available to the backend without additional effort. Such changes will be implemented at the lexical, syntax, or semantic analysis phases of the standard Android compilation process. A minor disadvantage to this, is that any code that is uncompilable by Android will also be uncompilable by LLVM. Nevertheless, as shown in Section 5.5, these cases represent a very small percentage of the total execution runtime. Optimizing them would

Type	Description
`uint32_t*`	Method arguments packed in an integer array.
`JValue*`	Store return result in a union.
`Thread*`	Class for managing the current Android Thread.
`ArtMethod*`	Class for managing the current Android Method.
`uint32_t`	Offset to the boot image caches.

Table 5.1: Method arguments of the entrypoint that invokes LLVM code from ART. It follows a similar calling convention with the default Android code. The first two contain the callee arguments and the return result. The following two are heavily used by LLVM code to access ART structures (strings, classes, or methods), other caches, or synchronized objects. The last one is used to access caches that contain immutable runtime objects.

not have affected the overall performance in a major way anyway. In any case, a front-end at the bytecode level would require an even greater effort for functionality that was already well implemented and tested.

5.3.1.2 Entrypoint from ART to LLVM

The first step of the IR translator is to generate an entrypoint for the transition of the execution flow from ART code to LLVM. This follows a similar calling convention with the default backend and is shown in Table 5.1. The arguments are packed into a 32 bit integer array. The `double` and `long` primitive types are taking two consecutive slots. The return result, in case of a non `void` method call, will be stored in a `union` that is passed as the second argument to the LLVM code. The `Thread` and `Method` arguments will be used for accessing various ART entrypoints, methods, classes, strings, or other caching mechanisms. Finally, an offset to a special ART cache, called *boot image*, will be used to optimize access to immutable objects in some specific cases.

5.3.1.3 Translating HGraph nodes to LLVM bitcode

The Android compiler's IR is not in a materialized form. Instead, it consists of several nodes that reside in main memory in a graph-based structure called *HGraph*. Once the *HGraph* of a method is generated, it is then optimized using the same default optimization passes that the Android compiler normally runs. Then, the optimized *HGraph* is translated into LLVM bitcode. Initially, the LLVM backend generates placeholders for

the basic blocks, and then for the *phis*, if any. Subsequently, it visits each basic block in a *post-order traversal* and populates the instructions. This IR-to-IR translation is done by generating code for each node using relevant instructions from the LLVM API. For any nodes that are not yet supported by the LLVM backend, the compiler driver will present a relevant message and bail out. The supported instructions follow the same basic block structure with the *HGraph* nodes for simplicity, which avoids any modifications to the branching between the phis and the basic blocks. This facilitates compiler testing and debugging. Any operation that requires additional branching is wrapped into a separate method that contains hint metadata for *inlining*. One such example is shown in Listing 5.5. Once all of the instructions are in place, the phi nodes are populated and the basic blocks are linked with relevant branch instructions.

5.3.2 Android-specific LLVM optimization passes

The LLVM backend implements two Android-specific optimization passes. The first one attempts to decrease loop overheads by removing check calls that become redundant after specific code transformations. Loops on Android are flooded with additional check instructions for heavy-weight runtime operations, like garbage collection. The second one attempts to increase the amount of the generated code by converting specific JNI calls to LLVM IR.

The garbage collection optimization, named *post-unroll*, runs after loop restructuring transformation passes. Consider the simple Java code sample on Listing 5.1, that initializes an array based on a given parameter. When the Android compiler generates its *HGraph*, it does not know at compilation time when, if ever, this loop might exit. Therefore, it is obligated to insert a check call in each loop's body to transfer control to the Android runtime. Once the IR-to-IR translation pass runs, it will result in the LLVM IR shown in Listing 5.2. The loop body contains two instructions that perform useful work (initializing array indices) and three that relate to operations required for looping. One of them is the check call, shown as `SuspendCheck` (line *20*). During that check, the runtime decides whether operations like garbage collection must run, and does so if needed. A single check per loop is sufficient, but passes like loop unrolling do not have this knowledge. Therefore, when a loop's body is duplicated the runtime check call is also unnecessarily duplicated.

The LLVM backend specially annotates bitcode at the check call, as shown on the Listing 5.3 (line *2*). It also places prediction weights on branches (line *6*), in favor of

skipping the checks. Then, after transformations that duplicate a loop's body, LLVM performs an Android-specific simplification pass, shown by Listing 5.4. During that pass, any redundant checks are eliminated. Those are annotated calls that belong in duplicated basic blocks. As shown in Section 5.5, this *post-unroll* simplification pass can readily extract additional speedup after loop unroll passes. Also, it can potentially be expanded for eliminating additional loop-related overheads, like some specific cases of array *bounds checking*.

```
1 public class Demo {
2     double a[] = new double[10];
3
4     void InitArray(double y) {
5         for (int i = 0; i < a.length; i++) {
6             a[i] = y;
7         }
8     }
9 }
```

Listing 5.1: A Java loop that initializes an array based on the given parameter value.

```
1  define void @Demo.InitArray(i8* %this, double %value) {
2  entry_llvm:
3    %thread = call i8* asm "mov $0, x19", "=r"() ; Android Thread
4    br label %init_block
5  init_block:
6    %GetObjInstance = ; .. ART entrypoint (offset from %thread)
7    %array = call i8* %GetObjInstance(i32 %array_offset, i8* %this)
8    call void @NullCheck(i8* %array)
9    %array_length= ; .. relevant pointer arithmetic/casting
10   br label %loop_check
11 loop_check:
12   %phi42 = phi i32 [ 0, %init_block ], [ %index, %loop_body]
13   %is_done = icmp uge i32 %phi42, %array_length
14   br i32 %is_done, label %loop_exit, label %loop_body
15 loop_body:
16   %index_ptr = ; .. relevant pointer arithmetic/casting
17   store double %value, double* %index_ptr
18   %index= add nsw i32 %phi42, 1 ; loop increment
19   ; Give ART the chance to perform operations like GC
20   call void @SuspendCheck() ;
21   br label %loop_check
22 loop_exit:
23   ret void
24 }
```

Listing 5.2: LLVM IR generated for the Java code of Listing 5.1. The instructions that are required for the loop operation itself take 60% all instructions in the `loop_body` basic block. These are loop-related overheads. On Android specifically, some additional checks are required, like the call at line 20.

```
1  ; Function Attrs: alwaysinline readnone
2  define void @SuspendCheck() #5 !android.check.suspend !1 {
3  entry:
4    %0 = call i16 @LoadStateAndFlags()
5    %1 = icmp eq i16 %0, 0
6    br i1 %1, label %skip, label %test_suspend, !prof !2
7  test_suspend:
8    %entrypointTestSuspend = call i8* @GetEntrypointTestSuspend()
9    call i8* %entrypointTestSuspend()
10   ret void
11 skip:
12   ret void
13 }
14
15 !1 = !{!"android_optimization_suspendcheck"}
16 !2 = !{!"branch_weights", i32 100, i32 0}
```

Listing 5.3: Specially annotating LLVM bitcode on check-calls for performing heavy-weight runtime operations (lines 2 & 15). This check call can become redundant if it resides in a loop body that gets duplicated due to some previously applied transformations. Branch prediction also becomes biased towards skipping the check (lines 6 & 16).

```
LoopUnrollResult llvm::UnrollLoop(
    Loop *L, UnrollLoopOptions ULO, LoopInfo *LI, ..) {

  // LLVM code for unrolling a loop:
  // .. unroll body of loop
  // .. preserve loop-simplified form
  // .. setup branches to connect new basic blocks
  // .. merge adjacent blocks if possible
  // .. simplify unrolled loop:
  // .... constant propagation
  // .... dead code elimination

  // post-unroll simplification
  for (BasicBlock *BB : L->getBlocks()) {
    // not the original block
    if (IsDuplicated(BB)) {
      for (Instruction &I : *BB)
        if (isa<CallInst>(I))
          Function *F = cast<CallInst>(I).getCalledFunction();
          if (F->getMetadata("android.check.suspend"))
            I->erase();
    }
  }

   // .. other code
   return unroll_result;
}
```

Listing 5.4: Eliminating redundant check-calls of heavy-weight runtime operations, like garbage collection. This simplification pass runs after transformations that duplicate a loop's body. Additional speedups can be extracted on any code that already benefits from such transformations. This pass runs on specially annotated code.

The second optimization that was implemented tries to increase the amount of the generated LLVM code by translating some of the JNI methods. This is done by converting particular math-library calls to LLVM IR, either by using LLVM intrinsics, or by providing an implementation of the method that matches the original code. This can also be expanded to methods outside of the math library. By doing so, not only it avoids the overhead of setting and returning from a JNI environment [KUR+01], but it also increases the amount of code that can be compiled and optimized. With more available code, compiler optimization only becomes more effective [COO+86a].

5.3.3 Assembling and running LLVM-generated code

Once the LLVM IR is generated for specific hot regions of code, then it should be optimized, assembled, and finally executed in the place of the original Android-compiled code. The complete compilation toolchain is visualized in Figure 5.1 and is described in the remainder of this section.

Usually a hot region of an application contains multiple methods. The bitcodes of these methods are linked together and compiled using the -O3 optimization level. Any special registers are reserved, like the *Thread Register* or the garbage collector's *Mark Register* [GOO20c]. Subsequently, the code is assembled and finally linked against some libraries, including the Android runtime, and C/C++ libraries. Constructors and initialization routines for the C runtime are also linked at this stage. The output of the linker is a shared object file that is dynamically linked during an application's bootstrap along with its other resources. All these tools were compiled to AOSP modules from the LLVM sources using the Soong build system [GOO20f] and were subsequently shipped to an Android device. They are invoked through an interface that was baked into the Android compiler driver (`dex2oat`). At runtime, if there is LLVM code available for a particular method then it is chosen for carrying out the execution.

5.3.4 Current limitations of the LLVM backend

Despite being able to outperform the Android compiled code for most cases, there is ample room for improvement in the LLVM backend implementation. There are several instructions not optimally implemented that rely more often than they should on the much slower runtime. Improving those can potentially have a noticeable effect on the runtime performance. This section explores some of these cases.

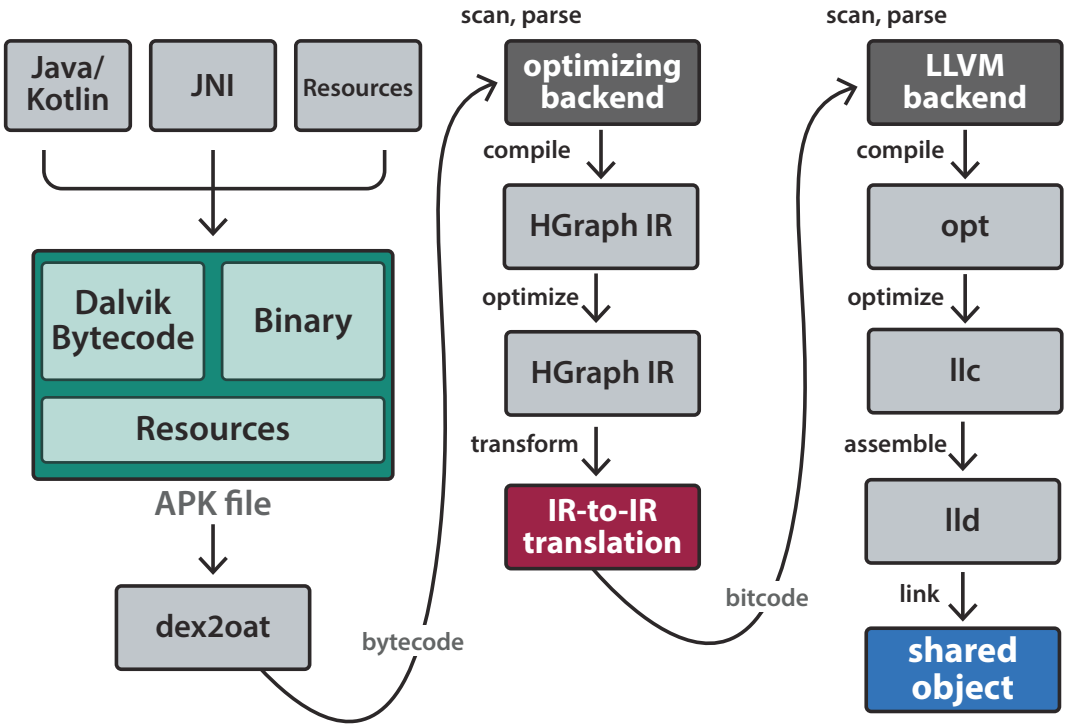

Figure 5.1: The LLVM backend implementation is the first alternative compilation toolchain for Android, able to aggressively optimize Dalvik bytecode through a significantly larger code optimization space. Initially, an application (APK file) is passed to the *dex2oat* compiler driver, which invokes the default backend (optimizing) to generate its IR. The IR, called *HGraph*, is then optimized using the default Android optimizations. Then, an IR-to-IR translation pass transforms the *HGraph* to LLVM bitcode. The bitcode is then aggressively optimized by the LLVM backend, using the *opt* compiler and the *llc* assembler. Finally, the resulting binary is linked using *lld* into a *shared object*, which can be invoked at runtime as an alternative to the Android-compiled code.

```
1  ; Function Attrs: inlinehint
2  define i8 @InstanceOf(i8* %object, i8* %class) {
3  entry:
4    %thread = call i8* asm "mov $0, x19", "=r"()
5    %isNull = icmp eq i8* %object, null
6    br i1 %isNull, label %_null, label %_not_null
7  _null:
8    ret i8 0
9  _not_null:
10   %object_class = ..      ; get object's class from LLVM code
11   %isSame = icmp eq i8* %object_class, %class
12   br i1 %isSame, label %_same, label %_diff
13 _same:                    ; preds = %not_null
14   ret i8 1
15 _diff:                    ; Slower runtime path
16   %entrypoint = ..        ; offset from %thread
17   %InstanceOfSLOW = call i8 %entrypoint(i8* %object, i8* %class)
18   ret i8 %InstanceOf_SLOW
19 }
```

Listing 5.5: LLVM IR generated for the `InstanceOf` instruction. It is wrapped into a method that is hinted for inlining. If the class of the object (`object_class`) is the same as the input class (`class`), then all of the code is executed fast inside LLVM. Otherwise, it calls the much slower runtime (line 17). This is an example of an instruction that is not fully optimized by the current version of the LLVM backend.

The instructions that have runtime interactions need to access particular ART structures. This is done through immutable offsets from the Thread Register (TR). That particular register is initially passed to the LLVM code as an argument to its entrypoint. Subsequently, it is stored in a physical register that is reserved in LLVM, matching the default behavior of the Android compiler. Nevertheless, any instructions that utilize the TR are not as optimized as the Android compiled code. This is because LLVM currently does not support the notion of Global Register Variables [GCC20b]. Instead, each time the TR is needed it is accessed through LLVM inline assembly. Despite applying any relevant inline assembly constraints [LLV20b], the compiler still has limited semantics over that code. As a result, it fails to correlate and optimize different inlined assembly codes through transformations, such as instruction reordering or merging. This limitation applies to other specially reserved registers as well, like the *Mark Register*. Unsurprisingly, all special registers are frequently used, which is

why they permanently occupy a physical register in the first place.

Another class of instructions that can be further improved concerns the caching mechanism of special ART components, such as methods, classes, or strings. Similarly with special registers, these are frequently used instructions in Android applications. The machine code generated by the default backend is packed into an OAT file that is based on ELF, as described in Section 2.4. It has a built-in caching mechanism, which cannot be directly accessed from the LLVM code. There are three ways of retrieving an ART component and two of those are utilizing such caching. The first, is through a special *bss* segment for any object that might occasionally be moved by the garbage collector. This segment is not the typical *bss* segment in ELF files, as the C Runtime objects are stored separately from the ART objects. The second type concerns objects whose address is immutable throughout the same run of an application. They are stored in another special cache called *boot image*. Lastly, an object might always have to be retrieved through the runtime. For the last case, LLVM matches the behavior of the Android compiler. For the first 2 cases, the LLVM backend utilizes a sub-optimal workaround by indirectly accessing any relevant caches. Additional effort is required to match the caching efficacy of the Android compiler. Firstly, the OAT files must embed the LLVM code and also be extended to provide any relevant runtime support. Secondly, the code generator should emit empty cache-slot placeholders as not all of the information is statically known. These slots will then have to be populated with cached objects after the application is bootstrapped.

A third and final example of a not fully optimized instruction is shown in Listing 5.5. If the class of the input object (resolved at line 10) does not match the input class (`class` at line 2), then some additional checks have to be performed. This transfers the execution flow to line 15, where LLVM proceeds with a slower runtime call. When this instruction gets fully implemented into LLVM IR, it would walk instead the class hierarchy of the object without ever having to call the runtime. This increases the amount of code that LLVM can optimize and decreases the runtime dependencies, which can result in performance improvements. Similar examples of slower operations include unimplemented intrinsics that are likewise invoked through the runtime.

5.4 Experimental Setup

The goal of the LLVM backend implementation is to optimize real Android applications by exposing them to aggressive compilation through a significantly larger code

Component	Type	Dependencies	Description
libart	library	-	Android Runtime (ART) library
libLLVM	library	-	LLVM library
llvm-link	binary	libLLVM	Links several LLVM bitcode files
opt	binary	libLLVM	Optimizer and analyzer of LLVM bitcode files
llc	binary	libLLVM	Generates assembly from LLVM bitcode files
lld	binary	libLLVM	Linker of assembled objects
libart-compiler	library	libart, libLLVM	Android compiler library
dex2oat	binary	libart, libart-compiler	Android compiler driver

Table 5.2: Listing the binary tools and library dependencies of for the compilation toolchain and runtime support required for generating and executing LLVM compiled code from Android applications. Those are overlaid on top of the otherwise read-only `/system` folder using modules built by a third-party tool, called Magisk [TOP20].

transformation space. A system was implemented for comparing the LLVM backend against the default backend of the Android compiler. Both benchmarks and interactive applications were used throughout the experiments. The system runs on *Android 10* mobile devices. It was evaluated on a recent *Google Pixel 4* device. Its processor unit is a *Qualcomm SDM855 Snapdragon 855*. It consists of eight *Kryo 485* cores, each configured with a different maximum clock frequency, ranging from 1.78 GHz up to 2.84 GHz. To reduce the measurement noise during the experimental evaluation, the same steps for setting up the execution environment in the previous chapter were applied here as well (see Section 4.4).

The LLVM backend uses the Android 10 default compiler backend, called *optimizing*, to generate the *HGraph* nodes. Then it performs an IR-to-IR translation which outputs LLVM bitcode based on LLVM version 10. The LLVM project's source code was imported as a new module in the AOSP Soong build system, since other modules depend on an already included obsolete version of LLVM, version 3.6 as of the time of writing [AND21d]. A shared library of LLVM was cross-compiled for `arm64` along with any other necessary binary tools and libraries, as shown in Table 5.2. All these software components were installed on an Android device using a slightly modified version of an open-source third-party framework, called Magisk [TOP20]. With cus-

Type	Name	Description
Scimark [POZ+04]	FFT	Fast Fourrier Transform
	SOR	Jacobi Successive Over-relaxation
	MonteCarlo	Estimates π value
	Sparse matmult	Indirection and addressing
	LU	Linear algebra kernels
Art	Sieve [NIH20]	Lists prime numbers
	BubbleSort [ALG20]	Simple sorting algorithm
	SelectionSort [ALG20]	Simple sorting algorithm
	Linpack [DON+79]	Numerical linear algebra
	Fibonacci.iter [ISH20]	Fibonacci sequence iterative
	Fibonacci.recv [ISH20]	Fibonacci sequence recursive
	Dhrystone [WEI84]	Representative general CPU performance
Interactive	MaterialLife [SOR20]	Game of life
	4inaRow [APP20]	Puzzle Game
	DroidFish [ÖST20]	Chess Game
	ColorOverflow [VEL20b]	Strategic Game
	Brainstonz [VEL20a]	Board Game
	Blokish [SCO20]	Board Game
	Svarka Calculator [VEL20d]	Generates odds for a card game
	Reversi Android [FEL20]	Board Game
	Poker Odds (Vitosha) [VEL20c]	Statistical analysis for poker cards

Table 5.3: Android applications used for the experimental evaluation of the LLVM backend. They were either benchmark or interactive ones. For benchmarks, the Scimark benchmark suite was used, as well other benchmarks that have been periodically used by Google or third parties for the evaluation of the Android compiler. The interactive applications were found in online software stores.

tom Magisk modules, the new or modified components were overlaid at boot time on top of the otherwise *read-only* `/system` directory on the device. The LLVM library is linked against the Android compiler.

For the experiments presented in the next section, real Android applications were used that are categorized into three types, as listed in Table 5.3. The first is the *Sci-*

mark benchmark suite. The second, named *Art*, contains benchmarks that have been used in the past by Google or third parties for the evaluation of the Android compiler. The third, named *Interactive*, contains 9 real interactive Android applications, found in online software stores like the *Google Play* [GOO21b] or the *F-Droid* [F-D21]. The reported speedups are based on 30 evaluations. The plots show 95% confidence intervals where applicable.

Any random initialization of the used applications is disabled, so that the applications start in approximately the same state throughout the experiments and perform a similar workload. This makes it easier to estimate the speedup with any kind of certainty. For benchmark applications, this is straightforward as the inputs are fixed. For interactive ones, this is more complicated. The logic of each application is being followed, in order to bring its state to a similar point each time, including manually interacting with the application before taking a measurement.

For the execution time measurements of interactive applications deciding the start and end point of the measurement is a balancing act. For a fair and representative measurement, the speedup of the interactive applications in this and the next section, consider also the code that surrounds the hot region, i.e. code that is not optimized by the system. At the same time it is desired that all time measurements capture similar behaviors. A longer execution is more likely to be affected by random high impact events, like GC, or by deviating system and user behavior. As a compromise, a certain number of iterations of the conceptual main loop of the application is measured. The main loop is an actual code loop in some cases, a conceptual loop including user interaction in others. The clock measuring the thread time is used, which discards waiting and sleeping time, if any.

Even for a small number of interactive applications, this was a very tedious process. This reaffirms one of the motivating cases behind this thesis: deterministic and repeatable evaluations without replaying are non-trivial and require significant effort from application developers.

5.5 Experimental Results

The quality of the IR-to-IR translation pass to LLVM bitcode is evaluated against the default Android compiler with two sets of experiments. The effectiveness of the *post-unroll* optimization is evaluated with an additional third set. The first set shows the amount of code that the LLVM backend is able to process. The effectiveness of the

applied optimizations is restricted to code that can be translated into LLVM IR. The second set shows the achieved performance improvements when using the highest optimization level of LLVM. The presented speedups show the performance gains over the entire execution runtime. The results are based on 21 applications, either benchmarks or real interactive ones. The third and final set compares the effectiveness of the *post-unroll* optimization on top of the default loop unrolling pass of LLVM. Different unroll factors were used on 12 benchmark applications.

5.5.1 Runtime code breakdown

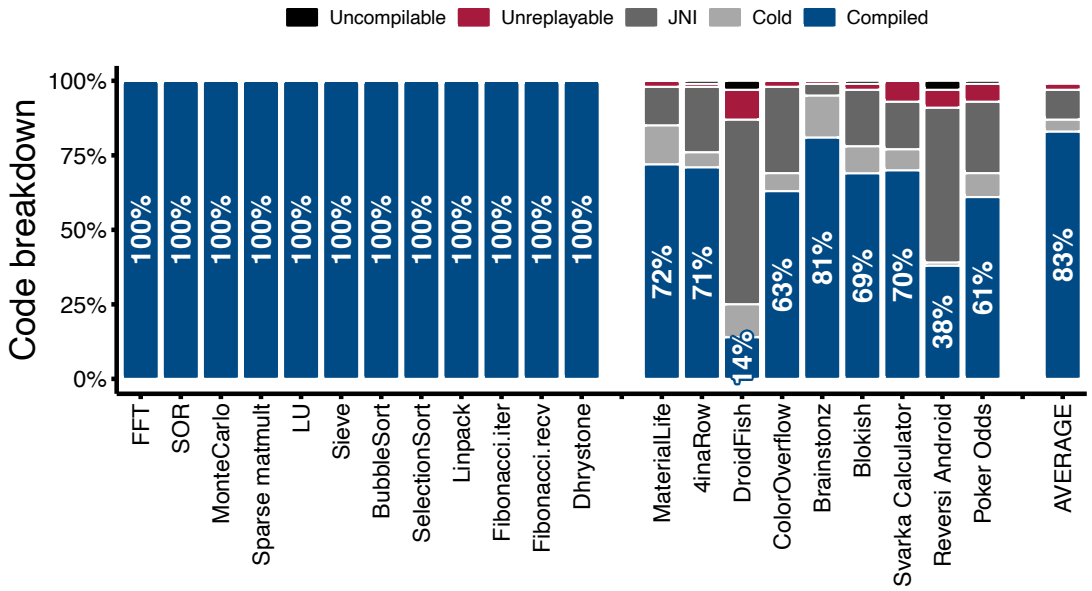

Figure 5.2: Runtime code breakdown that was measured online with a sample-based profiler for 12 benchmark (left) and 9 interactive applications (right). Compiled is code that can be processed by the LLVM backend. Cold is code that falls outside of the *hot region*. Uncompilable are *pathological* code cases that cannot be processed by the Android backend, and therefore by the LLVM backend. Finally, JNI is code that was written using the C / C++ language and belongs either to the application or the Android framework.

Figure 5.2 shows the runtime code breakdown that was measured online using a sample-based profiler. *Compiled* is code that the LLVM backend is able to transform into LLVM bitcode. It may still contain code segments that are executed through the runtime, like unimplemented intrinsics or not fully implemented instructions. Some examples of such cases are detailed in Section 5.3.4. *Cold*, is code that is not worth

optimizing, therefore it falls outside of the hot region. *Uncompilable*, is code that is marked as being a *pathological* case by the Android compiler backend as it cannot process it [AND20]. Finally, JNI is code that is written using C or C++ and is already in a binary form.

For the 12 benchmark applications, the LLVM backend was able to process the entirety of the code. For the 9 real interactive applications though, the amount of code that passes through the LLVM backend is more limited. *DroidFish* has the least amount of code that is compiled at 14%, while *Brainstonz* has the highest percentage of compiled code at 81%. Cold code ranges from 5% to 14%. JNI code takes a higher portion of the runtime in many cases. The highest amount of native code is executed by *DroidFish*, with as much as 72% of the total runtime. The application with the least amount of JNI code is *MaterialLife*, with 15% of the runtime. Finally, the code that cannot be processed by the Android compiler is generally low, ranging from 1% to 3%.

On average, for both benchmarks and real interactive applications, 83% of the runtime executed code can be compiled by the LLVM backend. However, this percentage includes cases where code is not optimally implemented yet and resorts more often than it should to slower runtime routines. As the LLVM backend implementation matures, it is expected that both the quality and the amount of the compiled code will increase, which will further improve the effectiveness of any subsequently applied code transformations.

5.5.2 Using LLVM to optimize Android applications

This experiment showcases how a well-established compiler infrastructure, like the LLVM, is able to improve performance against the Android compiler, which is limited to a handful of code transformations. The same set of applications with the previous experiment was used.

Figure 5.3 shows the speedup observed when compiling Android code using the LLVM backend for benchmark applications. It is compared against the baseline execution time that users get out of the box, which is based on the default Android compiler backend. Code generated through the LLVM backend is transformed using `-O3`, which represents the highest optimization level that can yield performance improvements on a wide range of applications.

LLVM is able to outperform the Android baseline in most cases. The performance ranges from an 11% slowdown to a 66% speedup. Most of the benchmarks had per-

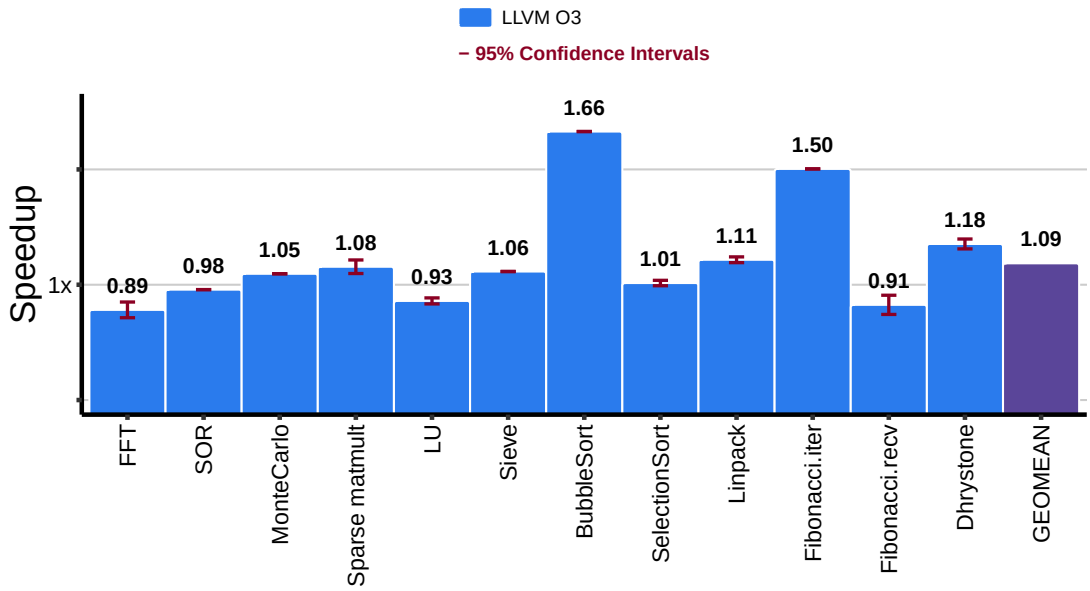

Figure 5.3: Speedups relative to the default Android backend for 12 benchmark applications. LLVM code was transformed using the `-O3` optimization level. The performance ranges widely, from an 11% slowdown to a 66% speedup. Most benchmarks had performance improvements, while on average the LLVM backend was able to produce 9% faster code.

formance improvements with an average speedup of 9%. Two of the benchmarks had similar performance with the Android compiled code, while three under-performed. Specifically, *FFT*, *SOR*, *LU*, and *Fibonacci.recv* had worse execution times. This was due to having an increased number of instructions that currently have sub-optimal implementations when compared to the code generated by the baseline. On top of that, almost all of the benchmarks had a significant amount of operations encapsulated in loops. Despite the implemented *post-unroll* optimization (described in Section 5.3.2) LLVM was not able to use it to its advantage. This was due to the fact that the default compiler heuristics, even in the highest optimization level (i.e., `-O3`), are quite conservative when it comes to controversial optimizations such as loop unrolling (see next subsection). As a result, *post-unroll* never engages.

The speedup of the LLVM backend against the baseline compiler for real interactive applications is shown in Figure 5.4. The performance ranges from a 1% slowdown to a 9% speedup. All applications, except *ColorOverflow*, had an improved performance with a speedup average of 3%. While this is noticeably lower than the benchmark applications, it was due to considering wider runtime regions. This includes

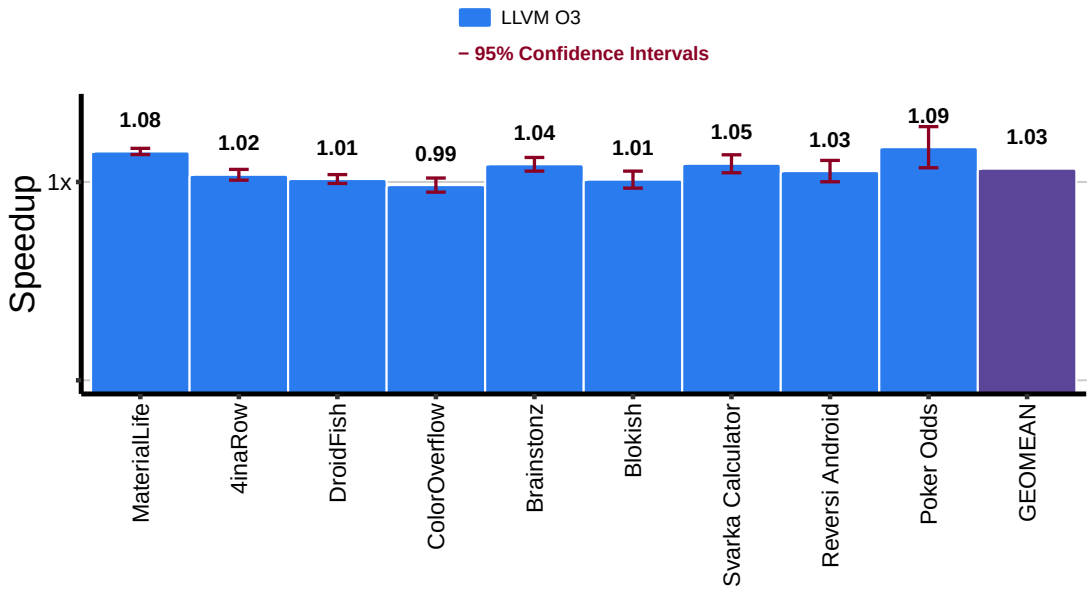

Figure 5.4: Speedups relative to the default Android backend for 9 real interactive Android applications. LLVM code was transformed using the `-O3` optimization level. Performance ranged from a 1% slowdown to a 9% speedup, while on average it was improved by 3%. The reported speedups consider the whole execution runtime of each application (see Section 5.5.1). As the IR-to-IR translation pass matures, it is expected that the performance gains will improve even further.

code that was not optimized by LLVM, as shown by the previous experiment. Other important factors are the current limitations of the backend (see Section 5.3.4).

5.5.3 Post-unroll GC optimization

The goal of this experiment is twofold. The first is to demonstrate the effectiveness of the post-unroll GC optimization pass that was presented in Section 5.3.2, on 12 benchmark applications. It runs after transformations that duplicate a loop's body and eliminates redundant high-overhead checks. The second, is to show that even a highly optimizing compiler cannot decide on its own when it comes to applying aggressive yet controversial optimizations, like *loop unrolling*.

Figure 5.5 shows speedups for 12 benchmarks when applying loop unrolling with different factors: 2, 4, 8, and 10. *Default* is the loop unroll transformation of LLVM and *PostUnroll* the additionally implemented pass applied on top of *Default*. No other optimizations were enabled. *Fibonacci.recv* does not have any loops so it is ignored in this discussion. On average, *Default* had 29% to 45% speedup, depending on the

unroll factor. *PostUnroll* improved performance even further, with noticeable averages between 42% and 58%. *Fibonacci.iter* and *Sieve* are notable examples where unrolling is very effective with up to 2.15x and 1.82x speedups respectively for *Default*. *PostUnroll* is even better, outperforming *Default* on each factor, reaching a maximum 30% of additional performance. *Dhrystone* on the other hand, does not benefit very much from unrolling with *Default* having speedups between 2% and 5%. Even so, *PostUnroll* squeezed additional performance with speedups ranging between 9% and 21%. Except *SOR*, on each other benchmark and any unroll factor combination, *PostUnroll* was improving performance. On *Sparse matmult* in particular, it had remarkable improvements, while on *MonteCarlo* it provided a maximum of 38% of additional performance. *SOR* was the only benchmark where *PostUnroll* was suboptimal, giving away a 5% of the gains from *Default*. On a quick glance, this does not make much sense as *PostUnroll* only eliminates some redundant instructions. However, it is well-known that transformations affecting code structure or size can often interfere with the underlying CPU caching mechanisms. This adverse effect was also noticed for around half of the benchmarks, where speedup dropped as the unroll factor increased to 8 or 10. This affected both *Default* and *PostUnroll*.

While it is established that loop unrolling yields significant speedups and *PostUnroll* can readily improve it even further, it is equally important to understand why compilers are hesitant when it comes to applying such controversial transformations. Figure 5.6 shows the size increments on the same unroll factors and benchmarks with the previous experiment. *None* did not perform any unrolling. *Fibonacci.recv* had no size changes as it does not contain any loops. Most of the benchmarks were between 9KB and 13KB in size and had a handful of additional size unit increments. The highest of those was *Sieve* with 90% code increase between *None* and *10* unroll factors. *Fibonacci.iter* had the lowest increase of 11%. Another observation is that these benchmarks had similar code sizes despite increasing their unroll factors. This might either relate to increased *constant folding* and *dead code elimination* passes or due to practical reasons, i.e., compiler's inability to unroll due to *trip count* mismatch [LLV21a]. The remaining 4 benchmarks however, have exploded in size. Several loops (some of which are nested) and more complex code structure can play a significant role. For their highest unroll counts, *Dhrystone* became almost 5 times larger, *FFT* and *LU* around 11 times larger, while *Linpack* more than 17 times larger. This significantly shifted the average size increases to 8x. With such high storage overheads and with only static information, a compiler cannot take such high-risk decisions a priori.

In retrospect, when considering both Figures 5.5 and 5.6, one becomes more confident on whether, and how much, to unroll a loop. *Fibonacci.iter* for example, despite having the lowest size increase, benefited the most with more than 2x speedup increases. One could have even decided to unroll it even further. The same applies for *SelectionSort*, *BubbleSort*, *Sieve*, *Sparce matmult*, and *MonteCarlo*. All these had good ratios between performance gains and code size increments. *LU* going from 2 to 4 unroll factors, had a 2.5x size increase while performance improved by only 7%. For *Linpack* one could have stopped at 2 or 4 unroll factors, depending maybe on storage size limitations. With *FFT* and *Dhrystone* the decision is more obvious, as the size was steadily increasing while the performance was declining.

The bottom-line is that a compiler cannot take high-risk decisions. Sizes are known at static time but performance is not. Increasing the unroll factor does not necessarily mean increased performance, while the size overheads vary wildly. This trade-off is just one such example. Several other factors key in, while the default heuristics are tuned for the common case. On Android, given that loops have significantly additional overheads, *PostUnroll* should be very effective in nearly all cases. Actually it improved speedups over the default unrolling for each factor and benchmark that contained loops, except the bizarre case of *SOR*. Despite its effectiveness, even the highest optimization level of LLVM cannot utilize this pass (see Section 5.5.2). This signifies how important actual evaluations are, and why an adaptive decision-making framework is needed to guide aggressive optimization of Android applications.

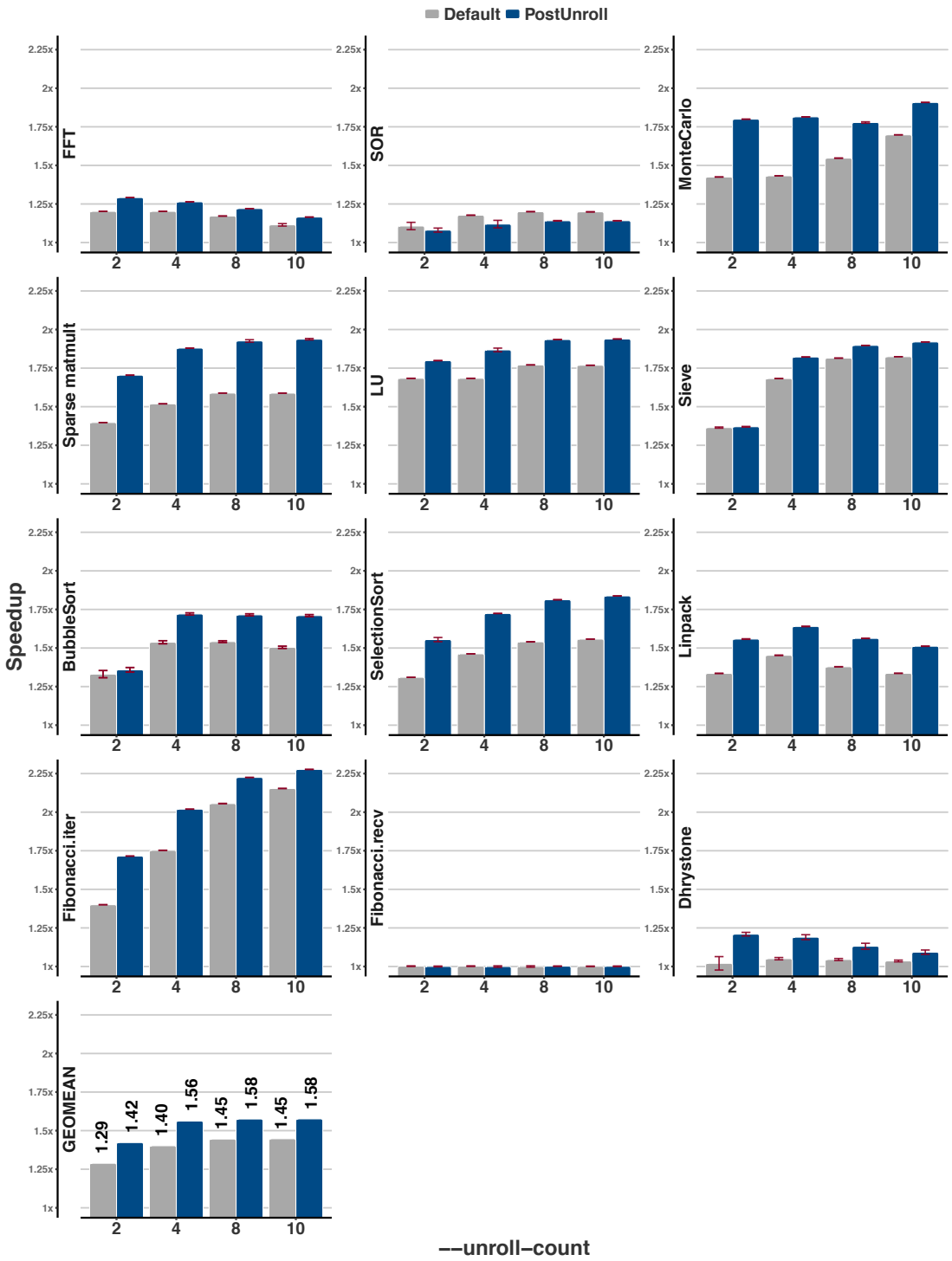

Figure 5.5: Comparing speedups of *loop unrolling* when *post-unroll* optimization is enabled or not. Different unroll counts were used against a baseline that is using LLVM `-O0` without any unrolling. In general, loop unrolling was effective in all benchmark applications, with the notable exception of *Fibonacci.recv* that does not contain any loops. The post-unroll optimization, described in Section 5.3.2, outperformed the default unroll optimization at the remaining 9 out of 10 cases. On average, it was faster on each unroll count, providing approximately additional 14% performance on top of loop unrolling, and a 38% on *MonteCarlo* with unroll count set to 2.

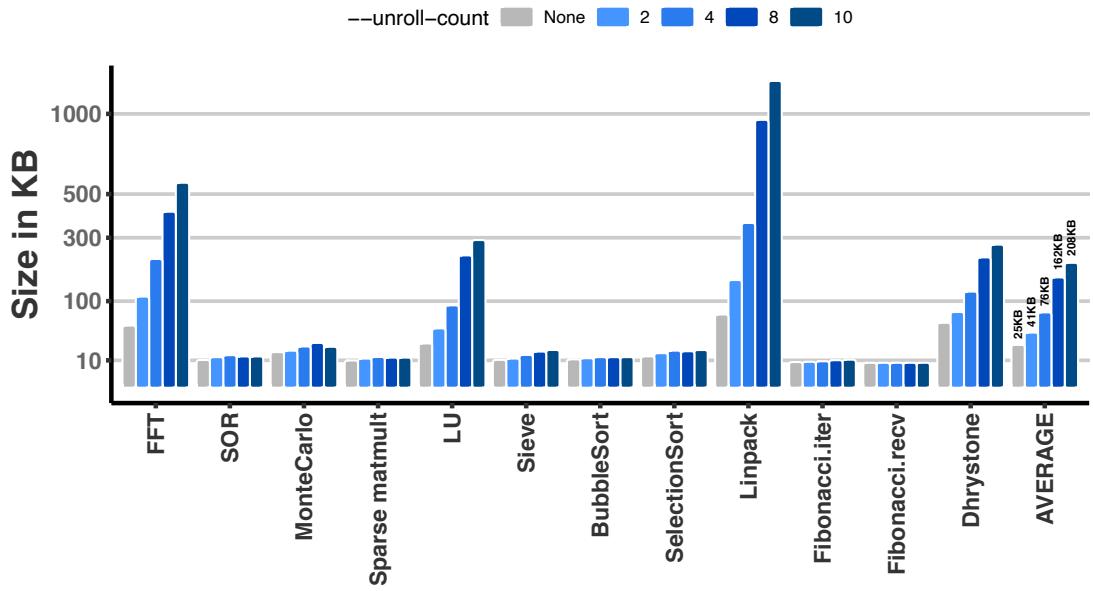

Figure 5.6: Comparing binary sizes of LLVM generated code using different loop unroll factors. No other optimization was applied. *None*, shown with the lightest shade, did not perform any unrolling. Most of the benchmarks had less than a handful of size unit increases for any unroll count. 4 out of 12 benchmarks had considerable size increases bringing up the average increases of the highest unroll count to a hefty 8x. Their highest unroll counts ranged from 5x to 15x, with Linpack exploding from 72KB to 1.2MB. While there can be several scenarios under which the performance gains outweigh the code size increases, this is a risk compilers are unwilling to take. Even in their highest optimization levels (e.g., `-O3`), compilers only hint for unrolling. The hint must then pass through very conservative heuristics before materializing.

5.6 Summary

This chapter investigates whether a more aggressive compilation toolchain can generate better binaries for Android applications. The default Android backend is lacking sophisticated code transformations as it includes only a handful of generally conservative optimizations. A novel backend was presented, which performs an IR-to-IR translation from the Android default compiler's IR to LLVM bitcode. This exposes interactive applications to a well-established compiler infrastructure and enables aggressive optimizations through a significantly bigger code transformation space.

The time consuming and computationally intensive code regions of applications are automatically identified without incurring noticeable overheads. Subsequently, they are targeted for optimization by the LLVM backend. Despite currently providing several sub-optimal implementations for various instructions, as described in Section 5.3.4, the LLVM backend is able to improve 16 out of the 21 tested applications.

The backend was evaluated on real Android applications, both benchmarks and interactive ones. On average the applications were improved by 7%. For the cases where the LLVM backend under-performed, there was an increased amount of instructions that are not currently implemented as optimally as the default backend. Additionally, the implemented *post-unroll* optimization was not being exploited by LLVM even at its most aggressive setting. This pass was extremely successful for almost any code that contained loops, adding in its highest case 38% of additional speedup on top of the default loop unroll. Yet, the default compiler heuristics are being quite conservative when it comes to unrolling, as it can cause considerable size overheads while speedups are not guaranteed. This is one of many examples that show why even a highly optimizing compiler cannot apply aggressive transformations on its own. Regardless, it is still noteworthy that the implemented backend was able to outperform, for most applications, a compiler that was purposely built from scratch and tailored to the Android runtime.

The next chapter presents an optimization framework for real interactive mobile applications able to take advantage to a much higher extent such a highly optimizing compiler. It evolves the ideas presented in Chapter 4, while gracefully solving most of its limitations. It realizes a highly scalable, user and application transparent aggressive optimizer. As the IR-to-IR translation matures, it is expected that the performance gains (presented in this and the next chapter) will increase even further.

Chapter 6

Crowd-Sourced, Input-driven optimization for interactive applications

6.1 Introduction

Despite being a well-established code optimization technique, iterative compilation is still impracticable on real Android applications. While the approach presented in Chapter 4 can operate on the mobile environment, it only works for C benchmarks. Additionally, it targets code regions that are manually identified, analyzed, and extracted. Also, the correctness of the iterative compilation findings are manually verified, requiring a significant effort from application developers. On top of that, interactive applications are fundamentally different from C programs, operating under a complex runtime environment that requires different approaches for input capture and replay.

This chapter describes the implementation of an optimization framework tailored for real Android applications. Two capture mechanisms were implemented. The first one is based on an adapted *Page* capture mechanism that works for targeted hot regions of interactive applications. The new mechanism removes from the captures any memory pages that were found to be immutable. It also performs heap object intersection with the captured data, without introducing any additional overheads, to minimize the capture sizes even further. The second one focuses only on the runtime overheads, at the cost of higher storage requirements. A capture is now canceled when complex runtime operations might interfere with its time or space complexities. This is because the capture mechanism, despite being invoked infrequently, aims to always

remain unnoticeable from the users at all times. Several optimizations have also been implemented. The code regions targeted by the optimizer are now automatically identified through static bytecode analysis. These regions must be able to replay accurately, without causing any side-effects while doing so. Additionally, the timing of a capture as well the capture frequency can be adjusted, which enables captured input diversity. A new replay mechanism was also developed. It starts as a purely C program and gradually transforms itself into a partial Android process. It works well alongside memory-shuffling security mechanisms, which are critical components of any modern operating system, including Android. Once everything is restored, the mechanism can replay the targeted hot region with different types of code. The first, is Android-compiled code, which is generated using the default compiler backend. It is used as one of the baselines during the experimental evaluation and it represents what Android offers out of the box. The second, is interpretation and it is used to extract information from the captures in order to automate correctness verification and further optimize the code. The last, is code generated using the LLVM backend, which was introduced in the previous chapter. For this last type, the presented system uses several different binary versions. One that represents the best code LLVM offers out of the box, and several others that are discovered through genetic search over its transformation space. Each version is evaluated offline using a replay-based iterative compilation. A previously captured input is restored for each evaluation through the replay mechanism. This ensures the region's workload remains the same, making comparisons between different code types and versions sound. Evaluations happen offline, while the device is idle and charged, hiding from the users any side-effects from slower or erroneously optimized code. This, combined with a tighter control over the device environment, make the execution noise manageable. Ultimately, GA discovers better transformation sequences for each input, which are used to optimize the application code. On top of that, a crowd-sourcing module was implemented to significantly accelerate the offline evaluation efforts. It allows performing a collaborative search among different users, splitting the heavyweight evaluations required by iterative compilation. By leveraging the accumulated evaluation data, participating users can now discover better genomes at a fraction of the time when compared to individual user searches. This joint-search enables deeper searches with less effort from each participating user.

The proposed system is evaluated on 21 Android applications. The infrequent captures remain transparent from the users. The *Page* capture mechanism (presented at Chapter 4 and adapted to work for Android applications) and its improvement, called

Intersection, minimize the capture sizes. Both incur a slowdown of just 15ms on average. An alternative approach, named *Everything*, incurs only 4.6ms, as it does not take any action to minimize the captured state. *Page* requires 5.06MB of storage, while *Intersection* reduces that by an additional 64%. These sizes are manageable even for lower-end devices, especially when considering that snapshots are discarded once the optimization search has concluded. The capture *Everything* approach, requires on average 165MB of storage and it can be used for the cases where the device storage is not very limited.

Regarding optimization, the system readily outperforms the Android compiled code by a significant margin, with a 44% average speedup. The genetic algorithm kept discovering better genomes for multiple generations. During search, it also evaluated significantly slower code or code that produces a wrong outcome. While those are simply discarded by the presented offline replay-based evaluation, in an online system they can be detrimental to the user experience. The joint search achieves only 6% less performance that the individual user searches, while it requires only a fraction of a user's evaluation time. In particular, with 10 users the system was able to accelerate the search by 7x for the user that had the biggest workload. Additionally, the proposed approach is able to extract near optimal results even when a user participation lasts only a handful of minutes.

This chapter is organized as follows. Section 6.2 describes how a replayable hot region of an application is detected. Section 6.3 describes the capture mechanisms (*Pages* / *Intersection* and *Everything*) that store inputs of Android application hot regions, at different granularity. It is followed by a novel replay mechanism, described in Section 6.4. Section 6.4.1 presents how the generated code is automatically verified for correctness and Section 6.4.2 how it is further optimized with extracted information from the captures. The optimization search is performed using a genetic algorithm as described in Section 6.5. All these components, as well the LLVM backend from the previous section, are fused into a system that is described in Section 6.6. A crowdsourcing module that can accelerate the offline evaluations is described in Section 6.7. The experimental setup can be found in Section 6.8 and the experimental evaluation in Section 6.9. Finally, the concluding remarks of this chapter are in Section 6.10.

Algorithm 2: With static Dalvik-bytecode analysis, the methods that might introduce side-effects when replayed are identified. The identification algorithm is quite conservative. If any of the hot region methods reach a non-replayable method, then the hot region is excluded altogether. Algorithm 1 is extended to reflect this.

```
method: outermost method

def markNonReplayable(method):
    def f(m, visited):
        if m ∉ visited then
            add(visited, m)
            if io(m) or blocklist(m) then
                m.replayable ← False
            end
            foreach c ∈ m.callees() do
                f(c, visited)
                if not c.replayable then
                    m.replayable ← False
                end
            end
        end
    visited ← ∅
    f(method, visited)
```

6.2 Detecting replayable code regions

The system for C programs presented in Chapter 4 performed a manual inspection of a program's source code to ensure that only deterministic input sources are used. Otherwise, replaying those programs would have made comparisons between different evaluations as well code correctness verification, manual or not, nearly impossible. For the small set of chosen benchmarks, each of which consisted of a single function, this was a relatively easy task. For real applications, however, this process is not trivial and certainly cannot scale. Applications tend to have complex method call nesting, multiple dependencies on external or runtime-internal calls, or invoke code that is based on a different environment, like the JNI.

To overcome these limitations, a static analysis pass of the Dalvik bytecode was implemented and integrated into the Android compiler driver. As shown in Algorithm 2, the proposed approach automatically identifies any sources of non-deterministic input for a given application. Performing I/O is one example of non-determinism. Replaying it without any special infrastructure is either impossible or it can lead to an inconsistent outcome. Writing information to a storage device is even worse since it can corrupt the application's permanent state. Emulating such I/O is doable, however, it can incur significant overheads given the latency-sensitive environment of interactive applications. Other excluded sources of non-determinism are calls to clocks and pseudo-random number generators. Additionally, almost all of the JNI calls are aggressively blocklisted, regardless of whether they can be accurately replayed or not. Determining that their low-level code does not perform I/O and it is deterministic requires a significant amount of engineering work that is beyond the scope of this thesis. The only JNI calls that are not blocklisted are the ones that are replaced with intrinsics by LLVM, as described in Section 5.3.

6.3 Transparent input capture mechanisms

This section describes the input capture mechanisms that were implemented for interactive Android applications. The *Page* capture mechanism, presented in Section 4.2, was extended to support the Android runtime. A novel capture mechanism, called *Intersection*, significantly decreases the storage requirements of the *Pages* approach without requiring additional overheads. It achieves this by intersecting the hot region reachable objects with the input memory pages identified by the first approach. Fi-

nally, the capture *Everything* approach favors a simpler runtime operation with lower runtime overheads over storage size.

6.3.1 Page-Capture mechanism for interactive Applications

Capturing C programs is relatively straightforward, especially for the benchmarks in Section 4.4 that consisted of a single function without any external library dependencies. Capturing real Android applications, however, is a more involved process. There are multiple different resources that need to be loaded like fonts, dictionaries, timezone data, among others. There are also many different types of code residing in special memory-mapped areas. Dalvik bytecode might be stored in APK or JAR files. It might also be AOT compiled to native code and then stored in some special OAT files [AND21c]. Some of the regularly used objects in the compiled code might support caching through separate special ART files. Internal runtime libraries are packed into custom modules, called APEXes [AND21b], while several shared libraries are loaded for each application as they inherit from a special process named `zygote` [GOO20e]. All these resources ultimately reside in the memory address space of an Android application and might require special handling during capture. Arbitrarily removing their read permissions, as a means of identifying the input to a hot region, can potentially crash the application. On top of that, complex runtime operations, like the garbage Collection, might interfere with the capture mechanism.

The proposed approach is transparent to the users and application developers as it requires no manual instrumentation, controls the frequency of taking captures, and performs space saving optimizations. The remainder of this section explores the required actions that enable partial input capturing in the first place, followed by some optimizations.

6.3.1.1 Required changes for the capture mechanism

Entrypoint for capturing and extensions to the fault-handling mechanism.
The *ArtMethod* structure (see Section 2.4) is extended to support a captured execution mode. When a capture should occur, a special entrypoint overrides the default one in order to initiate it. Otherwise, the code is executed as normal. Captures might still bail out, as explained in the following section. The signal fault manager of ART [GOO20d] is also extended to handle the deliberate segmentation violations caused by the capture mechanism. The remaining faults will normally follow the standard handling chain

until they are eventually served by the appropriate handler.

Specially handling particular main memory areas.
Removing the access rights of some memory pages by read-protecting them might crash an Android application directly, while still being captured, without ever reaching the signal fault manager. For captures to succeed, the protection mechanism conservatively chooses to include the entire memory state of areas that were found to be prone to crashes. Those areas were identified empirically. Some examples include the stack of the fault handler, the data segments of the ART APEX module and any libraries that it is directly linked against[1], as well as read-write areas of some auxiliary structures that are managed by the garbage collector.

Canceling captures on high-impact runtime events.
The main priority of the capture mechanism is to remain unnoticed from actual device users under any circumstances. Therefore, in the case of an imminent garbage collection, any scheduled captures are canceled. This is because GC algorithms, like the copying collector, can cause undesired side-effects by moving the allocated heap objects to different places in memory. This can increase the amount of the page faults that will then have to be handled, increasing both the runtime overheads as well the amount of the stored data in a snapshot.

6.3.1.2 Optimizing for performance and storage

Tuning the capture frequency.
Taking multiple snapshots during the same application run is supported, however, just a single one is sufficient to drive compiler optimization. Since there are plenty of opportunities to take a snapshot, the frequency of doing so is kept to a low value. This can be adjusted both per hot region and per application run. Additionally, the proposed approach allows providing a capture pattern, which can dictate the point in time that the snapshot should occur. This allows taking snapshots at different input states of an interactive application.

Saving space from pages that contain immutable data.

[1] Android Runtime shared library dependencies: cs.android.com/../runtime/Android.bp

A significant portion of the used memory pages is not process specific. Compiled code is mapped into main memory using immutable text segments. It only changes when security updates are installed or when an OS update occurs. Fonts and other framework resources are another example of data that change only during OS updates. These use main memory mappings that are backed by files. Therefore they can be kept outside of the snapshots. The replay process can simply remap those files to the relevant places in memory while restoring the input. On OS updates, the proposed system would have requested a fresh capture anyway, since the underlying runtime behavior would most certainly change.

There are also many objects that reside in main memory areas that only change across different device boots. Those areas are mainly caches of frequently used ART components (i.e., methods, classes, or objects). They are stored only once and are reused among different captures that have happened during the same device boot. This optimization aims to further reduce the storage footprint of the page capture mechanism.

6.3.2 Heap-Object Intersection captures

The *Page* capture mechanism, presented in Section 6.3.1, identifies memory pages that have been read by the hot region and captures them in their entirety. This section presents an improvement over it, able to store significantly less data without causing any additional overhead to the users. It operates by intersecting any reachable heap objects with the input memory pages that the *Pages* approach can identify.

Figure 6.1 visualizes the proposed object intersection mechanism. Similarly with the *Page capture* mechanism, it operates after the hot region has executed, on a child process. As the child process executes with low-priority (see Section 4.2), the intersection mechanism will engage only when there are free computing resources, not to interfere with any other, unrelated user activities. Therefore, it does not cause any noticeable overheads. Initially, the child process creates a set of root objects that can be accessed directly from the hot region. Then, it uses this *root set* to compute the *reachable set* of objects. In essence, this set consists of all the objects that are potentially accessible by the region. As a third step, the child intersects any data belonging to this reachable set with the input pages. Typically, several reachable objects reside outside the input pages (i.e., $O_1, ..., O_n$) while many unreachable ones reside inside the input pages (i.e., $U_1, ..., U_n$). This is why the intersection between the reachable

and input pages is ideal for capture size optimization. Subsequently, any heap-page data that were marked as unreachable by the region are zeroed-out. To reclaim most of the erased space, the child compresses these pages [DEU96], and lastly it stores the reduced input data to permanent storage. Section 7.2 discusses how this approach overestimates the reachable object set, as some paths of objects that reside in input pages might be reached through unreachable pages.

Algorithm 3: Object intersection with input pages.

```
method: hot region's entry point method
inputPages: pages that were identified as the hot region's input
def performIntersection(method, inputPages):
    roots ← getClasses(method)
    foreach param ∈ getParameters (method) do
        if type(param) ∉ Primitive then
            enqueue(roots, param)
        end
    end
    reachableAreas ← ∅
    reachableObjects ← findReachable(roots)
    foreach obj ∈ sortReferences(reachableObjects) do
        if resides(getReference(obj), inputPages) then
            start, end ← objectBoundaries(obj)
            markReachableArea(reachableAreas, start, end)
        end
    end
    zeroOutUnreachableArea(inputPages, reachableAreas)
    compress(inputPages)
def findReachable(workQueue):
    reachable ← ∅
    while workQueue ≠ ∅ do
        obj ← dequeue(workQueue)
        insert(reachable, obj)
        foreach field ∈ getFields (obj) do
            if type(field) ∉ Primitive and field ∉ reachable then
                enqueue(workQueue, field)
            end
        end
    end
    return reachable
```

Algorithm 3 outlines in greater detail the intersection mechanism. Initially, the hot region's *root set* is calculated. This includes all the classes as well the parameters of the hot region's entry point method. Then, each object in the *root set* is visited using a breadth first search traversal, as the Android code hints at better locality. During visit, objects are added to a *reachable set* and their references are scheduled (in *workQueue*) to be visited afterwards. Once all reachable objects are found, they are sorted based on their memory addresses before the proposed mechanism iterates over them. For each

object, it is checked whether it *resides* inside the *input pages* and if so the approach calculates its memory footprint (the beginning and the ending offsets) and marks this area as potentially reachable by the region. This allows calculating the unreachable area so it can then be zeroed out. Once it is done, those trimmed pages are compressed, which now only contain region reachable data, leading to noticeable space savings.

6.3.3 Capture Everything approach

This section introduces another input capture mechanism for interactive applications, named *Everything*. This approach favors a simpler runtime operation over capture sizes. As the name suggests, this approach stores all data present in memory and it was used as a comparison in evaluation.

The main benefit of this approach is a simpler implementation and operation, which leads to less runtime overheads. Similarly with page captures, it uses a `fork` call so the execution does not have to freeze until all of the original input is stored. This is the only runtime overhead introduced, as it does not need to analyze and memory-protect the entire address space, or subsequently handle deliberate segmentation violations that the previous two mechanisms rely upon. Then it stores all of the data that the program might use. These data are found by parsing the `proc/maps` kernel structure [LIN20]. As it will be demonstrated later on, this significantly decreases the online operation overheads, at the cost, however, of increased storage size.

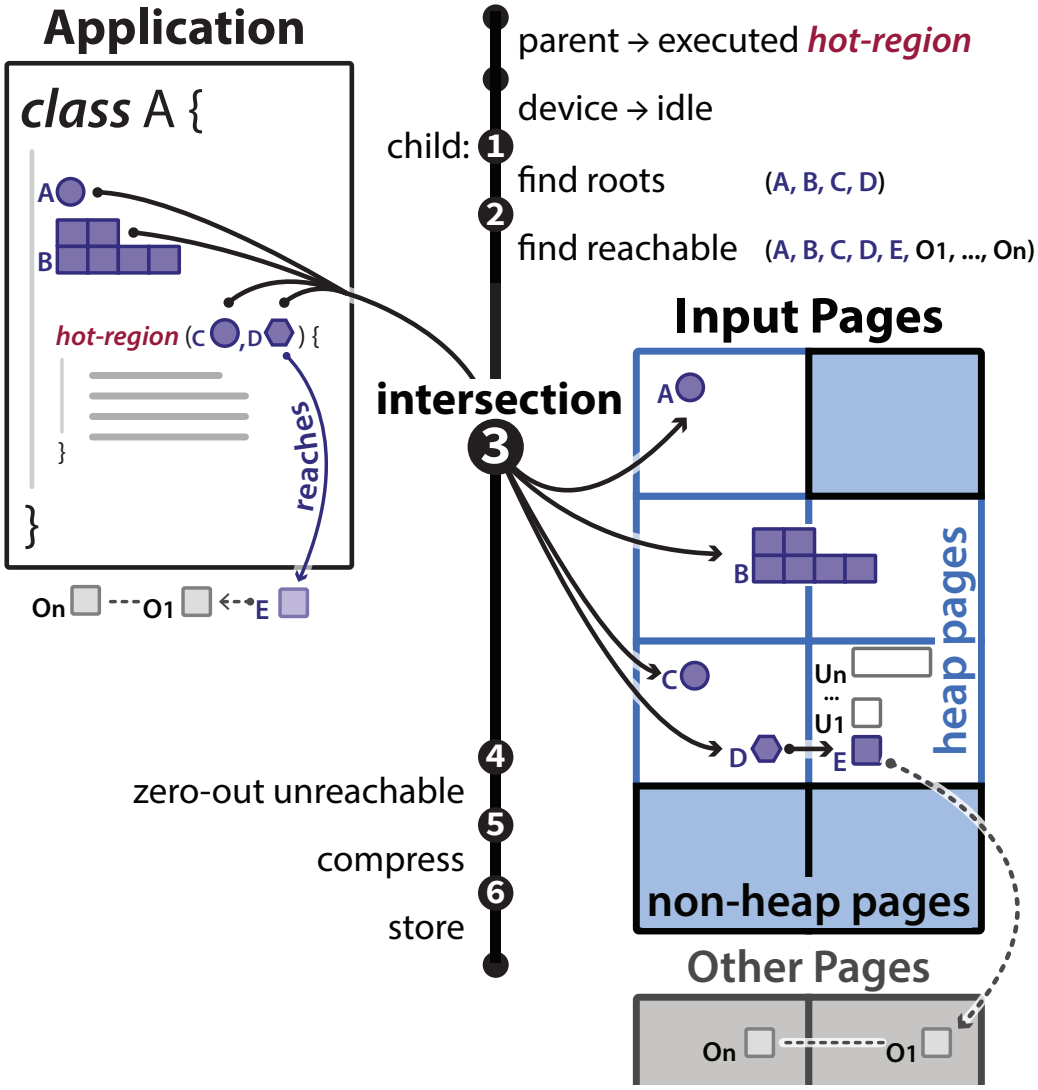

Figure 6.1: Reducing capture sizes by intersecting reachable objects with memory pages containing the input. By visiting the hot region's parameters and fields, the reachable set can be constructed. Most reachable objects typically fall outside of the input pages ($O_1, ..., O_n$), while several unreachable objects the opposite ($U_1, ..., U_n$). By identifying the reachable ones inside the *Input Pages* the space shown as a white area can be reclaimed.

6.4 Replaying Android code regions

With all of the state used by the hot region captured, the next step is to use it to recreate offline the behavior an Android application exhibited online. At its most basic, replaying a previously captured execution is just a matter of reloading the saved state of the application and jumping into the hot region's entry point [VIR20]. With the architectural registers and all used memory having the same values as when the hot region was originally executed, the execution should flow the exact same way. The globally visible results of the replayed execution should also be the same.

In Chapter 4, the replay mechanism that was presented had some limitations. First, it was restricted to a single function of a C program that had no external library dependencies. Each time a replay was needed, the restore procedure was initiated through manual instrumentation. As both the input application and the restore mechanism were based on the C runtime, replaying was relatively straightforward. For Android applications though, a different and more involved procedure is required. Once the input is restored, the mechanism must allow the execution of the hot region using different code types like interpretation, the original Android-compiled code, and multiple code versions generated through iterative compilation. Additionally, it must operate well in the presence of memory-shuffling security mechanisms that are an essential part of any modern operating system, including Android. The remainder of this section describes in detail the implementation of the replay mechanism for Android applications.

Replays are initiated through a *loader* program that gradually transforms itself into a partial Android process. Only the pages read by the region are restored, which conserves storage space and reduces the time needed to set up each replay. Once all captured pages are put into main memory, any residues of the *loader* program are removed. The mechanism works well in the presence of memory-shuffling security mechanisms by dynamically handling any page collisions that might arise. Once the processor state is restored, it chooses the type of code to carry out the execution of the hot region. This whole *replay* operation is shown in Figure 6.2 and it is composed of the following steps:

❶ Load the captured state:
A program named *loader*, written in `C`, undertakes the task of restoring all of the captured state into main memory. Most of the memory pages are placed directly into the virtual addresses they originally had while in the capture process. Some pages may

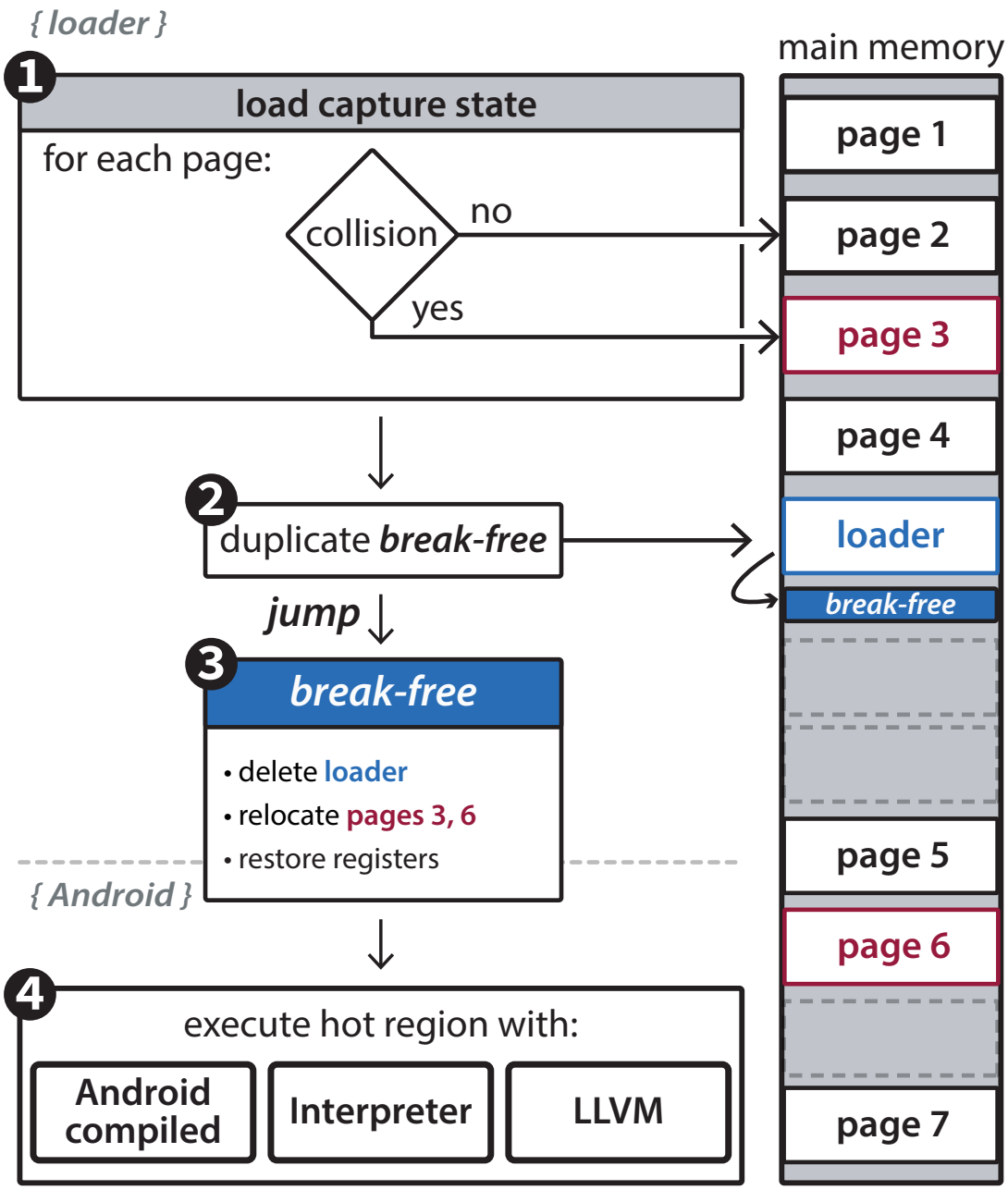

Figure 6.2: The replay mechanism starts with a *C loader* program that gradually transforms itself into a partial Android application. The resulting process has an identical input state with a previously captured process from a hot region's standpoint. Once everything is restored, the loader can choose different types and versions of code to carry out the execution of the *hot region*. The proposed mechanism works well alongside `ASLR`, a memory-shuffling security mechanism that is present on all modern operating systems.

map to memory locations already occupied by the loader's pages because of Address Space Layout Randomization (ASLR), a security mechanism that randomly shuffles processes' memory. To work regardless of ASLR, the proposed mechanism does not make any assumptions about the address layout of the loader. When captured pages collide with its context, they are placed into temporary memory locations. Those locations are chosen so that they do not introduce further collisions. This means consulting the captured memory layout and choosing locations not used by it. The captured architectural state of the processor is also read and stored into a temporary location.

❷ Duplicating the `break-free` method:
To resolve collisions caused by the loader, all of its state must first be discarded before any captured pages can be relocated to their original position. Simply put, the loader program must both delete itself and then somehow keep setting up the replay process. Unfortunately, this cannot be done directly. Therefore, during this step the binary code of a special position-independent function is duplicated, named `break-free`, to a non-colliding area. The `break-free` function belongs to the loader itself and it is copied by simply duplicating the relevant bytes from its *text-segment*.

❸ Becoming a partial Android process:
Subsequently, the execution flow jumps to the duplicated `break-free` function, which becomes self-contained by switching to its own stack and data segments. At this point, the transition from a C process to a partial Android one can be completed in three steps. First, the original loader pages are released. Then, the `break-free` method consults some structures found in its data segment to apply relocations of any colliding pages. Those pages are now placed to the memory locations they had while in the original capture process. Finally, the `break-free` method completes the transition to a partial Android process by restoring the architectural state of the processor. Once the registers are restored, including the PC, the execution flow transfers from the `break-free` method to the Android runtime. If there needs to be a code-mode switch (i.e., from `arm` to `thumb` mode), then it is performed at the same time the PC is set.

❹ Choosing code type to execute the hot region:
The final step of the replay mechanism is to choose the code type that will carry out

the execution of the *hot region*. Three different code types are supported. The first is replaying the original Android compiled code. This is used as the evaluation baseline in the experiments presented in Section 6.9. The second is calling the Android interpreter. This execution type is used to extract information from captures to enable correctness verification and further optimization, as it is described in Section 6.4.2. The final code type is calling a new optimized binary. In this case the binary is also loaded into memory before the replay process jumps to its entry point. This binary is generated using the Android LLVM backend, described in Chapter 5.

6.4.1 Automatic code correctness verification

During iterative compilation the compiler is quite often presented with transformation sequences with which it has not been tested. This may hang or crash the compiler, or yield a program that crashes or hangs at runtime, or successfully completes but produces the wrong output. In all these cases the compilation flags must be rejected and the optimization search must move on to the next point. All but the last case are easily checked. Catching the wrong output is critical as it might lead to silent data corruption. It requires verifying the hot region's observable behavior, which can be a non-trivial problem especially when it needs to be done automatically and at scale.

The system for C programs, presented in Chapter 4, performs a manual verification step. During that step, hard-coded tests are manually injected after the call-site of the hot region to validate the correctness of the return value. In a real scenario, this would require a significant effort from the application developers. Another severe limitation is that verifying only the return value is insufficient. This is because the externally observable behavior of the hot region might very well include modifications to instance or static fields.

With a special interpreted replay of the hot region a *verification map* is built, as visualized in Figure 6.3. To populate this map, the interpreter stores key-value pairs of memory locations and values for every object field or array element that was modified. Only the final values are recorded. Any object or array creation is also stored. Finally, the interpreter records the hot region's return value.

Together, these data represent the externally observable behavior of the region. After each iterative compilation replay, the execution correctness is verified by comparing the memory of the process against the verification map. This does not require any online instrumentation or manual effort by the application developers. This interpretation

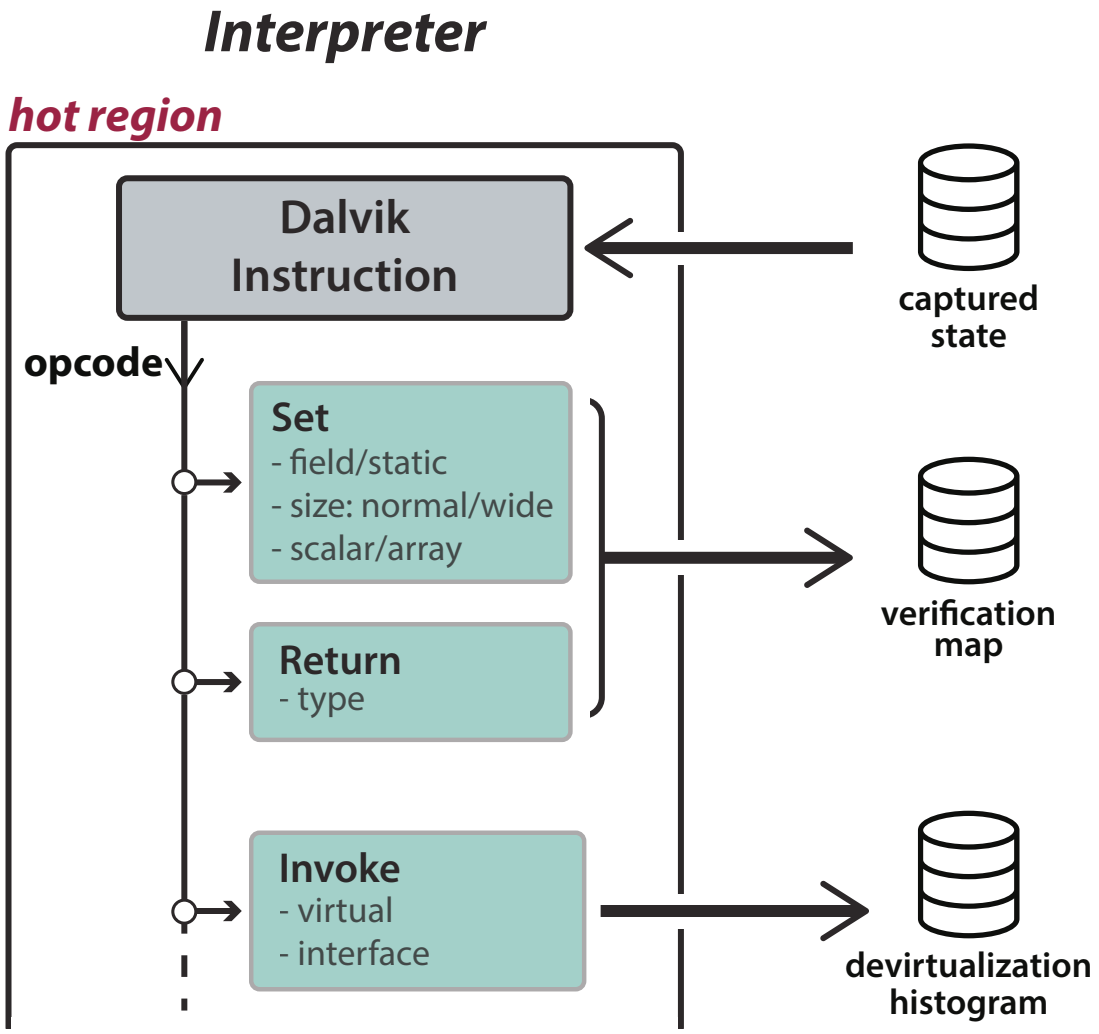

Figure 6.3: With a special interpreted replay of a hot region, any instructions that relate to variable writes, method return values, and method invokes, are intercepted. This allows the generation of a *verification map* that can be used after each replayed execution of a hot region to verify the correctness of the used code. It is also used to dynamically extract information for code optimization. Specifically, it can enable aggressive inlining, speed up virtual and interface calls with speculative devirtualization, and tune branch prediction.

step is quite slow, but as with iterative compilation it happens when the mobile device is not otherwise being used, therefore it does not affect the user experience.

6.4.2 Exploiting capture data for code optimization

```
define void @InvokeSpeculative(i8* %obj) {
_entry:
  %obj_class = ; .. %obj->class_offset
  switch i32 %obj_class, label %_miss [
    i32 100, label %_class1
    i32 101, label %_class2
  ], !prof !1
_class1:
  call void @Cat.walk(i8* %obj, ..)
  ret void
_class2:
  ret void
  call void @Dog.walk(i8* %obj, ..)
_miss:  ; Do a slower runtime call:
  %thread = call i8* @LoadThread()
  ; Resolve the method
  %AnimalWalk = call i8* %entrypointResolveVirtual(obj, ..)
  ; allocate & setup arguments
  %args = alloca [3 x i32], i32 3, align 1
  %result= alloca %union.jvalue.126
  ; .. bitcode to setup ART stack frame
  %7 = call i8* %entrypointLlvmPushArtFrame(..)
  call void %AnimalWalk(obj, thread, args, result, ..)
  ; .. bitcode to remove ART stack frame
  call void %entrypointLlvmPopArtFrame(..)
  ret void
}

!1 = !{!"branch_weights", i32 30, i32 10, i32 0}
```

Listing 6.1: With speculative devirtualization, the overheads of virtual or interface call sites can be optimized. The class of the object is compared to a previously generated histogram to directly invoke a method without ever leaving LLVM. The histogram is consulted at compilation time to aggressively inline on the speculated methods. The `switch` branch prediction weights are also tuned, according to the frequency histogram.

An added benefit of being able to replay a captured execution offline is that additional information can be collected regarding the execution of an application, which

would have been too costly to perform online. This section describes how this information is extracted through interpretation and how it is used to further optimize the code.

The interpreted replay, presented in the previous section, is extended to collect additional profiling data. Such data are used to improve the quality of the code that is generated by the LLVM backend, as shown in Listing 6.1. In particular, at each virtual or interface call-sites, the frequency histogram and the actual dispatch types are recorded. This additional data reflects information from real user inputs and can be used to reduce the call overheads. With speculative devirtualization of call sites, aggressive inlining can be performed, which is quite beneficial in object-oriented programming languages [SUN+00]. Additionally, branch prediction is tuned on each speculated type according to the frequency histogram.

6.5 Using a genetic algorithm for optimization search

LLVM has a very large optimization space with almost 200 passes that can be applied multiple times with a different effect each time that depends on previously as well subsequently applied passes [COO+02a]. On top of that, it has more than 1300 optimization parameters and flags. Applying a pass or changing a flag might improve performance, but it may also degrade it, produce a faulty binary, or have no effect at all. What is needed is a quick way to explore the rewarding areas of such a complex code transformation space.

This section describes an optimization search based on a Genetic Algorithm that extracts better code transformation sequences to improve an application in terms of speed and/or code size. It is a well established approach that has worked really well in the past for similar problems [COO+99; FAT+04; KUL+12; LIN+08].

Genomes encode the sequence of passes, the parameters, and the flags. They may vary in length to account for different numbers of optimization passes. There are three different mate selection pipelines. The first chooses mates from the elite genomes only, the second chooses the fittest individual, and the third one uses a tournament selection. Once mates are selected, they are crossed over with a single random point. It is ensured that the resulting genome will be higher in length than a configurable minimum. There are several mutation operators for different types of genes:

- If a pass is a *boolean*, then it might be enabled or disabled.
- If a pass accepts a parameter, then it might be modified according to its type.

- New passes might be introduced.

The amount of mutations is configurable. The genetic algorithm begins with a fixed population size and progresses until either a threshold number of generations is reached or a number of generations has elapsed without any improvement over the best performing genome. At the end, a hill-climbing step is performed, which ensures that the local maximum is reached.

The fitness function focuses primarily on performance, which is measured by replaying an application's hot region, as described in Section 6.4. However, if the performance of two different code versions is sufficiently close, then the binary version with the smaller size is preferred. The full parameters of the genetic algorithm are provided in Section 6.8.

6.6 System for real Android applications

The goal of this thesis is to optimize applications in the mobile environment. It is done by comparing the effect of different optimization decisions, without ever negatively affecting the user experience. For sound comparisons, the application needs to perform the same amount of work each time an optimization is being evaluated, and that work should also be representative of actual usage. Since performance evaluations can affect the user experience, they should be performed only offline, when the device is idle and charged. Any solution must also handle the inherently noisy mobile environment. Additionally, the optimization search must be specialized per mobile device, application, code version, among other factors.

This section describes a system that realizes an input-driven iterative compilation search, able to optimize real Android applications without imposing unbearable overheads to the users. A client-server model is used for orchestrating the optimization search. Clients are mobile devices that request code transformation sequences and upload their findings to a server. The server then accumulates information and orchestrates the whole iterative compilation process. Once finished, the findings are reported back to the mobile device. A high-level overview of the proposed approach is shown on Figure 6.4 and is described in more detail in the remainder of this section.

During an initial profiling phase, a hot region is automatically identified. Then, real user inputs are transparently captured for it. It is followed by an input-driven offline iterative compilation that is performed as follows. A GA probes the complex transformation space of a compiler to construct optimization sequences. Those are

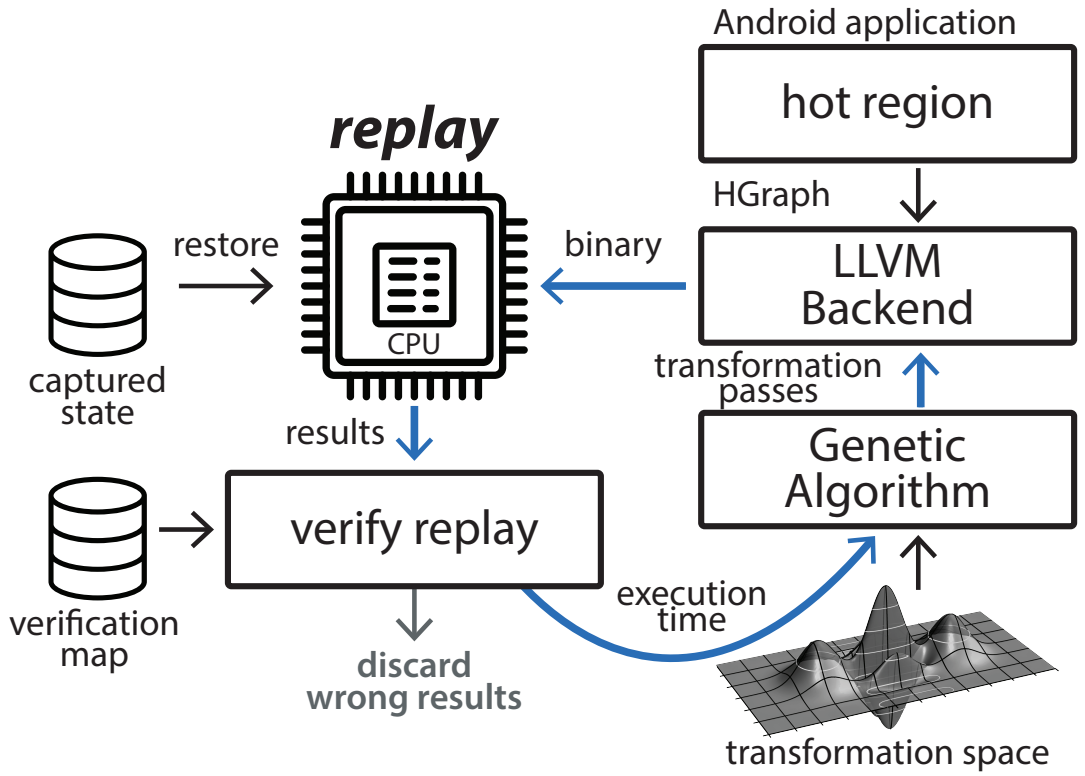

Figure 6.4: A high level overview of a system that operates on real Android applications. It uses offline iterative compilation that replays real user inputs as a means of evaluating different code transformations. It repeatedly compiles and replays a hot region with transformations extracted from the LLVM's space, found using a GA. On each replay, it restores previously captured input states. It performs multiple replays per evaluation and reports the execution times back to the GA. Any transformation that produces a wrong output is discarded. This whole process is repeated until the GA converges.

then fed to the LLVM `opt` and `llc` tools to transform the hot region's bitcode and compile to machine code respectively. The captured execution is then replayed using the generated binaries, in order to evaluate their performance. The GA then advances to the next generation and continues the process. Each replay has the same input since the same captured state is restored each time. The soundness of the comparisons is ensured by performing multiple replays per binary and using statistical methodologies, as described in Section 5.4. To reduce the random performance variation, the execution environment of Chapter 4 is used (see Section 4.4).

Using a previously generated verification map, any transformations that lead to a wrong externally observable behavior of the hot region are discarded automatically, without requiring any developer efforts.

A client-server model is used to orchestrate the optimization search. The mobile devices act as clients. They simply communicate with a server machine, which instructs them on how to proceed. To allow multiple users to optimize the same code region, a hash method is used over a string that contains all of the region methods as well the application identifier and version. Their iterative compilation findings, which include execution time and compilation size, are uploaded to the server. They are classified per device architecture, device model, compiler and OS version, application and its version, and the device's user. The server uses all this information to guide the search. It might generate a new code transformation sequence, using genetic search, and instruct a mobile device to evaluate it. It might also request a sequence to be minimized, which is done by removing all transformations that have no effect on the resulting binary. Finally, once the search for a particular input has finished (including the hill-climbing step) it reports back to the device the best performing code transformations.

6.7 Using crowd-sourcing to accelerate the genetic search

By pushing iterative compilation offline, through replay-based evaluations, the proposed approach can hide from the device users the significant overheads associated with the technique. These include compilations and repeated evaluations, necessary for statistically meaningful comparisons, several of which have a detrimental effect on performance. This section describes a scalable, crowd-sourcing architecture to accelerate this offline effort. It performs a collaborative search between several users. As the evaluation data will now be jointly produced, both a greater search can be performed and new users can benefit from the findings faster.

Figure 6.5 illustrates how a crowd-sourced collaborative search is performed. Once several users have had their inputs captured, they can participate in a joint search, as coordinated by the server, as long as their devices remain idle and charged. At no time the users send their snapshots to the server. Initially, a device requests the next genome to evaluate. The server then queries the database for existing evaluation data, as well the code transformation space of a compiler (i.e., LLVM) to generate the next evaluation point. As the previous genomes are utilized for the construction of the new ones, freshly joined users discover better results in less time. The device then proceeds by compiling and evaluating the genome, utilizing the replay-based evaluation system presented by the previous section. Finally, the server updates the database with the

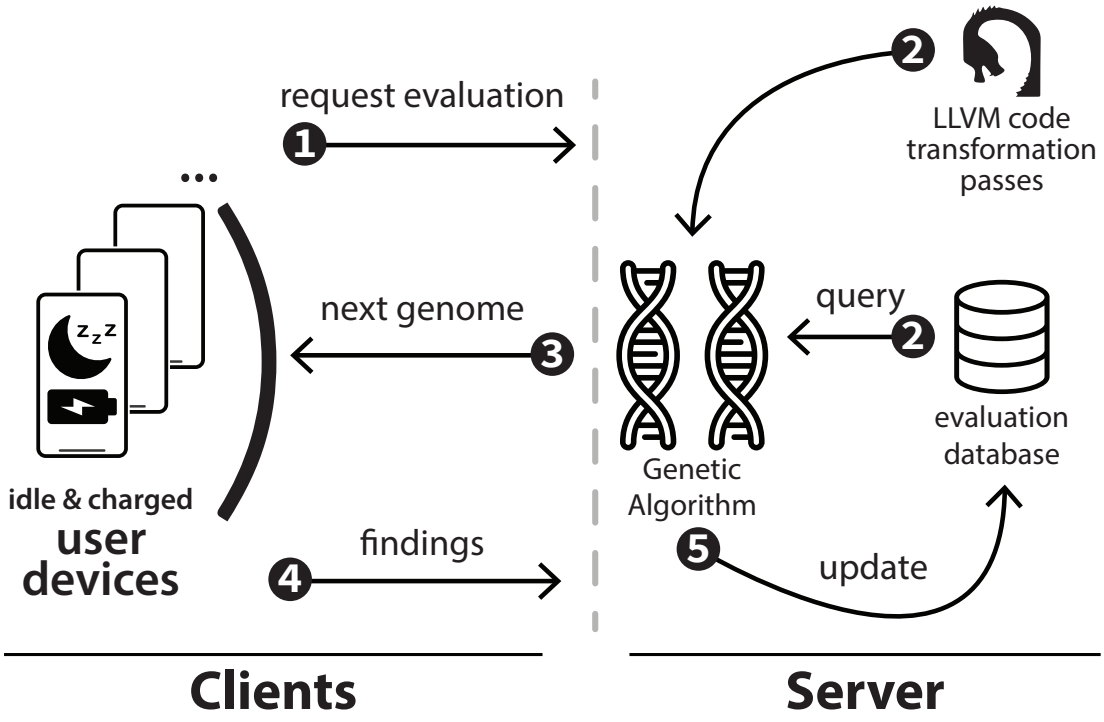

Figure 6.5: Accelerating the GA search with crowd-sourcing. Several idle and charged devices collaborate for spreading the offline evaluation efforts. Participating client devices communicate with a server which instructs them on what genome to evaluate next, based on the code transformation space and the previous evaluation data. The device then evaluates the next point in space and reports the findings to the server, which finally updates its database.

findings reported by the device. These include compilation time and size, any runtime or compilation errors, and of course the evaluation timings. All this data is used to tune the search for the best genome of a hot region among several users. This approach, as it will be shown in results, is capable of finding near optimal genomes for most cases, when compared to individual searches, for a fraction of the total evaluation time.

6.8 Experimental Setup

A system was implemented to evaluate the proposed approach. It uses the same mobile device and Android applications described in the Section 5.4. It also uses the same execution environment and statistical methodologies described in Section 4.4. The remainder of this section provides the additional experimental setup that is specific to this chapter.

An Android application orchestrates the operations of the proposed optimization framework. It provides interfaces to the following functionality:

- Run the profiler on a new application
- Schedule and manage input captures
- Perform iterative compilation on captured inputs
- Use the best findings of iterative compilation

The application invokes the LLVM backend that is introduced in Chapter 5 to compile Android application code to LLVM bitcode. It communicates with an Apache web server through a RESTful API to receive instructions on how to perform the optimization search. The server employs a custom GA implementation that utilizes a document store for managing its data. The GA searches through the enormous space of the LLVM compiler. It uses 11 generations, the first randomly generated, while the other 10 are driven by genetic search. Each generation consists of 50 genomes. While in the first generation, the GA attempts up to three times to replace each genome that leads to worse performance than both LLVM and Android baselines. This biases the algorithm towards the more profitable areas of the transformation space. Once the first generation is complete, the algorithm instructs the mobile device to minimize any genomes before those participate in a crossover. This is done by removing all the redundant passes. A genome has a 5% probability of undergoing any mutation. If that is the case, each of its genes mutates with a 5% probability. In some cases, the GA might produce a binary that was already evaluated. This is called a *duplicate binary* and there

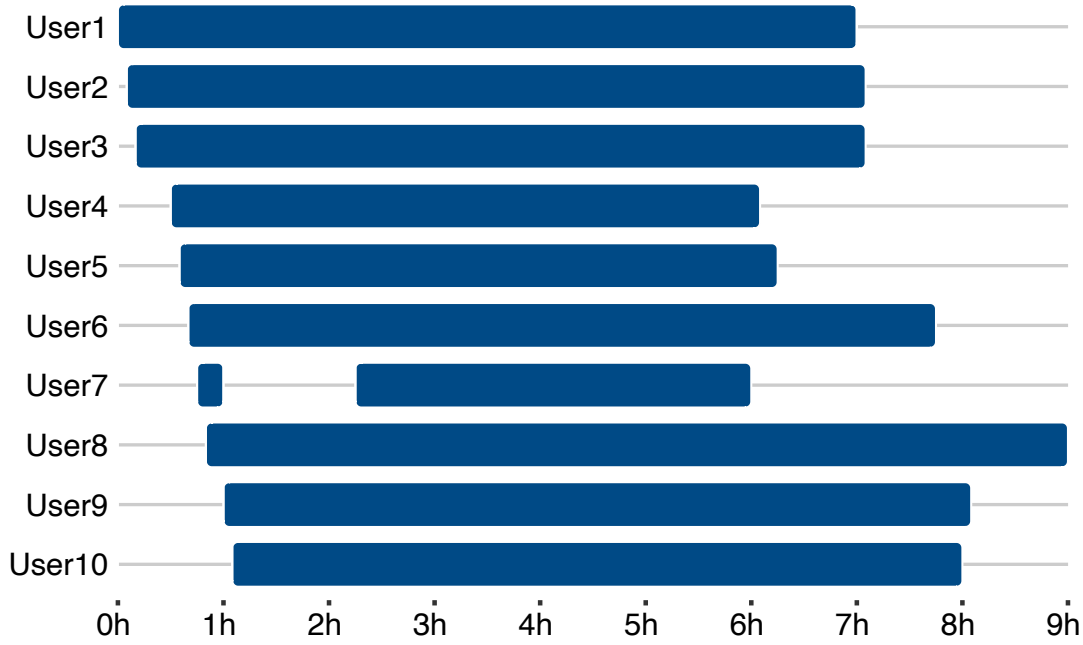

Figure 6.6: User availability for a joint optimization search.

is no point in evaluating it again. There might be edge cases where the algorithm (from one point onward) produces only duplicate binaries. To ensure it always halts we set the maximum number of allowed duplicate binaries to 100. Each tournament selection round considers seven candidates with a 90% probability.

The final reported speedups are generated by collecting the execution times of a hot region outside the replay environment.This guarantees that the findings are not an artifact of the replay environment. The best performing binary discovered by the GA earlier is used, however, the application is executed interactively instead of being replayed.

For the crowd-sourced GA search, a fixed initial population of 200 was used to ensure that the same amount of points in space are visited on each run. It was evaluated using ten *Google Pixel 4* devices. Each device represented a different user and each user had a different input. For the interactive applications, new inputs were generated using manual interactions. For the benchmark applications, the hot region's problem sizes were altered. Once all user inputs for each app were captured, the mobile devices were programmed to participate collaboratively, in a realistic way. Studies have shown that most users charge their phones overnight [FER+11], when devices are also conveniently idle. Using sleep data [CDC21], a user availability schedule was created, as visualized by Figure 6.6 . Users joined and exited the search at different times and had

different availability. *User7* had the least participation, as the device initially joined for a brief period, followed by a break, before joining again for a few more hours.

6.9 Experimental Results

The goal of the system presented in this chapter is to optimize real mobile applications using iterative compilation without incurring any noticeable overheads to the device users. The proposed system is evaluated with six sets of experiments that are described in the remainder of this section.

The first set presents the achieved performance gains, using Android applications from Table 5.3 to showcase the potential of the proposed approach. The second illustrates why an intelligent search over the optimization space is needed for automatically tuning the heuristics of a compiler. It also demonstrates how online approaches, regardless of whether being self-adapting, potentially introduce unbearable overheads while searching for better code transformations. Finally, it presents an analysis of the optimization passes that were found to be best. The third compares the online overheads of the proposed input capture mechanisms and the fourth one compares their storage requirements. The *Page* capture mechanism is used only for comparison. The *Intersection* mechanism improves upon the *Pages* approach as it further minimizes the capture sizes, making it applicable even on low-end devices. The capture *Everything* approach focuses exclusively on minimizing the runtime overheads. It is demonstrated that all approaches have transparent online operations from the users. The final set showcases the offline GA search acceleration when crowd-sourcing the findings from several users.

6.9.1 Speeding-up Android applications

The proposed approach readily outperforms the Android compiler with aggressive optimization through offline, replay-based iterative compilation. This section presents the performance gains achieved for 21 Android applications. The code breakdown that is presented by Figure 5.2 is applicable to this experiment as well.

Figure 6.7 shows the speedups achieved for 12 Android benchmark applications. The binaries are selected by the GA and are evaluated outside the replay environment. They are compared against two baselines. The first is the default Android compiler, against which all other speedups are measured, and it represents the performance users

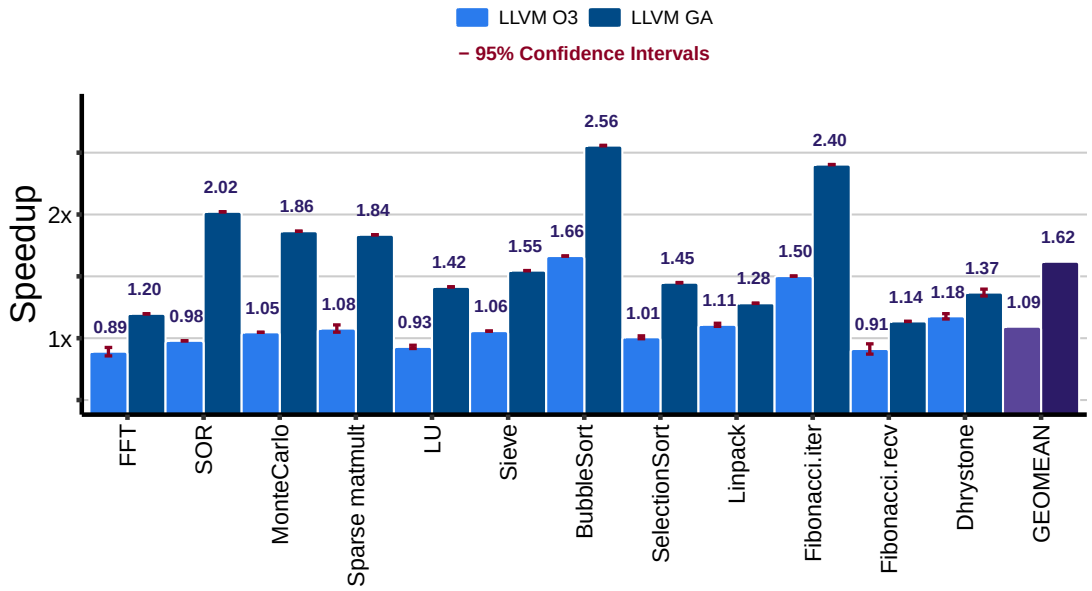

Figure 6.7: Speedup relative to the Android compiler for 12 benchmark Android applications. LLVM `-O3` performance ranges from a 0.89x slowdown to a 1.66x speedup. LLVM GA, which uses a replay-based iterative compilation is able to significantly improve execution times. It produces speedups from 1.14x to 2.56x and an average of 62%.

get out of the box. The second one, LLVM `-O3`, is an aggressive optimization setting for the LLVM backend, presented in Chapter 5. It represents the best LLVM can do without any application-specific information.

The proposed optimization framework is shown as LLVM GA. It builds on top of LLVM `-O3`, which is described in the previous chapter. It exploits capture data to aggressively inline and tune bitcode, which makes any subsequent optimization passes more effective. Then, the offline GA search generates progressively better code despite performing a relatively quick search. It operates with 11 generations at most, each one having 50 genomes. Despite visiting a tiny fraction of the optimization space, the system is able to improve performance over both baselines for all benchmarks, achieving a noticeable speedup average of 62%. The speedups range from 1.14x for *Fibonacci.recv* to 2.56x for *Bubblesort*, which was the highest overall when considering both benchmarks and interactive applications.

Figure 6.8 shows the speedups for interactive Android applications. When considering the hot region in isolation (see Figure 6.9) similar findings were observed. When considering wider regions that surround the code that was optimized, the speedups

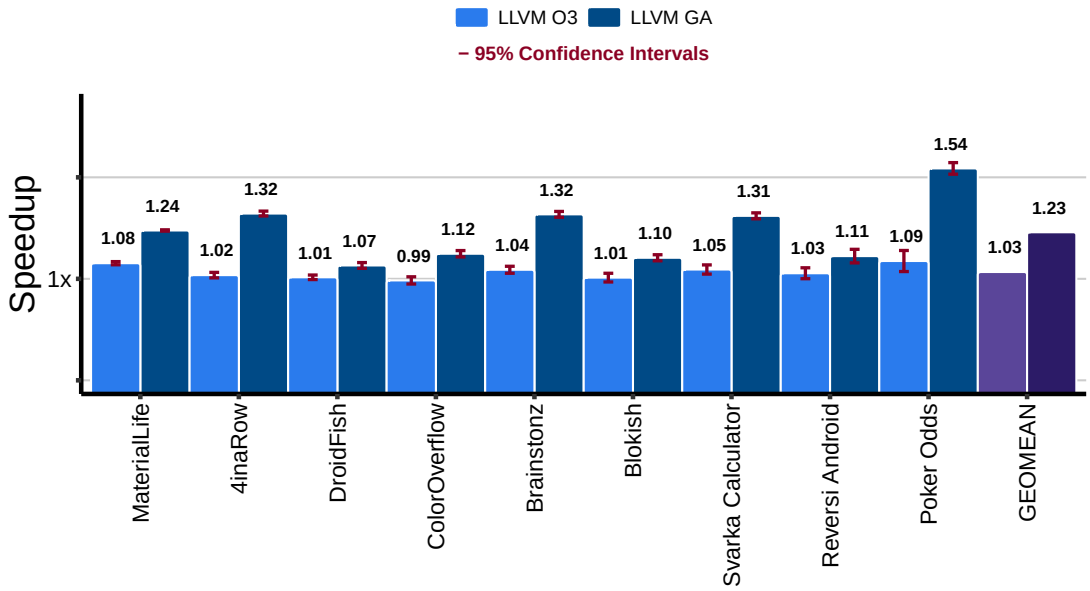

Figure 6.8: Speedup relative to the Android compiler for 9 interactive Android applications. The reported results consider code that surrounds the hot region to show the improvement over the whole application runtime. LLVM `-O3` performance ranges from a 0.99x to a 1.09x speedup. LLVM GA uses an offline, replay-based iterative compilation. It is able to improve performance further, with speedups ranging from 1.07x to 1.54x and an average speedup of 23%.

range from 7% to 54%. As described in Figure 5.2, the amount of code that benefits from iterative compilation ranges only between 14% and 81% of the total execution runtime.

When considering both interactive and benchmark Android applications, the average speedup is 44%. These findings underestimate the potential of the proposed approach, as it is held back by the current limitations of the LLVM backend, described in Section 5.3.4. Nevertheless, it still manages to improve performance significantly, including for applications like DroidFish where only a small fraction of the execution runtime can benefit from optimization. With a more mature compiler toolchain it is expected that these gains will grow even further. Chapter 7 discusses further improvements over the proposed system.

6.9.2 Using a GA for offline optimization search

With a GA search over the huge space of LLVM optimization decisions, the proposed approach is able to discover code transformations that significantly outperform the

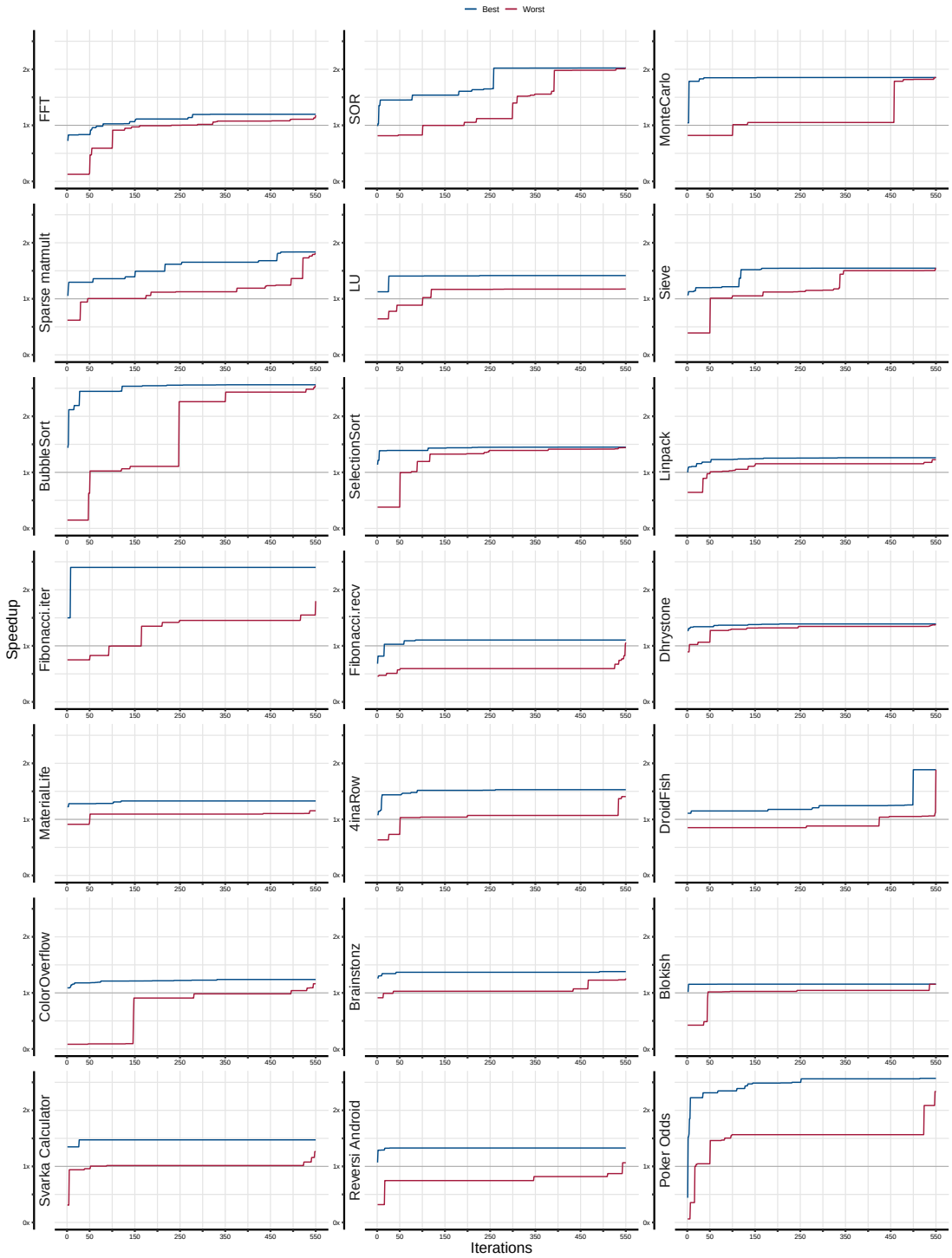

Figure 6.9: Range of speedups over the Android compiler for the LLVM transformation sequences selected by the GA. Speedups are estimated through replay for the *hot regions* only. The two lines represent the evolution of the best and worst genomes over time. Vertical grid lines indicate the change from one generation to the next. All applications benefit from the code optimization search. Application versions worse than the baseline are common across all applications, in some cases even after multiple generations.

Android compiler. This is shown by visualizing how the best genomes evolve over time. It is also demonstrated that GA, like any other self-adapting algorithm, will inevitably attempt to evaluate sub-optimal code transformations during search. This demonstrates that the optimization approaches that rely on online evaluations are not applicable in the mobile environment. The genome evolution is followed by a detailed gene analysis, globally and per application, to give some insight into the most relevant transformation passes.

6.9.2.1 Genome evolution over several generations

Figure 6.9 shows the evolution of the offline GA search in terms of best and worst genomes over time. In all cases the best binary outperforms the Android baseline. For almost all applications the genetic search improves performance over time, except for *Blokish* where the initial random search discovers the best performing binary. More than two thirds of the tested applications require multiple generations to discover their best binary, while a few might have benefited by an even longer search. In detail, six applications keep getting benefits from search until the very end of the process, ten stabilized within four to six generations, while the remaining five applications reach their optimal genome in three or less generations. Overall, the ability to search the transformation space and evaluate optimization decisions in a robust way provides clear benefits.

On the other end, several genomes have an extremely detrimental effect on performance with as much as 10x slowdown. If those were to be evaluated online, they would have had a dramatic effect on the user experience. For the whole first random generation of *FFT* and a significant portion of the first generation of *Fibonacci.recv*, even the best genomes are worse than the Android baseline. This is not limited to the early stages of the search. Four applications were still picking sub-optimal transformations even after more than seven generations, and five applications for three or more generations. The remaining nine had to enter their second generation to stop evaluating sub-optimal code. This does not take into consideration the even slower genomes that were evaluated but discarded during the construction of the first random generation, as explained in Section 6.8. In any case, only a handful of sub-optimal evaluations would have been enough to degrade the user experience and render any online approaches impractical. And still, online approaches have to deal with binaries that might crash at runtime or successfully execute but silently produce a wrong output.

6.9.2.2 Analyzing the best genomes of applications

Figure 6.10 shows the flags along with their usage counts of the best genomes across all applications. They are classified into 11 categories based on the code transformation that they perform. Each category is emphasized and includes a total sum of the used passes. The genomes were minimized beforehand to exclude any flags that do not have an impact on the generated binaries. The impact might be direct, or indirect for the case of the analysis passes.

Architecture, contains any device architecture or CPU specific passes, for *aarch64* and the *Kryo* processor respectively. *Arithmetic*, contains passes that perform mathematical operations. The passes that modify the control flow graph are grouped into *cfg*. *Improve*, contains the passes that replace or lower particular instructions with faster equivalent ones. *Loop*, has all the loop-related optimization passes, except vectorization. *Invoke*, contains the passes that optimize method calls. *Memory*, has all the passes that perform memory and pointer arithmetic optimizations. *Reduce*, are passes that decrease the number of the generated instructions. *Register*, are transformations that relate to register usage. *Vector*, are passes that relate to enabling or disabling loop vectorization. Finally, *other*, contains the remainder of the passes, which might relate to code analysis, altering binary symbols or sections, link-time optimizations, among others.

Loop optimization was the most popular flag category. Passes that relate to loop-unrolling were the most widely used in this category. This is an additional indicator, in addition to the results of Subsection 5.5.3, of the effectiveness of the *post-unroll* optimization presented in the previous chapter. The next most widely used category was *reduce*, followed by *cfg*. *Invoke* and *memory* optimizations come next, followed by the rest of the categories. CPU-specific optimization, parameterized using `--mattr`, was the most widely used flag overall. Those passes included address, literal, arithmetic, or logical fusing. Method inlining comes second, in par with all the unrolling passes. It is assisted by the *speculative devirtualization* optimization that was introduced in Subsection 6.4.2.

Figure 6.11 visualizes the flag category usage per application. It is overlaid on top of the average flag category usage, which is shown with a light gray color. It allows understanding which optimization categories have the most or least relevance to the underlying code. The best genomes for all applications, except the obvious case of *Fibonacci.recv*, include loop optimizations. Around half of the applications have used

pealing or unrolling, while others include sinking, rotation, extraction, unroll-and-jam, unswitching, and guard-widening. *BubbleSort*, *MonteCarlo*, and *ColorOverflow* had a higher number of loop-related optimizations, while *Sieve* had the highest with a total of 12 passes. More than half of the applications have used full or partial inlining, with or without custom thresholds, while *FFT*, *SelectionSort*, *ColorOverflow* and *PokerOds* used the most *invoke* transformation passes. Around a quarter of the applications have used passes that optimize the instruction scheduling, with *DroidFish* having the most *improve* passes. *MaterialLife* have used extensively *cfg* passes. *ColorOverflow* have used 12 instruction *reduce* passes and its genome contained the most passes overall. *Arithmetic* passes were mostly used by *SOR* and *DroidFish*. Other commonly used optimizations include global value numbering, sub-expression elimination, pointer arithmetic optimizations, floating-point improvements, load/store vectorization, and instruction/function/return merging.

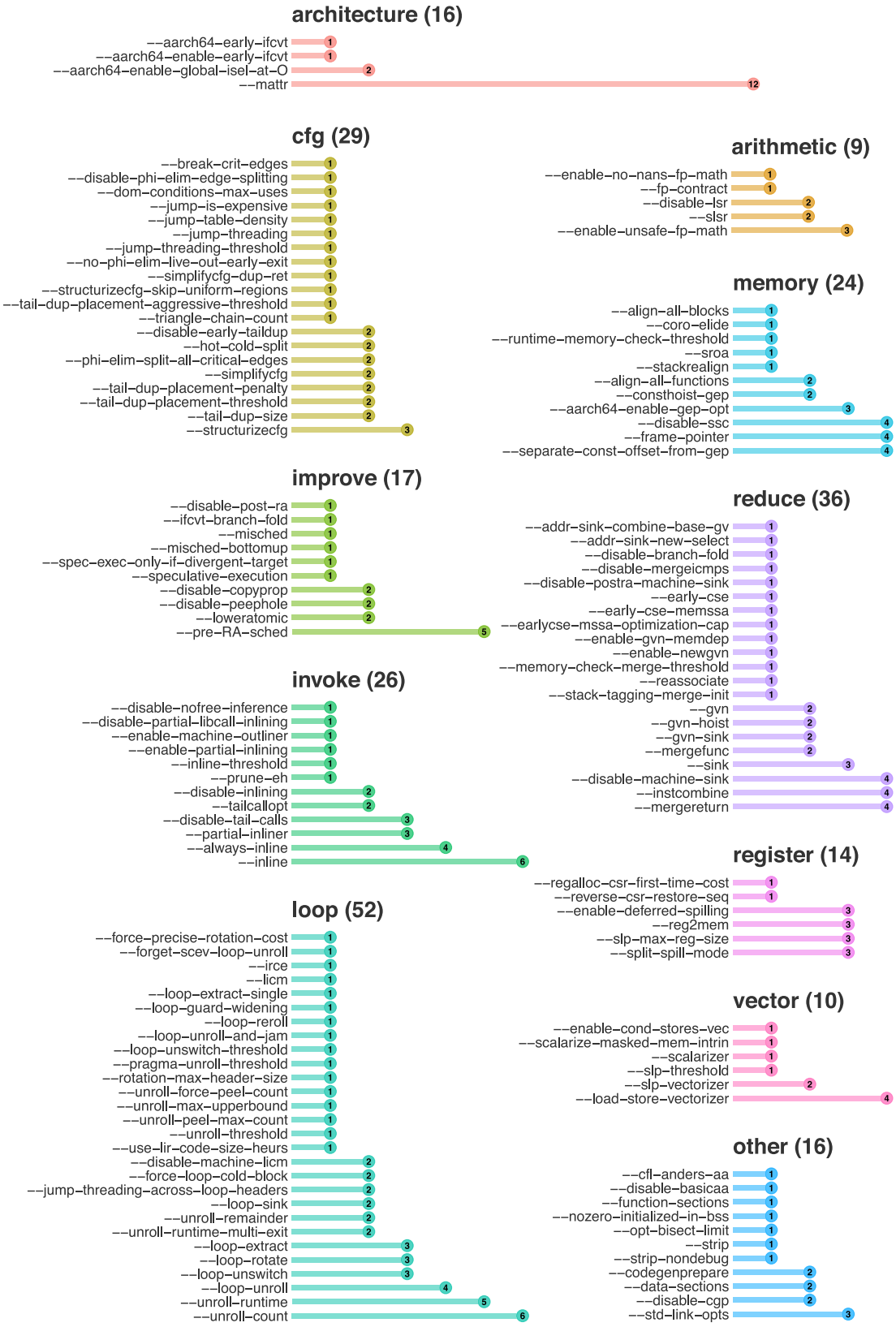

Figure 6.10: Categorizing the flags of the best GA findings based on the performed code transformations. The flag usage count across all 21 applications is also shown.

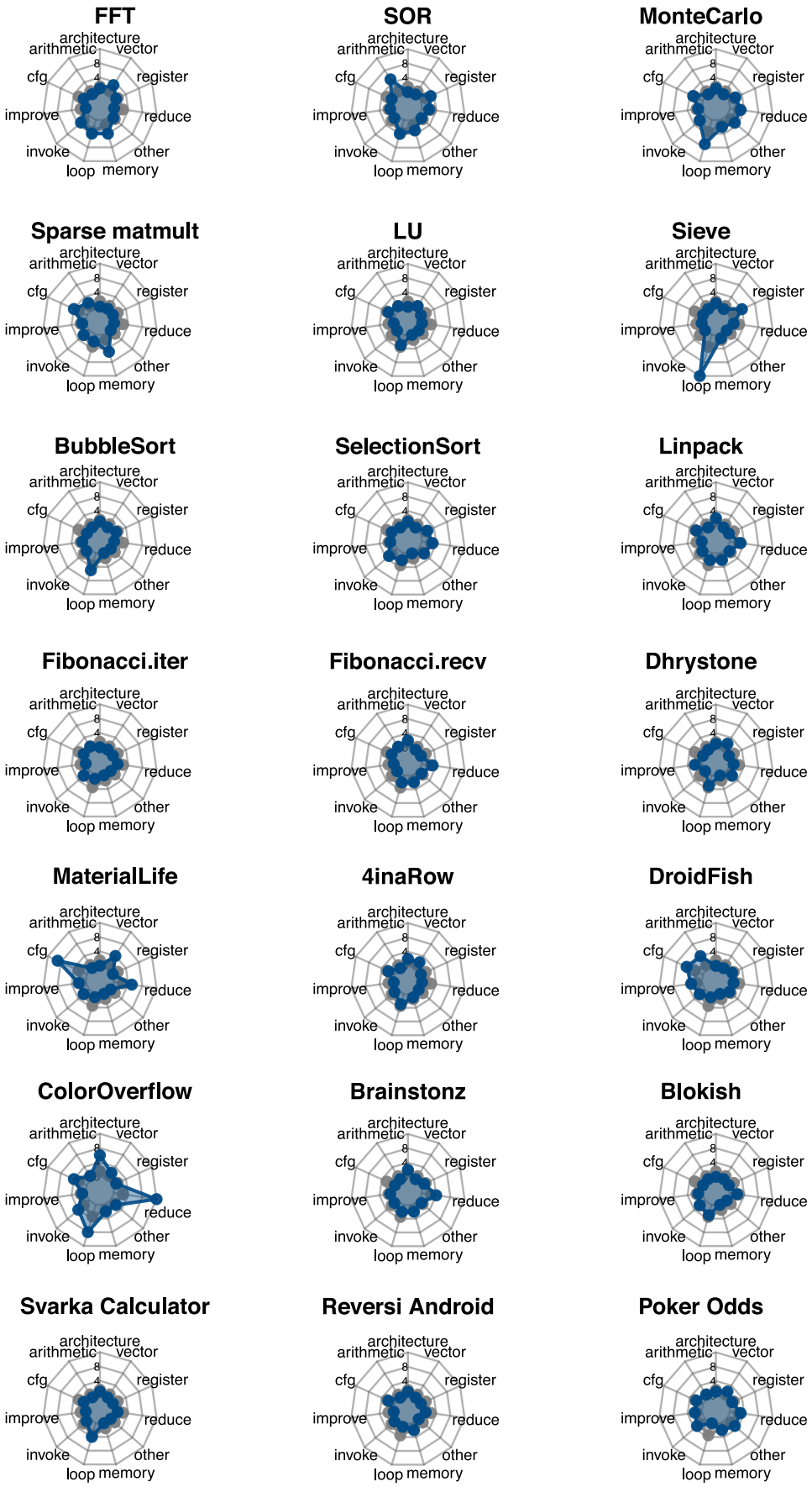

Figure 6.11: Overlaying the amount of flags used per category (see Figure 6.10) for each application, on top of the average usage across all 21 applications. The average usage is depicted with light gray color.

6.9.3 Transparent input capture mechanisms for Android processes

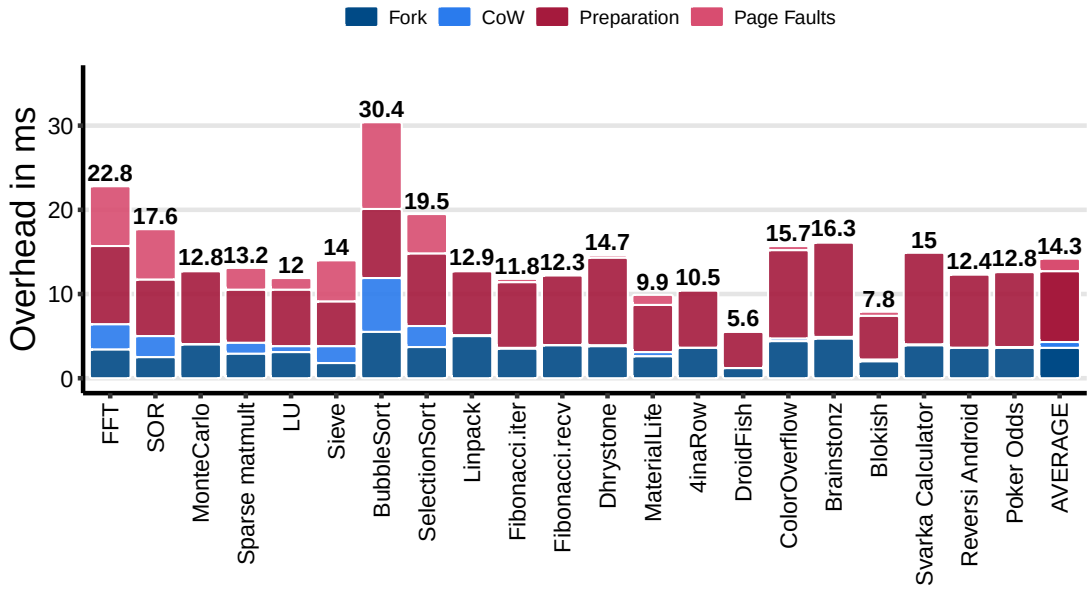

Figure 6.12: Breaking down the online capture overheads into 4 categories. All three capture mechanisms (*Pages*, *Everything*, and *Intersection*) require the entire *Fork* and *CoW* (Copy-On-Write) overheads. However, the capture *Everything* approach does not require any of the *Page Fault* overheads and almost none of the *Preparation*. Visually, the total overhead of *Everything* are the groups shown with the blue shading in the plot. On average, those are less than 6ms. The overheads of the *Pages* and *Intersection* mechanisms are identical. Visually, they require all four groups shown in the plot, with an average of 14.3ms and a maximum of roughly 30ms. Despite some higher overhead, those are still unnoticeable by the users.

Any optimization approach that might temporarily deteriorate the user experience is hard to justify. The only stage of the proposed system that happens online and could affect the user is the input capture. Therefore, it is important to establish that capture runtime overheads are low enough to have a negative effect on the user. This section presents the runtime overheads for the proposed input capture mechanisms: *Everything*, *Pages*, and *Intersection*. *Intersection* is an improvement over the *Pages* approach. It incurs the same overheads with *Pages*, as the extra operations performed are on a low-priority, child process. The sum of these overheads is 14.3ms on average, while the maximum was roughly 30ms. The more lightweight capture *Everything* approach, requires less than 6ms overhead on average. These runtime overheads are further broken down into four categories, which are described below.

Figure 6.12 shows a detailed breakdown of the overheads introduced by the infrequently invoked hot region input captures. *Fork* is the time it takes to call `fork` and return. It ranges roughly from 1ms to 6ms, with 3.6 ms on average, depending on the application and the state that needs to be replicated for the child. *CoW* is the overhead caused by the Copy-On-Write mechanism and ranges between roughly 1ms to 7ms, with less than 1ms on average, depending on the amount of pages that were modified. These two overhead sources are common for all three mechanisms. Preparation includes everything that is performed before executing the *hot region*, except the *fork* call. For *Pages* and *Intersection* this time is almost entirely spent parsing page mappings from the `/proc` *pseudo-filesystem* and read-protecting pages. It can take anything between 4ms and 11ms, depending on the number of page map entries that are processed. *Everything* does not require any of this preparation, requiring just 1% of that time. During the execution of the hot region there is an additional overhead for the *Pages* and *Intersection* approaches, due to the handling of the deliberate page faults. It is usually a very small fraction of the overhead, except for a few cases, like *BubbleSort* (16ms) and *FFT* (10ms). These benchmarks have a large number of input pages leading to increased *Page Faults*. *CoW*, which also occurs when the region runs, is increased as well for those cases, being around half of the *Page Fault* overheads.

6.9.4 Capture storage overheads

It is important that the capture mechanism does not hog the mobile device's limited storage capacity. While a single capture is not a problem on its own, a realistic system would have to work on optimizing multiple applications in parallel with, perhaps, multiple captures for each application. Making sure that the captured page set is small enough to allow tens of distinct captures is absolutely necessary. This section compares the snapshot size requirements for the proposed input capture mechanisms. One that requires less runtime operation and more storage, as it does not attempt to minimize capture sizes, against two that store significantly less data.

Figure 6.13 shows the storage overheads for all applications using different capture granularity. *Everything*, captures all pages that were marked as active in main memory by the time the hot region entered. It does not attempt to minimize the capture data, requiring 165MB on average with the highest being roughly 300MB. As the previous experiment has illustrated, it operates significantly faster, making this approach appealing for the cases where the storage is not very limited.

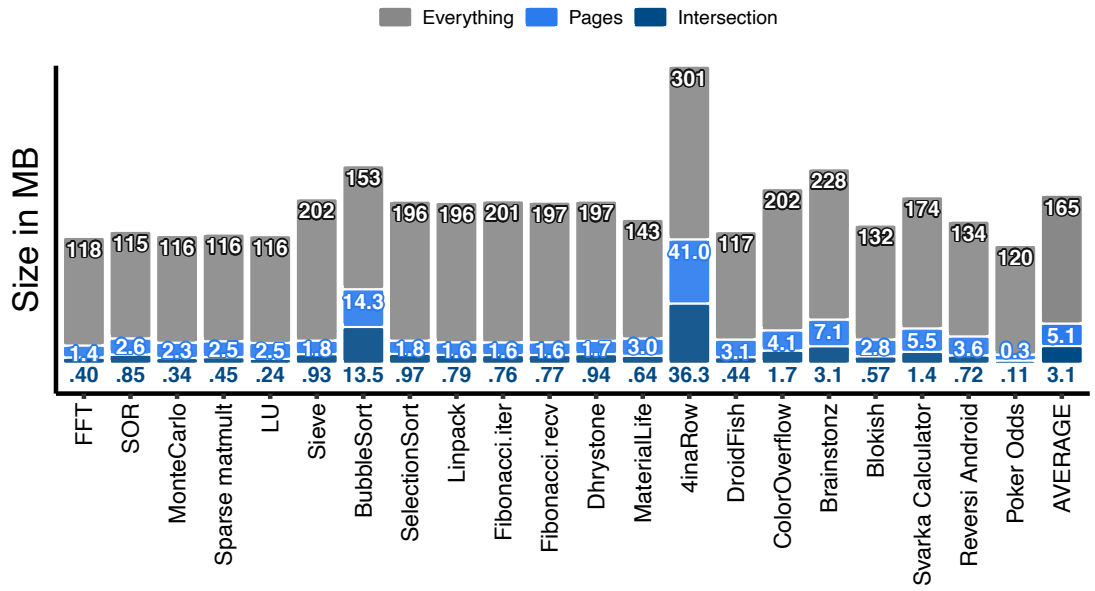

Figure 6.13: Storage overhead for capturing user inputs at different granularity. *Everything*, does not make any attempt on minimizing the capture sizes. *Pages* captures only the memory pages read by the *hot region* and decreases sizes by an order of magnitude. By intersecting the reachable heap objects with those read pages the required storage is further decreased, by an additional 64%.

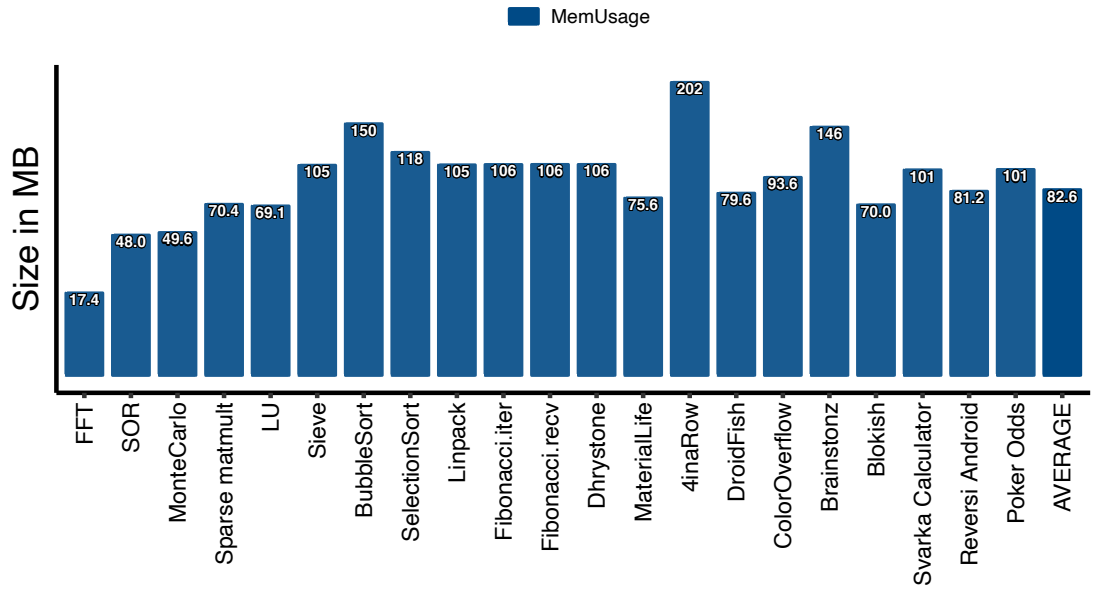

Figure 6.14: Memory usage of the captured benchmark and interactive applications. On average, applications utilized 82.6MB of heap data, with *FFT* having the lowest at 17.4MB and *4inaRow* the highest at 202MB. While this represents the majority of an application's runtime state, it is insufficient for setting up a working replay environment. This is because other areas are also needed while executing a hot region. For example, the GC is operating on an application's heap data, but to do so it depends on the runtime's (ART) memory areas like the stack, its internal heap (different than the application heap), and its text/code segment. The same applies for any other library dependencies that an application or the runtime has. Regardless, what the *Pages* capture mechanism actually stores is just a fraction of the heap space, while the object *Intersection* mechanism significantly reduces the required storage even further (see Figure 6.13).

The *Pages* approach minimizes the captured data by storing only the unique pages that have been read by the hot region. More than two thirds of the read pages are not unique to the process. They represent runtime instances of immutable structures that are identical across all processes created during the same device boot. A single capture of these common pages is enough for all apps, which requires 12.6MB of storage. The average size of unique pages per region is 5.06MB, the smallest is 356KB for *Poker Odds*, and the largest is 41MB for *4inaRow*. Mostly, it is between 1MB and 5MB and is in the same order of magnitude as the data the region actively uses.

Figure 6.14 shows the memory utilization by the applications. That is the total amount of data that resides in the runtime heap, which represents the majority of the application's dynamic state. For replays to work, the capture mechanism needs to store data that reside in other areas as well, which is why the capture *Everything* approach always contains more data than the entire heap alone. In any case, when considering this heap memory area in isolation, the *Pages* approach captures only a small fraction of what the capture *Everything* approach would store. This is because several heap objects reside outside the hot region input pages.

On average, the heap contained 82.6MB of data (Figure 6.14), while the *Pages* approach stored 5.1MB in total (Figure 6.13). The *Pages* mechanism, despite including data from other areas as well (apart from the application's heap), it required only 6% of what resides in the heap exclusively. Depending on the case, that percentage can be as low as 0.3%, for example capturing only 0.3MB (see Figure 6.13) out of the 88.4MB (see Figure 6.14) of *Poker Odds*'s heap space. For *4inaRow*, which had both the highest heap usage and the highest captured size, the proposed approach stored only 20% of what resided in the heap. For *FFT*, which had the least amount of runtime data, it stored 8% of the heap.

Despite these savings, there is still room for improvement. There are several heap objects that are unreachable by the hot region, and yet reside inside the input pages. These data are unnecessarily stored by the *Pages* approach. By doing a reachable object *intersection* with the input pages this space can be reclaimed, to reduce even further the capture sizes. The object *Intersection* mechanism requires only 3.1MB of storage on average, 64% less than the *Pages* approach. It stores an order of magnitude less data for *LU*, 7x less for *MonteCarlo* and *DroidFish*, and 2x less for most of the other ones. *BubbleSort* and *4inaRow* were two notable exceptions (still 10% improved) as they had dense objects in their input heap pages.

The *Pages* and *Intersection* capture mechanisms minimize the amount of required

storage. That level of storage is manageable even for low-end devices, especially given that this overhead is transient. Once the optimization search has finished for a particular application, then the capture data can be discarded releasing the storage space back to the user. Additionally, by reducing the size of the captures, the amount of work needed to set up a replay is similarly reduced, which speeds up the time required for mass-evaluating code transformations.

6.9.5 Acceleration of the optimization search with crowd-sourcing

Running iterative compilation offline means that any associated overheads from the technique will not have a negative impact on the user experience. Nevertheless, a decent evaluation effort must be put before better transformations are discovered. Finding better genomes with less effort makes the proposed replay-based iterative compilation approach even more effective.

To evaluate the collaborative GA search, 10 different mobile devices were used, as described in Section 6.8. Figure 6.15 shows the percentage that each user has contributed to such joint search. The first 3 users joined the search with only a 5-minute delay between them. As no other user was initially participating, they have contributed roughly half of the total search effort. The next 3 users also joined with a 5-minute delay between them, on top of a 30-minute delay from the first batch of users. Collectively they contributed 30%. The last 4 users have contributed a significantly less 22%. *User7* had the least contribution as it suspended evaluations when all other 9 users were actively participating.

Figure 6.16 visualizes the time each user has put into the joint search. Each user had a different contribution. This was due to their different availability for the joint search (see Figure 6.6), but also due to the different inputs they operated on. Only a fraction of each user's availability was required. *User7* contributed the least, for around 30 minutes. The most was 1 hour and 43 minutes by *User1*, who was the most active and also operated on bigger workloads for the benchmarks. The difference in the amount of computations performed by different users is also evident when comparing *Total* with *SingleUser1*. *Total* is the sum of all user contributions, while *SingleUser1* is the time required for performing a same-length search by *User1* individually (without crowd-sourcing).

The isolated search of *User1* required 13 hours and 7 minutes (*SingleUser1*), while the total time for joint search between all users required 11 hours and 15 minutes (*To-*

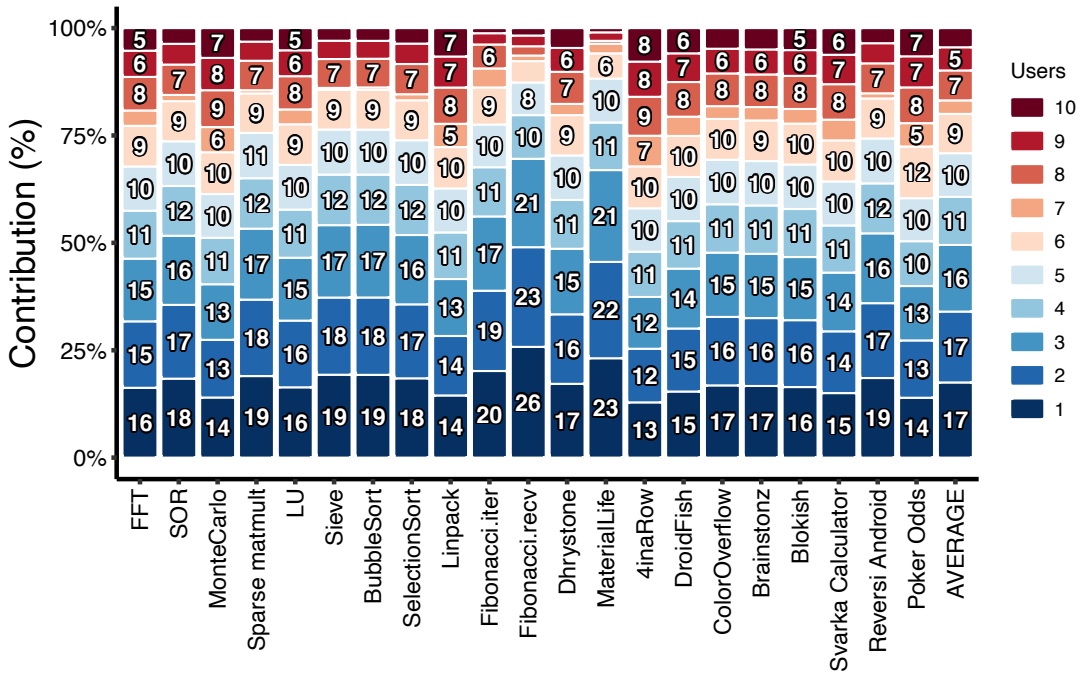

Figure 6.15: User participation to the collaborative optimization search for all applications. Users that joined the search earlier had higher contributions. The least contribution came from *User7*, who was absent at a period where all of the remaining users were actively contributing.

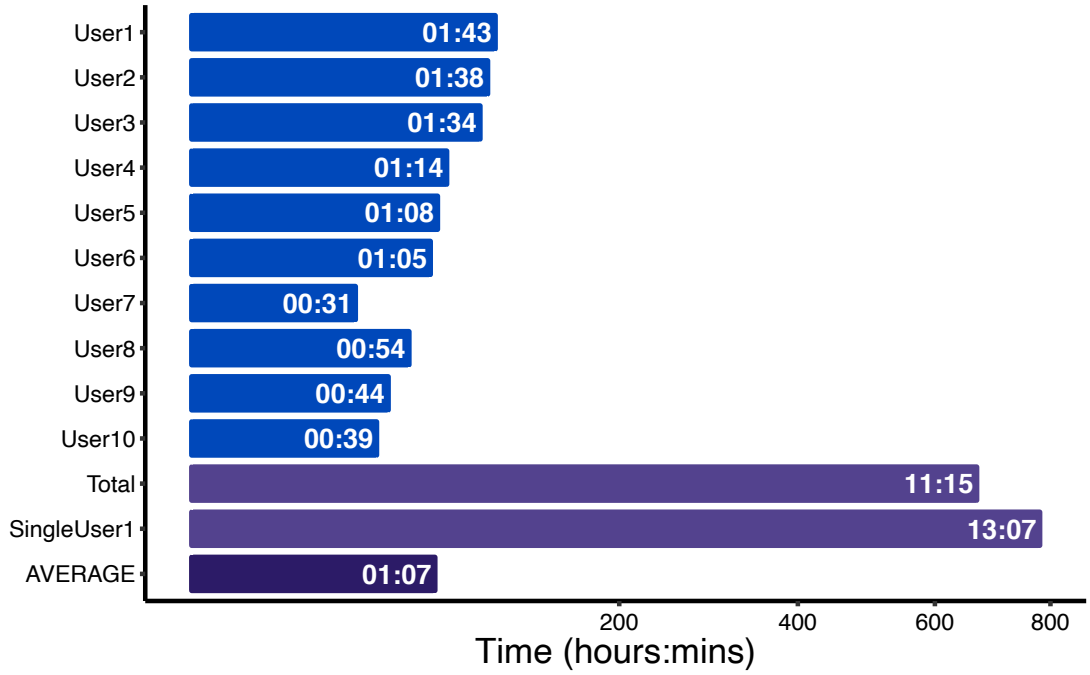

Figure 6.16: User search contribution time alongside with the total time and the average. *SingleUser1* shows the time required by *User1* to perform a same-length search on its own. Users had different contributions as they performed a different amount of computation. This was due to their different availability (see Figure 6.6) and the different inputs they operated on. *Total* shows the sum of all user contributions. It is actually less than the *SingleUser1*, as that particular user operated on input that caused increased workload. On average, the joint search required about an hour, roughly 12x less than *SingleUser1*.

tal). Therefore, in this particular scenario *User1* puts less than 12x less effort when compared to the *AVERAGE* contribution (*SingleUser1* ÷ *AVERAGE*). When not collaborating, 7x less effort is required (*SingleUser1* ÷ *User1*). When considering the availability time (see Figure 6.6), users were available for roughly 7 hours, with only 16% of that being utilized. These accelerations are enabled by the fact that new users leverage existing information accumulated by the previous ones. While variation is expected, as we only present a single run of a specific 10-user scenario, it becomes clear that the average user contribution time will decrease as the number of users increases, allowing us to perform deeper searches with less individual user effort.

6.9.6 Speedups for several users of a joint, crowd-sourced optimization search

While the acceleration benefits of a crowd-sourced approach are appealing, it is important to show that the findings of such collaborative search are still effective. This section presents the best genomes of a joint GA search among 10 users. It also analyses the cases where the input was significant.

Figure 6.17 shows the speedups from the joint optimization search of each user's *exit point*. That is the point in time where a participating user has exited the joint search. In essence, this point represents the best genomes that have been discovered from each user's participation periods. The geometric mean of all users and applications was 39%, only a 6% less than the individual-input search (see Section 6.9.1). Given the acceleration gains of a collaborative optimization search, this is a quite satisfactory result for a fraction of users' availability time.

For half of the cases the user input played a significant role in the findings. For *FFT*, most of the genomes were the same, except for 3 users that were 80% better than the individual search ones. A user in *Reversi* benefited an additional 60% improvement, while another one for *ColorOverflow* has tripled its findings. Users in *LU*, *4inaRow*, and *Poker Odds* had improvements between 25%-30%. Only 8 cases did not have much variation between different users. These results collectively indicate that despite doing a collaborative search among different users, the proposed approach is able to specialize, in most cases, for each user individually. *Fibonacci.iter* and *MonteCarlo* were two notable exceptions, where a single user deviated a lot from the other ones. In future work, we will consider either widening or even performing an individual search for such cases.

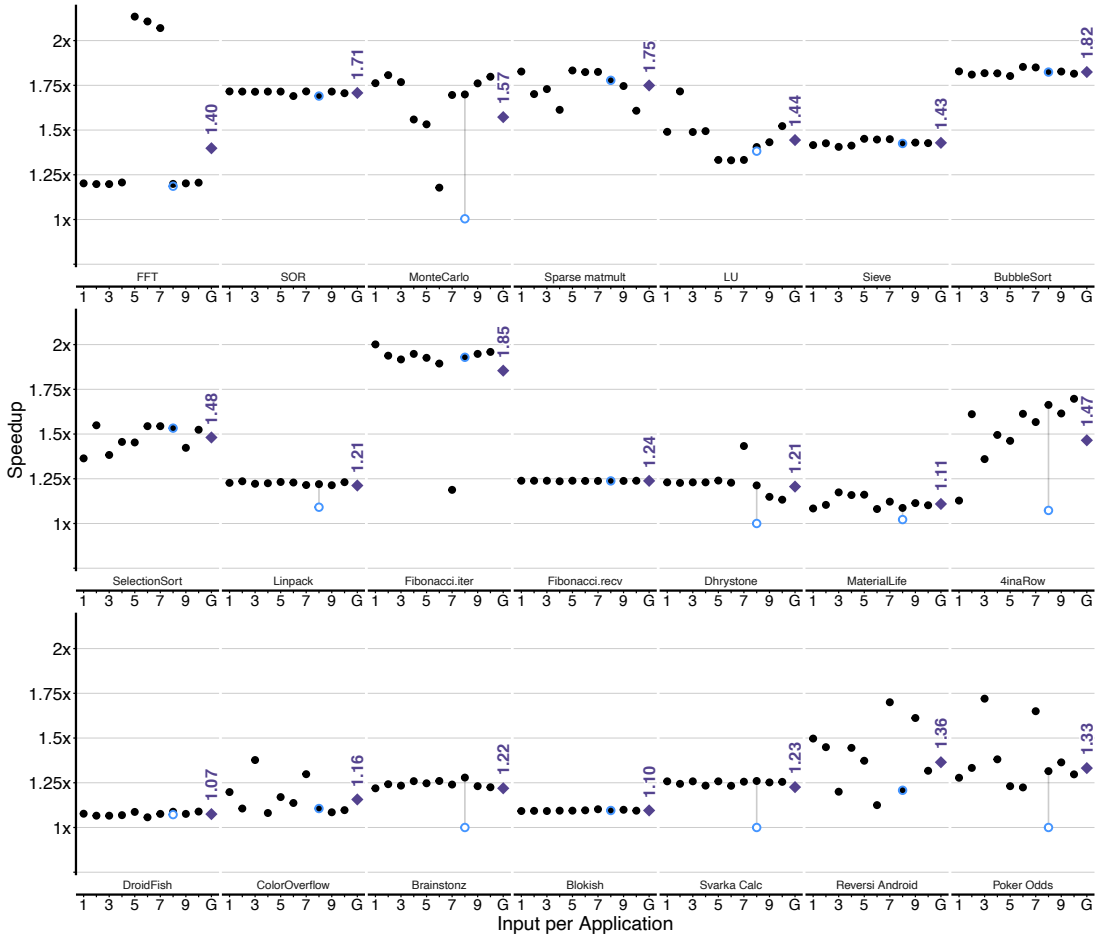

Figure 6.17: Showing the best genomes for each application from a joint search, at each user's *exit point*. That is the point in time where a user has exited the collaborative search. *User7* had two such points with the first shown as a circle outline drawn with a lighter color. The average speedup is shown with a rhombus. These findings suggest that the input was significant for half of the cases. Although they are not directly comparable with the single-user search of Section 6.9.1 (i.e., different number of points visited, plus variation between individual runs), 70% of the user averages were either better or within 15% of the best single-user genome.

Over time, the proposed architecture is also able to discover decent genomes within a handful of minutes. *User7*, participated for two distinct periods, with the first lasting only 15 minutes. Nevertheless, the user retrieved for around half of the cases a genome that was close to its optimal one. It is just for 5 cases it could not outperform the baseline. This makes the proposed technique effective even for users that participate just quick-charge device cycles.

6.10 Summary

There are many challenges for building an optimization framework for interactive mobile applications that is scalable, considers real inputs, introduces no overheads to the users, all without developer assistance. It should automatically identify code regions that are worth optimizing and can be accurately replayed. The input capture mechanism must work well in a complex runtime environment. The replay-based evaluation mechanism must repeatedly restore the captured input while being able to execute different code types as well different code versions of a hot region. It should operate even in the presence of memory-related security mechanisms. The code versions should be built by quickly exploring the beneficial areas of a compiler's transformation space. Finally, the correctness of tested code transformations must be verified automatically.

This chapter presented a system able to operate on interactive Android applications. By analyzing the bytecode of an application, code regions that are worth optimizing and can be accurately replayed are identified. Inputs to those regions are automatically captured on subsequent invocations of the application. Different input capture mechanisms were introduced. The *Pages* mechanism was reworked to operate on interactive applications. It minimizes the storage size requirements by storing only the read by the hot region memory pages. The *intersection* mechanism improves it by intersecting the reachable heap objects with the input pages. This significantly reduces capture sizes, without introducing any additional overheads. The capture *Everything* approach has a more lightweight runtime operation, as it does not attempt to minimize the storage sizes. It is ensured that complex runtime operations, like garbage collection, do not interfere with the capture mechanism. Also, the capture frequency and the capture input diversity can be controlled while the storage overheads are minimized. Then, offline iterative compilation can be performed by replaying the captured inputs. The replay mechanism starts as a vanilla C program and gradually transforms itself into a partial Android process. It works well in the presence of the ASLR security mechanism. It

can also replay the hot region using three different types of code. The first is the Android code, which is used as a baseline for comparisons. The second is interpretation and is used to extract data from the captures to automate correctness verification and enable further optimizations. The last one is code generated by the LLVM backend. Multiple versions of LLVM code are evaluated, which are generated by searching the optimization space using a genetic algorithm. Finally, with crowd-sourcing, the significant offline evaluation effort can split among different users to enable deeper searchers at a fraction of the time.

The proposed system was evaluated on 21 real Android applications, either benchmarks or interactive ones. The *Pages* and *Intersection* capture mechanisms introduce a slowdown of 14.3ms on average. The capture *Everything* approach requires less than 5ms on average. The *Pages* approach required 5.1MB of storage, while *Intersection* decreases that by an additional 64%. These two approaches allow multiple captures to be taken even on low-end devices. The capture *Everything* approach requires 165MB on average, making it a good fit only for the cases where the storage size is not very limited.

The replay-based optimization system was able to outperform the Android baseline for each application. The performance of the tested applications was improved by 44% on average. For almost all of the applications, the genetic algorithm keeps improving performance over time. A flag analysis suggested that some optimizations implemented in this and the previous chapters were exploited to a great extent by the proposed system. This, along with the significant performance improvements over the LLVM `-O3` baseline, showcase that even a highly optimizing compiler is not enough on its own. The GA also clearly demonstrates that online approaches, even if they improve over time, can significantly degrade the user experience. Some of the tested transformations were as much as 10x slower. Nine applications were still picking sub-optimal transformations even after three generations, while a few kept testing sub-optimal genomes to the very end. The collaborative search was just 6% short of the user-individual searches and required just a fraction of that time. With 10 users, the search was accelerated by 7x for the user with the highest workload, while another user was able to extract near optimal results only within a handful of minutes.

Chapter 7

Conclusions

This thesis attempts to address the main obstacles that prevent iterative compilation, a well-established optimization technique, to be widely adopted on mobile devices. Chapter 4 presents a lightweight capture mechanism for C functions, able to store real user inputs without causing noticeable overheads to the users. It also presents an offline replay-based evaluation mechanism that is combined with a random search based iterative compilation to optimize C programs. Chapter 5 introduces a novel LLVM backend for Android applications, to address the limited code transformation space of the default backend. Some Android-specific optimization passes were developed, along with an analysis pass to automatically detect code regions that are worth optimizing. Chapter 6 presents different input capture mechanisms that operate on different granularity and support Android applications. A novel replay mechanism is also introduced. It works well alongside memory-shuffling security mechanisms and supports executing different code types: Android default code, interpretation, and LLVM generated code. It is used by an iterative compilation system that automatically identifies accurately replayable code regions, searches the space using a genetic algorithm, and performs correctness verification without requiring any developer effort. Finally, with a crowd-sourcing module it accelerates the offline evaluation efforts.

This chapter is organized as follows. Section 7.1 briefly summarizes the main contributions of this thesis. Section 7.2 presents a critical analysis of the proposed approaches. In Section 7.3 there is a discussion on interesting future directions. Finally, Section 7.4 summarizes this chapter.

7.1 Contributions

The motivation examples presented in Section 1.1 showcase why neither offline nor online iterative compilation approaches are suitable for the highly restricted mobile device environment. The challenges of transparent capture and replay mechanisms were briefly outlined in Section 1.2. The shortcomings of existing systems in both research areas were reviewed in Chapter 3. The following subsections outline the main contributions of this thesis that allow iterative compilation to be widely adopted on mobile systems, by addressing these problems and filling the necessary gaps in the research literature.

7.1.1 Transparent captures of real user inputs

An offline code transformation evaluation mechanism relies on representative inputs that are captured online. Despite several existing capture approaches, successfully deployed in various domains, none was designed around a transformation evaluation mechanism with the mobile device environment in mind. A lightweight and unnoticeable online operation is imperative given the low-latency interactions and the limited computing, storage, and energy resources of mobile devices. Most of the existing approaches either minimize capture sizes at the cost of significant performance penalties, or capture at a coarser granularity requiring a considerable amount of storage. Some omit information that are necessary for an offline evaluation mechanism, while others introduce avoidable online overheads.

This thesis presented online input capture mechanisms for targeted code regions that are tailored for a low-latency mobile environment. The mechanisms are lightweight enough to remain unnoticeable from the users and are infrequently invoked, as a single capture is enough to drive code optimization. Chapter 4 details a capture mechanism that targets a hot method for C programs. It re-purposes two kernel mechanisms to store only a subset of the memory space while minimizing the overheads. Chapter 6 introduces different input capture mechanisms. The first, extends the capture mechanism presented in Chapter 4 to support code regions of the more complex Android applications and provides several improvements. In particular, captures are deferred on high-impact events that can potentially increase overheads, the capture frequency can be tuned, and the capture sizes are decreased by omitting immutable data from the snapshots. The second further minimizes the storage requirements, through heap object intersection with the accessed by the region memory pages. The last favors a

more lightweight runtime operation at the cost of higher storage sizes.

The capture mechanisms that minimize the snapshot sizes (*Pages* / *Intersection*) require 11ms for either C programs or Android applications. The mechanism that does not, requires 4.6ms for Android applications. Regarding storage, C programs, which are more lightweight than Android applications, required between 100KB and 200KB of storage. This was at least two orders of magnitude less than the program's total virtual memory. For Android applications, the *Page* capture approach required 5.06MB of storage on average. This amounts to only a 6% of the application's memory, which refers to the memory pages containing just the runtime heap (e.g., allocated objects/classes). To fully reconstruct a process, pages from other areas are needed as well. The object *Intersection* mechanism decreases the stored data even further, with an extra 64%. Once code optimization has finished, the capture is discarded. Given such low online execution overheads that only require a small and transient storage, the proposed approaches are suitable even for low-end mobile devices. The capture *Everything* approach requires 165MB on average, making it fit only for the cases where storage size is not limited.

7.1.2 Replay-based code transformation evaluation

Once real inputs are transparently captured, the next step is to use them for offline code transformation evaluations. The captured data have to be repeatedly restored to re-create for each evaluation a partial process environment for a particular code region. Each time, the mechanism must be able to consistently replay the region with a fixed input state while the underlying code for carrying out the execution can change.

Chapter 4 presents a replay evaluation mechanism for C functions. It employs compile and link time strategies for consistently replaying a function using the same input but under different optimization decisions. The input will be representative, as it was captured online, on a real user execution. Since the input is the same, comparisons between different decisions will be sound. As evaluations happen offline, at times when the device is charged and otherwise idle, any suboptimal, crashing, or erroneous evaluations can trivially be discarded without ever causing an inconvenience to the user. Offline evaluations additionally allow a tighter control over the execution environment. Combined with robust statistical methodologies, the execution noise becomes manageable even for the inherently noisy mobile environment.

Chapter 6 evolves the replay mechanism to work for the more complex Android

runtime. The mechanism is able to replay code regions using different code types and each region might consist of several methods. The first code type is the default Android code and is used as a baseline in evaluations. The second is interpretation. This enables dynamic profiling to be performed offline as a means of extracting information from the captures that would have been too costly to do online. This information is then used to provide automatic correctness verification of each evaluated transformation. It is also used to enable additional code optimizations, like speculative devirtualization. The final code type is LLVM code, which is utilized by the next contribution (see Subsection 7.1.3). Finally, the replay mechanism works in the presence of ASLR, a security mechanism that is present in all modern OSes, including Android.

7.1.3 Realizing iterative compilation on mobile systems.

Iterative compilation has been successfully applied over the years in several domains. Despite being a well-established optimization technique it has not been widely adopted on mobile systems. Offline approaches rely on representative hardware, software, and inputs. This assumption simply does not hold in the mobile environment as there is a multitude of hardware configurations, different software components are being updated frequently, and the artificial creation of representative inputs is non-trivial. Online approaches solve some of these issues, as they inherently operate on real inputs, but introduce a whole new set of problems. They might expose the user to significantly slower, crashing, or erroneous executions. On top of all these, input variability can extremely prolong the evaluation process to a point that is no longer practicable.

Chapter 4 presents a novel iterative compilation system for C functions that utilizes an offline, replay-based evaluation mechanism. Initially, a lightweight mechanism stores real user inputs without causing any noticeable overheads to the users. Then, an optimization search is performed by replaying offline the captured input under different optimization decisions. The proposed approach avoids the shortcomings of both purely offline and online approaches. Real user inputs are inherently representative and require no developer effort to be collected. Replaying the same inputs for each iterative compilation evaluation ensures sound comparisons between different optimization strategies. Doing this whole operation offline guarantees that at no time a user will be negatively affected by the optimization search.

Chapter 6 presents an iterative compilation system able to optimize interactive Android applications. It automatically identifies code regions that are worth optimizing,

which may contain multiple methods. With static analysis passes, it ensures that these regions can both be compiled by the LLVM backend and that they are able to replay in a deterministic fashion. It significantly increases the code transformation space of the default Android compiler utilizing the LLVM backend, presented in Chapter 5. It searches through this now huge space using a genetic algorithm, and automatically verifies each genome for correctness without requiring any developer effort. Finally, with a crowd-sourcing module, it allows several users to perform a joint search as a means of accelerating the offline evaluation efforts.

The proposed techniques were evaluated against C programs and Android applications. For C programs, the space was probed with a random search improving performance by 29%. For Android applications, the code was generated by the LLVM backend. On its own, and without utilizing iterative compilation, the LLVM backend improved applications by 7%. When utilizing a replay-based iterative compilation along with a genetic search, the performance of Android applications was improved by 44%. The collaborative GA search was just 6% short of that speedup, which is impressive given the acceleration gains. The user with the highest workload concluded the search 7x faster versus when not collaborating, and 12x when compared to the average. Another user was able to discover near-optimal results within a handful of minutes.

7.2 Critical Analysis

This section outlines the most important limitations of the techniques presented by this thesis.

7.2.1 Capture and Replay mechanism limitations

Specialized Android capture mechanism. To enable capturing of interactive Android applications the initial capture approach (see Chapter 4) had to change substantially. As ART had a more complex runtime environment, the page protection mechanism had to specially handle some virtual memory areas. This additional information, on which areas to memory-protect and which not, was acquired empirically. While it worked well, this specialization might need to be reworked on big OS updates.

Parsing serialized kernel structures. The *Page* capture mechanism operates entirely on the user space and is unnoticeable from the users. Most of its overhead is spent for parsing the VMA address space. It is necessary for identifying and protecting

the relevant memory areas, as well doing the specialization explained by the previous paragraph. This is done by reading and parsing the `/proc/self/maps` file, which is a serialized format of the kernel's internal VMA structures. Especially on the bigger 64-bit address space of Android applications, this parsing is slow.

Copy-On-Write and memory page protection overheads. These mechanisms are crucial for both identifying and preserving the original input pages. They are used by both the *Pages* and object *Intersection* capture mechanisms. They are triggered exclusively by user space code, and cause some actions to both user and kernel spaces. As demonstrated by the following example, these actions are not always optimal. When a parent process attempts to modify a memory-protected page, a segmentation violation is raised. This is handled in the user space by restoring the page permissions, so the parent can proceed to read or write the page. On write operations the Copy-On-Write mechanism will also be triggered. In that case, another set of actions are performed, now on the kernel space. The virtual page might now have the relevant permissions, however the physical page is still marked as non-writable. This is `fork`'s way of triggering Copy-On-Write. A page fault at the kernel space level is also raised, which is handled by copying the physical page, making the copy read-writable, and assigning it to the VMA of the parent process. In a nutshell, when writing a page both a segmentation violation and a page fault is raised and subsequently handled on the user and kernel spaces respectively.

Unnecessary data in captures and replay setup overheads. Regardless of being performed offline, a faster replay mechanism means more evaluations in less amount of time. Evaluations are already accelerated, as only the code that is being optimized is actually being replayed. Nevertheless, setting up less state during replays can decrease the time required for restoring. The child process can contain more data than what is actually needed. The Android capture mechanism never protects and therefore never captures pages containing executable code. For each of those, any relevant files and offsets are logged and remapped from disk during replay. Despite the file-backed memory mappings being an optimization regarding the storage space, they may introduce some overhead during the replay setup as all of those files are re-mapped.

Overestimating the reachable object set in input pages. The intersection mechanism identifies the reachable objects in the input pages of the runtime heap. These are the pages that belong to the heap and have been read by the hot region during its execution. The mechanism then proceeds by removing any unreachable objects that reside in those

heap pages, before materializing them to disk. The reachable object set, however, is overestimated in the current version. There might be cases where an object is marked as reachable without having a valid traversal path. Such path, is one that on each step, during a traversal from the roots to the object of interest, has any of the intermediate objects residing only in read pages.

7.2.2 Limitations of the LLVM backend

Being at its infancy, the LLVM backend has several opportunities for improvement. Several instructions are currently not as optimally implemented as the Android default backend. Others are either not fully or not at all implemented, relying on much slower runtime routines. This decreases the amount of code that LLVM can compile, and increases the external runtime calls. Both hinder compiler optimization. Special register operations are another set of instructions, extensively used by application code, that cannot fully benefit from optimization passes such as instruction reordering or merging. Also, the generated LLVM code is not neatly packed into special binary files as the Android compiled code is, therefore it cannot efficiently access several special caches for frequently used objects or classes. Section 5.3.4 presents these limitations in more detail.

7.2.3 Iterative compilation limitations

While it is shown that the proposed optimization mechanisms can operate well on methods that have deterministic code regions, it is harder to adapt to code that has inter-mi xed CPU intensive and non-deterministic operations. Smaller code traces can still be optimized in those cases, however, the bigger the contiguous code blocks the better when it comes to code optimization. Another limitation is that the input can change at runtime. When it does, the pre-optimized code might not be as efficient as it was for the input that it was originally optimized for.

7.3 Future work

This section outlines interesting future directions for the work presented in this thesis.

7.3.1 Improving the capture and replay mechanism

Despite being unnoticeable from the users, the overheads of the *Pages* and object *Intersection* capture approaches can be further reduced. A significant amount of the introduced overhead relates to parsing serialized kernel structures for understanding the virtual memory layout. Other overheads include the page protection and handling mechanism, as well the Copy-On-Write mechanism. Also, some specialization is needed on the memory areas that are being protected, otherwise the user space mechanism can prematurely crash.

With the introduction of a special system call, along with relevant kernel support, these overheads could be decreased and the mechanism can become independent of the runtime environment being captured. A high level overview of such call is described as follows. The system call will create another process, in a paused state, but in contrast with `fork` it will not duplicate all of the VMAs. It will still mark the physical pages of the original process, to cause kernel-space page faults. For each of those faults, it will update the new process with the relevant virtual page. It will preemptively copy the pages before they are modified by the parent in the new process, and it will note whether they are eventually read by the parent. Those that are only written but never read, will be removed from the new process. Regarding the executable pages, it will update the VMAs of the new process with the whole area, instead of only the relevant page. Those will be re-mapped directly from the filesystem during replay, using any relevant offsets. This will omit from the replay setup any code areas that were not used during the original execution.

This would require no preparation phase by the capture process. Also, there is no need to engage the page protection mechanism or do any user-space fault handling. The new process will gradually get filled with the read pages, the original content of the written pages, and the used executable areas. Then, the new process will disengage from the original process. It will be set to low priority and a special text segment will bootstrap for storing all the data that should be captured. This approach will work regardless of the runtime environment being captured.

7.3.2 Replaying code on different processing units

The presented capture and replay mechanisms operate on particular CPUs that run on a Linux OS device. Specifically, the processors based on the *arm* and *arm64* architectures are supported. By specializing on which registers to store, one could port the

system to more CPU architectures, like the *x86*. However, more work is required to support different processing units, like Graphics Processing Units (GPUs) or a Digital Processing Units (DSPs). The remainder of this section describes those changes, given that the device has some relevant Linux kernel support.

To enable replaying code on a CPU, capturing data from two sources is required. The first is the main memory and the second is the CPU-specific memory (i.e., registers). The main memory is captured by utilizing OS interfaces to the Memory Management Unit (MMU) controller [LIN21b], exposed through the `/proc` interface [LIN20]. The CPU-specific memory is captured by accessing the registers through some assembly code. In particular, the contents of the non-volatile registers are captured. To enable replaying of code that runs on different processing units (like a GPU or a DSP), access to equivalent data sources is required. In particular, to store the device's memory one must utilize the Input/Output Memory Management Unit (IOMMU) of the kernel [LIN21a]. Similarly, the device registers have to be stored. This can be accomplished by retrieving the relevant *PCI device* through an interface [HER21]. By using such interface, the I/O regions can be computed, which contain the device's registers. Then, those can be memory mapped (`mmap`) in an operation that requires elevated privileges. Once this is done, the register input data can be retrieved. With an inverse procedure, the captured data can be restored back to the processing unit at the replaying phase.

7.3.3 Higher quality of the IR-to-IR translation pass

Despite being a work in progress, the LLVM backend outperforms the default Android compiled code for most cases. Nevertheless, the code produced by the LLVM backend is not as efficient as it could be. Particular JNI methods could be translated into LLVM bitcode, either by using LLVM intrinsics or by re-writing any relevant functionality into LLVM IR. This can increase the amount of code that is compiled and therefore optimized. The *post-unroll* optimization could be expanded to additional slow check operations in loops, such as cases where array bounds checking is unnecessary. Also, Android compiled code accesses caches faster than LLVM. This is because it is packed into special OAT files that enable, with some link-time patching, direct memory accesses to caches of objects, classes, or strings. With additional effort this mechanism can be ported to LLVM. Finally, LLVM can be extended to support reserving and re-purposing registers, similar to what the Android compiler does for some

special purpose registers, like the Thread or the Marking register.

7.3.4 Input-Specialization on the crowd-sourced optimization search

The findings of the collaborative search suggest that the input was significant for many cases when it came to code optimization. While the approach is capable of doing a joint search, it can still specialize for individual users when needed. However, this was not the case for a few of the cases. Particular users deviated a lot from the others. To improve this, we are considering doing individual user searches, at least to some extent. The joint GA search could also be extended, to cluster over time different users according to the genomes that are most effective for them. This will encourage deeper exploration in areas that might be beneficial for particular user inputs.

7.3.5 Optimizing for common inputs and using at runtime the best

The presented iterative compilation approaches perform an input-driven optimization search. Therefore, the findings are optimized for a particular input. As these inputs change, the previously discovered optimized code might not be as efficient. To overcome this problem, one must maintain different optimized code versions for the frequently encountered inputs and dynamically choose the best one at runtime.

By periodically taking captures of an application, a set of the most frequently encountered inputs can be maintained. Running a replay-based iterative compilation on these inputs will result in a set of pre-optimized binaries for a particular code region. At runtime, all these pre-optimized versions could be available for carrying out the execution of the region. With a lightweight input identification mechanism, the runtime could predict which version will have the best performance and choose it for execution. This dynamic optimization is known as code multi-versioning.

7.4 Summary

Neither offline nor online iterative compilation approaches are well suited for the mobile environment. Offline approaches require representative hardware, software, and inputs. This is an assumption that is hard to make. There are a myriad of hardware designs, software is constantly being updated, while creating artificial input (representative or not) is a non-trivial endeavor for mobile applications. Online approaches on

the other hand, inherently solve some of these issues, however they expose users to suboptimal or even erroneous executions.

This thesis has presented a novel fusion of iterative compilation with a capture and replay mechanism. With infrequent captures, real inputs are stored without causing any noticeable overheads to the users. Then, those are used to drive an offline, replay-based iterative compilation. The replay mechanism is able to repeatedly execute the code region with the same input, but under different optimization decisions. As the input is the same, comparisons between different decisions are sound. As this happens offline, any suboptimal or erroneous executions never affect the users. This additionally allows a stricter execution environment, which is combined with robust statistical methodologies to make the execution noise manageable.

The proposed approaches were evaluated on C programs and then were extended to support Android applications. Different input capture mechanisms were also developed, able to store input at different granularity. To increase the code transformation space of the default Android compiler, an LLVM backend was implemented. The backend was open-sourced to give more researchers the option of producing highly optimized code for Android applications. The compilable, replayable, and optimization-worthy code regions are automatically identified through profiling and static bytecode analysis. A novel replay mechanism allows performing dynamic profiling offline, which enables automatic correctness verification, as well further compiler optimization. The optimization search is performed using a genetic algorithm. Finally, with crowd-sourcing, the offline evaluation efforts are significantly accelerated. The preliminary results were promising, and the potential future directions for this research look very interesting.

Bibliography

[AAR+97] Bas Aarts, Michel Barreteau, François Bodin, Peter Brinkhaus, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John Gurd, Jan Hoogerbrugge, Ping Hu, et al. "OCEANS: Optimizing compilers for embedded applications". In: *Euro-Par'97 Parallel Processing*. Springer, 1997, pp. 1351–1356.

[AGA+06] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. "Using machine learning to focus iterative optimization". In: *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE. 2006, 11–pp.

[ALG20] The Algorithms. *Sorting Algoirthms*. 2020. URL: `https://github.com/TheAlgorithms/Java` (visited on 01/11/2021).

[ALM+03] L Almagor, Keith D Cooper, Alexander Grosul Timothy J Harvey, Steve Reeves Devika Subramanian, Linda Torczon, and Todd Waterman. *Compilation order matters: Exploring the structure of the space of compilation sequences using randomized search algorithms*. Tech. rep. Technical report, Los Alamos Computer Science Institute, 2003.

[ALM+04] Lelac Almagor, Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven W Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. "Finding effective compilation sequences". In: *ACM SIGPLAN Notices* 39.7 (2004), pp. 231–239.

[AND20] Android. *Android compiler: pathological compilation cases*. 2020. URL: `https://android.googlesource.com/platform/art/+/refs/tags/android-10.0.0_r11/compiler/compiler.cc#48` (visited on 11/12/2020).

[AND21a] Android. *Android JIT compilation*. 2021. URL: https://cs.android.com/android/platform/superproject/+/master:art/compiler/jit/jit_compiler.cc (visited on 01/11/2021).

[AND21b] Android. *APEX: A Container format introduced in Android 10*. 2021. URL: https://source.android.com/devices/tech/ota/apex (visited on 01/11/2021).

[AND21c] Android. *OAT files format on ANdroid*. 2021. URL: https://cs.android.com/android/platform/superproject/+/master:art/runtime/oat.h (visited on 01/11/2021).

[AND21d] Android/LLVM. *Existing LLVM module on AOSP (external/llvm)*. 2021. URL: https://android.googlesource.com/platform/external/llvm/ (visited on 01/11/2021).

[APP20] Quarzo Apps. *4 in a row*. 2020. URL: https://play.google.com/store/apps/details?id=com.quarzo.fourinarow&hl=en&gl=US (visited on 01/11/2021).

[ASH+17] Amir H Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni, and John Cavazos. "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 14.3 (2017), pp. 1–28.

[BAL+00] 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*. 2000, pp. 1–12.

[BAL+94] Thomas Ball and James R Larus. "Optimally profiling and tracing programs". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.4 (1994), pp. 1319–1360.

[BAO+16] Wenlei Bao, Changwan Hong, Sudheer Chunduri, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P Sadayappan. "Static and dynamic frequency scaling on multicore CPUs". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.4 (2016), pp. 1–26.

[BOD+98] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. "Iterative compilation in a non-linear optimisation space". In: *Workshop on Profile and Feedback-Directed Compilation*. 1998.

[BRI20] University of Bristol. *The BEEBS Benchmark Suite*. 2020. URL: https://github.com/mageec/beebs (visited on 01/11/2021).

[CAS+15] Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. "CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization". In: *ACM Trans. Archit. Code Optim.* 12.1 (Apr. 2015).

[CAV+07] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. "Rapidly selecting good compiler optimizations using performance counters". In: *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE. 2007, pp. 185–197.

[CDC21] CDC. *CDC Sleep Statistics for the United States*. 2021. URL: https://www.cdc.gov/sleep/data_statistics.html (visited on 01/11/2021).

[CHE+12] Yang Chen, Shuangde Fang, Lieven Eeckhout, Olivier Temam, and Chengyong Wu. "Iterative optimization for the data center". In: *ACM SIGARCH Computer Architecture News* 40.1 (2012), pp. 49–60.

[CLA20] LLVM Clang. *Clang: LLVM front-end for C programs*. 2020. URL: https://clang.llvm.org/ (visited on 01/11/2021).

[CLA21] Clang. *Clang compatibility with GCC*. 2021. URL: http://clang.llvm.org/compatibility.html (visited on 01/11/2021).

[COH+05] Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. "Facilitating the search for compositions of program transformations". In: *Proceedings of the 19th annual international conference on Supercomputing - ICS '05* (2005), p. 151.

[COO+02a] K Cooper, Timothy Harvey, Devika Subramanian, and Linda Torczon. *Compilation order matters*. Tech. rep. Technical Report, Rice University, 2002.

[COO+02b] Keith D Cooper, Devika Subramanian, and Linda Torczon. "Adaptive optimizing compilers for the 21st century". In: *The Journal of Supercomputing* 23.1 (2002), pp. 7–22.

[COO+05] Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. “ACME: adaptive compilation made efficient”. In: *ACM SIGPLAN Notices* 40.7 (2005), pp. 69–77.

[COO+86a] Keith D Cooper, Ken Kennedy, and Linda Torczon. “Interprocedural optimization: Eliminating unnecessary recompilation”. In: *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*. 1986, pp. 58–67.

[COO+86b] Keith D. Cooper, Ken Kennedy, and Linda Torczon. “The impact of interprocedural analysis and optimization in the Rn programming environment”. In: *ACM Transactions on Programming Languages and Systems* 8.4 (Aug. 1986), pp. 491–523.

[COO+99] Keith D Cooper, Philip J Schielke, and Devika Subramanian. “Optimizing for reduced code space using genetic algorithms”. In: *ACM SIGPLAN Notices*. Vol. 34. 7. ACM. 1999, pp. 1–9.

[CUM+17a] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. “End-to-end deep learning of optimization heuristics”. In: *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE. 2017, pp. 219–232.

[CUM+17b] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. “Synthesizing benchmarks for predictive modeling”. In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 86–99.

[CUM+18] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. “Compiler fuzzing through deep learning”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM. 2018, pp. 95–105.

[DEN06] Peter J Denning. “The locality principle”. In: *Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe*. World Scientific, 2006, pp. 43–67.

[DEU96] Peter Deutsch. *RFC1951: Deflate compressed data format specification version 1.3*. 1996. URL: `https://www.rfc-editor.org/info/rfc1951` (visited on 01/11/2021).

[DIN+07] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. “Software behavior oriented parallelization”. In: *ACM SIGPlan Notices* 42.6 (2007), pp. 223–234.

[DON+79] Jack J Dongarra, Cleve Barry Moler, James R Bunch, and Gilbert W Stewart. *LINPACK users’ guide*. SIAM, 1979.

[F-D21] F-Droid. *F-Droid: FOSS Android application store*. 2021. URL: `https://f-droid.org/` (visited on 01/11/2021).

[FAN+15] Shuangde Fang, Wenwen Xu, Yang Chen, Lieven Eeckhout, Olivier Temam, Yunji Chen, Chengyong Wu, and Xiaobing Feng. “Practical iterative optimization for the data center”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 12.2 (2015), pp. 1–26.

[FAT+04] Deji Fatiregun, Mark Harman, and Robert M Hierons. “Evolving transformation sequences using genetic algorithms”. In: *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. IEEE. 2004, pp. 65–74.

[FEL20] FelipeRRM. *Reversi Android*. 2020. URL: `https://github.com/FelipeRRM/AndroidReversi` (visited on 01/11/2021).

[FER+11] Denzil Ferreira, Anind K Dey, and Vassilis Kostakos. “Understanding human-smartphone concerns: a study of battery life”. In: *International Conference on Pervasive Computing*. Springer. 2011, pp. 19–33.

[FUR+02] GG Fursin, Michael FP O’Boyle, and Peter MW Knijnenburg. “Evaluating iterative compilation”. In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 2002, pp. 362–376.

[FUR+05] Grigori Fursin, Albert Cohen, Michael O’Boyle, and Olivier Temam. “A practical method for quickly evaluating program optimizations”. In: *International conference on high-performance embedded architectures and compilers*. Springer. 2005, pp. 29–46.

[FUR+08] Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, et al. “MILEPOST GCC: machine learning based research compiler”. In: *GCC Summit*. 2008.

[FUR+09] Grigori Fursin and Olivier Temam. “Collective optimization”. In: *International Conference on High-Performance Embedded Architectures and Compilers*. Springer. 2009, pp. 34–49.

[GCC20a] GCC. *GCC compiler*. https://gcc.gnu.org/. 2020. (Visited on 10/01/2020).

[GCC20b] GCC. *Global Register Variables*. 2020. URL: https://gcc.gnu.org/onlinedocs/gcc/Global-Register-Variables.html (visited on 01/11/2021).

[GOM+13] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. “Reran: Timing-and touch-sensitive record and replay for android”. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 72–81.

[GOO20a] Google. *Android Optimizing compiler backend*. 2020. URL: https://android.googlesource.com/platform/art/+/refs/tags/android-10.0.0_r11/compiler/optimizing/optimizing_compiler.cc (visited on 11/01/2020).

[GOO20b] Google. *Android Optimizing compiler backend: code transformations*. 2020. URL: https://android.googlesource.com/platform/art/+/refs/tags/android-10.0.0_r11/compiler/optimizing/optimization.h#68 (visited on 11/01/2020).

[GOO20c] Google. *Android Optimizing compiler backend: special registers*. 2020. URL: https://cs.android.com/android/platform/superproject/+/master:art/runtime/arch/arm64/registers_arm64.h;l=63 (visited on 11/01/2020).

[GOO20d] Google. *Android Runtime fault handler*. 2020. URL: https://cs.android.com/android/platform/superproject/+/master:art/runtime/fault_handler.cc;l=209 (visited on 01/11/2021).

[GOO20e] Google. *Android Zygote: A special process for optimizing memory pages for applications*. 2020. URL: https://developer.android.com/topic/performance/memory-overview#SharingRAM (visited on 01/11/2021).

[GOO20f] Google. *Soong build system for the AOSP project*. 2020. URL: https://source.android.com/setup/build (visited on 01/11/2021).

[GOO21a] Google. *Bionic: Android's C library, math library, and dynamic linker*. 2021. URL: https://android.googlesource.com/platform/bionic/ (visited on 01/11/2021).

[GOO21b] Google. *Google Play store: Official Android application store*. 2021. URL: https://play.google.com/store/apps (visited on 01/11/2021).

[GOR+02] Ann Gordon-Ross, Susan Cotterell, and Frank Vahid. "Exploiting fixed programs in embedded systems: A loop cache example". In: *IEEE Computer Architecture Letters* 1.1 (2002), pp. 2–2.

[GSM20] GSM. *The Mobile Economy 2020*. 2020. URL: https://www.gsma.com/mobileeconomy/wp-content/uploads/2020/03/GSMA_MobileEconomy2020_Global.pdf (visited on 11/12/2020).

[HER+11] Ben Hertzberg and Kunle Olukotun. "Runtime automatic speculative parallelization". In: *International Symposium on Code Generation and Optimization (CGO 2011)* (Apr. 2011), pp. 64–73.

[HER21] Rob Herring. *Generic PCI access library*. 2021. URL: https://github.com/robherring/libpciaccess (visited on 01/11/2021).

[HU+15] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. "Versatile yet lightweight record-and-replay for android". In: *ACM SIGPLAN Notices* 50.10 (2015), pp. 349–366.

[HUR+10] Joshua Hursey, Chris January, Mark O'Connor, Paul H Hargrove, David Lecomber, Jeffrey M Squyres, and Andrew Lumsdaine. "Checkpoint/restart-enabled parallel debugging". In: *European MPI Users' Group Meeting*. Springer. 2010, pp. 219–228.

[ISH20] Takanori Ishikawa. *Fibonacci*. 2020. URL: https://gist.github.com/ishikawa/16670 (visited on 01/11/2021).

[JAN+05] G John Janakiraman, Jose Renato Santos, Dinesh Subhraveti, and Yoshio Turner. "Cruz: Application-transparent distributed checkpoint-restart on standard operating systems". In: *2005 International Conference on Dependable Systems and Networks (DSN'05)*. IEEE. 2005, pp. 260–269.

[JET20] JetBrains. *Kotlin programming language*. https://kotlinlang.org/. 2020. (Visited on 10/01/2020).

[JHA+13] Ajay K Jha and Woo J Lee. “Capture and Replay Technique for Reproducing Crash in Android Applications”. In: *Proceedings of the 12th IASTED International Conference in Software Engineering*. 2013, pp. 783–790.

[JOS+07] Shrinivas Joshi and Alessandro Orso. “SCARPE: A technique and tool for selective capture and replay of program executions”. In: *2007 IEEE International Conference on Software Maintenance*. IEEE. 2007, pp. 234–243.

[KEL+09] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. “Fast Track: A Software System for Speculative Program Optimization”. In: *2009 International Symposium on Code Generation and Optimization* (Mar. 2009), pp. 157–168.

[KIS+00] Toru Kisuki, P Knijnenburg, M O’Boyle, and H Wijshoff. “Iterative compilation in program optimization”. In: *Proc. CPC’10 (Compilers for Parallel Computers)*. Citeseer. 2000, pp. 35–44.

[KIS+99] Toru Kisuki, Peter M Knijnenburg, M O’Boyle, François Bodin, and Harry A Wijshoff. “A feasibility study in iterative compilation”. In: *High Performance Computing*. Springer. 1999, pp. 121–132.

[KNI+01] Peter MW Knijnenburg, Toru Kisuki, and Michael FP O’Boyle. “Iterative compilation”. In: *International Workshop on Embedded Computer Systems*. Springer. 2001, pp. 171–187.

[KUL+03] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. “Finding effective optimization phase sequences”. In: *ACM SIGPLAN Notices* 38.7 (2003), pp. 12–23.

[KUL+12] Sameer Kulkarni and John Cavazos. “Mitigating the compiler optimization phase-ordering problem using machine learning”. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 2012, pp. 147–162.

[KUR+01] Dawid Kurzyniec and Vaidy Sunderam. “Efficient cooperation between Java and native codes–JNI performance benchmark”. In: *The 2001 international conference on parallel and distributed processing techniques and applications*. Citeseer. 2001.

[LAT+04] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, p. 75.

[LEA+09a] Hugh Leather, Edwin Bonilla, and Michael O’Boyle. “Automatic feature generation for machine learning based optimizing compilation”. In: *2009 International Symposium on Code Generation and Optimization*. IEEE. 2009, pp. 81–91.

[LEA+09b] Hugh Leather, Michael O’Boyle, and Bruce Worton. “Raced profiles: efficient selection of competing compiler optimizations”. In: *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. 2009, pp. 50–59.

[LEE+99] Lea Hwang Lee, Bill Moyer, and John Arends. “Instruction fetch energy reduction using loop caches for embedded applications with small tight loops”. In: *Proceedings of the 1999 international symposium on Low power electronics and design*. 1999, pp. 267–269.

[LEI+09] Andreas Leitner, Alexander Pretschner, Stefan Mori, Bertrand Meyer, and Manuel Oriol. “On the Effectiveness of Test Extraction without Overhead”. In: *2009 International Conference on Software Testing Verification and Validation* (Apr. 2009), pp. 416–425.

[LIM+13] Ewerton Daniel de Lima, Tiago Cariolano de Souza Xavier, Anderson Faustino da Silva, and Linnyer Beatryz Ruiz. “Compiling for performance and power efficiency”. In: *2013 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE. 2013, pp. 142–149.

[LIN+08] San-Chih Lin, Chi-Kuang Chang, and Nai-Wei Lin. “Automatic selection of GCC optimization options using a gene weighted genetic algorithm”. In: *2008 13th Asia-Pacific Computer Systems Architecture Conference*. IEEE. 2008, pp. 1–8.

[LIN20] Linux. *`/proc` pseudo-filesystem on Linux*. 2020. URL: `https://www.kernel.org/doc/html/latest/filesystems/proc.html` (visited on 01/11/2021).

[LIN21a] Linux. *Input/Output Memory Management Unit (IOMMU) on Linux*. 2021. URL: https://www.kernel.org/doc/html/latest/x86/intel-iommu.html (visited on 01/11/2021).

[LIN21b] Linux. *Memory Management Unit (MMU) on Linux*. 2021. URL: https://www.kernel.org/doc/html/latest/admin-guide/mm (visited on 01/11/2021).

[LLV20a] LLVM. *LLVM Compiler Intrastructure*. http://llvm.org/. 2020. (Visited on 10/01/2020).

[LLV20b] LLVM. *LLVM inline assembly constrains*. 2020. URL: https://llvm.org/docs/LangRef.html#inline-asm-constraint-string (visited on 01/11/2021).

[LLV21a] LLVM. *LLVM Loop unroll optimization pass*. 2021. URL: https://llvm.org/doxygen/LoopUnroll_8cpp_source.html (visited on 01/11/2021).

[LLV21b] LLVM. *LLVM Passes*. 2021. URL: https://llvm.org/docs/Passes.html (visited on 01/11/2021).

[LU+04] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. "Design and implementation of a lightweight dynamic optimization system". In: *Journal of Instruction-Level Parallelism* 6.4 (2004), pp. 332–341.

[LUK+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Acm sigplan notices* 40.6 (2005), pp. 190–200.

[MAS87] Henry Massalin. "Superoptimizer: a look at the smallest program". In: *ACM SIGARCH Computer Architecture News* 15.5 (1987), pp. 122–126.

[MPE+16] Paschalis Mpeis, Pavlos Petoumenos, and Hugh Leather. "Iterative compilation on mobile devices". In: *The 6th International Workshop on Adaptive Self-tuning Computing System (ADAPT), HiPEAC* (2016).

[MPE+21] Paschalis Mpeis, Pavlos Petoumenos, Kim Hazelwood, and Hugh Leather. "Developer and user-transparent compiler optimization for interactive applications". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2021, pp. 268–281.

[MPE21] Paschalis Mpeis. *Experimental LLVM backend for Android applications (HGraph IR-to-IR translation)*. 2021. URL: https://github.com/paschalis/android-llvm/ (visited on 01/11/2021).

[NIH20] NIH. *Sieve*. 2020. URL: https://imagej.nih.gov/nih-image/java/benchmarks/sieve.html (visited on 01/11/2021).

[OGI+17] William F Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. "Minimizing the cost of iterative compilation with active learning". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2017, pp. 245–256.

[ORA20] Oracle. *Java programming language*. https://www.java.com/. 2020. (Visited on 10/01/2020).

[ORS+05] Alessandro Orso and Bryan Kennedy. "Selective capture and replay of program executions". In: *WODA* 30.4 (July 2005), p. 1. URL: http://portal.acm.org/citation.cfm?doid=1082983.1083251%20http://dl.acm.org/citation.cfm?id=1083251.

[OSM+02] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. "The design and implementation of Zap: A system for migrating computing environments". In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 361–376.

[ÖST20] Peter Österlund. *DroidFish Chess*. 2020. URL: https://play.google.com/store/apps/details?id=org.petero.droidfish&hl=en&gl=US (visited on 01/11/2021).

[PAR+04] David Parello, Alchemy Group, and H P France. "Toward a Systematic, Pragmatic and Architecture-Aware Program Optimization Process for Complex Processors". In: 00.c (2004).

[PAR+11] Eunjung Park, Sameer Kulkarni, and John Cavazos. “An evaluation of different modeling techniques for iterative compilation”. In: *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*. ACM. 2011, pp. 65–74.

[PAT+10] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. “Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs”. In: *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 2010, pp. 2–11.

[PAT+21] Harish Patil, Alexander Isaev, Wim Heirman, Alen Sabu, Ali Hajiabadi, and Trevor E Carlson. “ELFies: Executable Region Checkpoints for Performance Analysis and Simulation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 126–136.

[PLA+94] James S Plank, Micah Beck, Gerry Kingsley, and Kai Li. *Libckpt: Transparent Checkpointing under Unix*. Computer Science Department, 1994, pp. 213–223.

[POZ+04] Roldan Pozo and Bruce Miller. *SciMark 2.0*. 2004. URL: `https://math.nist.gov/scimark2/` (visited on 01/11/2021).

[PUR+13] Suresh Purini and Lakshya Jain. “Finding good optimization sequences covering program space”. In: *ACM Transactions on Architecture and Code ...* 9.4 (2013). URL: `http://dl.acm.org/citation.cfm?id=2400715`.

[RON+99] Michiel Ronsse and Koen De Bosschere. “RecPlay: A fully integrated practical record/replay system”. In: *ACM Transactions on Computer Systems (TOCS)* 17.2 (1999), pp. 133–152.

[SAI05] Yasushi Saito. “Jockey: a user-space library for record-replay debugging”. In: *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. 2005, pp. 69–76.

[SAM16] Diogo Nunes Sampaio. “Profile guided hybrid compilation”. PhD thesis. Université Grenoble Alpes, 2016.

[SCO20] Scoutant. *Blokish*. 2020. URL: `https://f-droid.org/en/packages/org.scoutant.blokish` (visited on 01/11/2021).

[SEE+14] Volker Seeker, Pavlos Petoumenos, Hugh Leather, and Björn Franke. "Measuring qoe of interactive workloads and characterising frequency governors on mobile devices". In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2014, pp. 61–70.

[SOR20] Juanky Soriano. *MaterialLife*. 2020. URL: `https://play.google.com/store/apps/details?id=com.juankysoriano.materiallife&hl=en&gl=US` (visited on 01/11/2021).

[STE+00] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. *jRapture: A capture/replay tool for observation-based testing*. Vol. 25. 5. ACM, 2000.

[SUN+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. "Practical virtual method call resolution for Java". In: *ACM SIGPLAN Notices* 35.10 (2000), pp. 264–280.

[TIA+08] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. "Copy or discard execution model for speculative parallelization on multicores". In: *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2008, pp. 330–341.

[TOP20] topjohnwu. *Magisk*. 2020. URL: `https://github.com/topjohnwu/Magisk/releases` (visited on 01/11/2021).

[UHS+08] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. "Exploiting hardware performance counters". In: *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE. 2008, pp. 59–67.

[VAL20] Valgrind. *Callgrind: records call history among functions*. 2020. URL: `https://valgrind.org/docs/manual/cl-manual.html` (visited on 01/11/2021).

[VEL20a] Velbazhd. *Brainstonz*. 2020. URL: `https://f-droid.org/en/packages/eu.veldsoft.brainstonz` (visited on 01/11/2021).

[VEL20b] Velbazhd. *ColorOverflow*. 2020. URL: `https://f-droid.org/en/packages/eu.veldsoft.colors.overflow` (visited on 01/11/2021).

[VEL20c] Velbazhd. *PokerOdds (Vitosha)*. 2020. URL: `https://f-droid.org/en/packages/eu.veldsoft.vitosha.poker.odds` (visited on 01/11/2021).

[VEL20d] Velbazhd. *Svarka Calculator*. 2020. URL: https://f-droid.org/en/packages/eu.veldsoft.svarka.odds.calculator (visited on 01/11/2021).

[VIR20] Virtuozzo. *Checkpoint and Restore In Userspace*. 2020. URL: https://www.criu.org (visited on 11/01/2020).

[VOS+01] Michael J Voss and Rudolf Eigemann. "High-level adaptive program optimization with ADAPT". In: *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*. 2001, pp. 93–102.

[WAN+14] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. "Drdebug: Deterministic replay based cyclic debugging with dynamic slicing". In: *Proceedings of annual IEEE/ACM international symposium on code generation and optimization*. 2014, pp. 98–108.

[WEI84] Reinhold P Weicker. "Dhrystone: a synthetic systems programming benchmark". In: *Communications of the ACM* 27.10 (1984), pp. 1013–1030.

[WHA00] John Whaley. "A portable sampling-based profiler for Java virtual machines". In: *Proceedings of the ACM 2000 conference on Java Grande*. 2000, pp. 78–87.

[XIA+12] Richard Xia, Tayfun Elmas, Shoaib Ashraf Kamil, Armando Fox, and Koushik Sen. "Multi-level Debugging for Multi-stage, Parallelizing Compilers". In: *EECS Department, University of California, Berkeley, Tech. Rep* (2012).

[XU+07a] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. "Efficient checkpointing of java software using context-sensitive capture and replay". In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07* (2007), p. 85. URL: http://portal.acm.org/citation.cfm?doid=1287624.1287638.

[XU+07b] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. "Efficient checkpointing of java software using context-sensitive capture and replay". In: *Proceedings of the the 6th joint meeting of the European software*

engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. ACM. 2007, pp. 85–94.

[ZHA+06] Xiangyu Zhang, Sriraman Tallam, and Rajiv Gupta. “Dynamic slicing long running programs through execution fast forwarding”. In: *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. 2006, pp. 81–91.

[ZHO+01] Hua Zhong and Jason Nieh. “CRAK : Linux Checkpoint / Restart As a Kernel Module”. In: (2001), pp. 1–18.