Developer and User-Transparent Compiler Optimization for Interactive Applications

Citation for published version:
Mpeis, P, Petoumenos, P, Hazelwood, K & Leather, H 2021, Developer and User-Transparent Compiler Optimization for Interactive Applications. in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21).* Association for Computing Machinery (ACM), pp. 268-281, 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 20/06/21. https://doi.org/10.1145/3453483.3454043

Digital Object Identifier (DOI):
10.1145/3453483.3454043

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)

Developer and User-Transparent Compiler Optimization for Interactive Applications

Paschalis Mpeis
University of Edinburgh
Edinburgh, UK
p.mpeis@ed.ac.uk

Pavlos Petoumenos
University of Manchester
Manchester, UK
pavlos.petoumenos@manchester.ac.uk

Kim Hazelwood
Facebook AI Research
Menlo Park, CA, USA
kimhazelwood@fb.com

Hugh Leather
Facebook AI Research
Menlo Park, CA, USA
hleather@fb.com

Abstract

Traditional offline optimization frameworks rely on representative hardware, software, and inputs to compare different optimization decisions on. With application-specific optimization for mobile systems though, the idea of a representative testbench is unrealistic while creating offline inputs is non-trivial. Online approaches partially overcome these problems but they might expose users to suboptimal or even erroneously optimized code. As a result, our mobile code is poorly optimized and this results in wasted performance, wasted energy, and user frustration.

In this paper, we introduce a novel compiler optimization approach designed for mobile applications. It requires no developer effort, it tunes applications for the user's device and usage patterns, and has no negative impact on the user experience. It is based on a lightweight capture and replay mechanism. In its online stage, it captures the state accessed by any targeted code region. By repurposing existing OS capabilities, it keeps the overhead low. In its offline stage, it replays the code region but under different optimization decisions to enable sound comparisons of different optimizations under realistic conditions. Coupled with a search heuristic for the compiler optimization space, it allows us to discover optimization decisions that improve performance without testing these decisions directly on the user.

We implemented a prototype system in Android based on LLVM combined with a genetic search engine. We evaluated it on both benchmarks and real Android applications. Online captures are infrequent and each one introduces an overhead of less than 15ms on average. For this negligible effect on user experience, we achieve speedups of 44% on average over the Android compiler and 35% over LLVM -O3.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Source code generation**; **Runtime environments**.

Keywords: iterative compilation, capture, replay, interactive

ACM Reference Format:
Paschalis Mpeis, Pavlos Petoumenos, Kim Hazelwood, and Hugh Leather. 2021. Developer and User-Transparent Compiler Optimization for Interactive Applications. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453483.3454043

1 Introduction

More and more of our computing needs rely on smart mobile devices. With active users now more than 5.2 billions [8], 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 necessary for maintaining the user's Quality of Experience, supporting novel capabilities, and 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 [24]. As a result, immense amounts of performance and energy are wasted, impacting the smartphone user's experience.

A major hurdle with extracting performance from mobile systems is evaluating optimization decisions. Whether it is about comparing competing implementations, tuning compiler heuristics, or applying iterative compilation [30], we need to determine whether each choice will be beneficial or

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454043

not. We could evaluate all choices on the same representative mobile system with the same representative program input and then assume that choices that work well on that setup will be always beneficial. With a multitude of processor designs, system capabilities, software stacks, operating environments, and usage patterns [45], this is a hard assumption to make. In any case, creating any kind of program input that induces the same application behavior every time while causing no side-effects is a non-trivial endeavor for interactive applications. For example, the Telemetry framework [43] which enables deterministic performance testing for the Chromium browser is larger than many applications, containing 92k lines of Python code. This does not include the code required to actually define each input.

Evaluating optimization decisions online, on the target system while the user is interacting with the application, could be a solution if it did not introduce a whole new set of problems. Making confident decisions is hard when the execution context is constantly changing. The operating environment, the application state, and the input will be different each time the application is executed. Their effect on latency can easily be more important than the effect of any optimization decision. A very large number of evaluations for each decision might be required before we are confident about its merit [19]. Many of these decisions can easily be suboptimal, severely affecting the user experience. Statistical approaches can keep the negative impact from growing out of control at the cost of an even slower optimization process [12] but in a low-latency environment even infrequent slowdowns that are noticeable by the users are unacceptable [45]. In any case, when tested optimizations lead to erroneous results, something relatively common with optimizing compilers [16], no online approach is acceptable.

To make it easy to choose the right mobile optimizations, we need a way to evaluate them that combines the benefits of offline and online approaches: adapted to each device and each application, driven by actual user input but at the same time fast, unrestricted, and invisible to the user.

In this paper we propose a novel way for achieving this by driving offline optimization with real input collected online. We build our approach around a lightweight capture and replay mechanism. While the user is interacting with the device, we identify important code regions and capture the state accessed by them. Later, we are able to restore the saved state and re-execute the code region in the exact same way it was originally executed. By replaying this execution under different optimization decisions but the same environment, we can directly compare optimizations using the time it took to execute the region.

Traditional capture mechanisms [10, 40, 48, 50] are not well suited to this task, especially in terms of introduced latency. Capture happens online and may affect the user, so it is critical to keep its overhead low. With this aim, we developed a new capture approach that leverages the kernel Virtual Memory subsystem to automate and accelerate the capture. We use memory protection to monitor, with little overhead, which pages are accessed during the execution of the targeted region. Future replays will only need to access these pages, a small fraction of the application memory space. For all accessed pages, we need their original state before they are modified by the application. Instead of copying them preemptively in user space [10], we use the Copy-on-Write mechanism to do it on-demand in kernel space. When the region's execution ends, we only have to store the original version of each accessed page in the phone's storage.

On top of the capture and replay mechanism, we build our novel iterative compilation system that optimizes mobile applications for the actual environment and context they are being used in. We use a genetic search engine to explore the space of compiler optimizations. We evaluate each optimization decision by applying it on the targeted code and replaying the captured execution with the new binary. We rank and select optimizations based on the time each replay took, apply genetic operations, and continue the process with another generation. We select the best performing optimization over the whole search and we apply it on the online version of the application, the one the user interacts with.

All replays happen offline, when the phone is idle and charged. That gives us tighter control of the execution environment, reducing the measurement noise. Since all replays process the same state under the same environment, we are highly confident in the ranking of the evaluated optimizations. Since this is the same device the user interacts with and these inputs were produced during the normal operation of the application, we are highly confident that optimizations performing well during replay will perform well online too.

This framework is compiler-agnostic: any compiler that can process Dalvik code and generate code for Android devices can be plugged in with minimal effort. Currently there is only one such toolchain, the default Android one, and it has a very limited optimization space with 18 distinct optimizations. We overcame this limitation by creating a new Android backend based on LLVM which offers a far more expansive and interesting optimization space. We contribute to the research community by making it publicly available[1].

We have evaluated our approach on a set of Android applications, both benchmarks and interactive ones. Our online capture mechanism runs a handful of times per day and is imperceptible to the user, introducing an overhead of less than 15ms on average. Powered by the capture and replay mechanism, our iterative compilation approach searches for good optimizations without exposing the user to suboptimal and erroneous binary versions. The final selected optimizations achieve on average a 44% whole program speedup over the Android compiler and 35% over LLVM `-O3`, significantly improving the user experience and device autonomy.

[1] Android LLVM backend sources: github.com/paschalis/android-llvm

The main contributions of this paper are:

- a lightweight code region capture mechanism for Android with little runtime and storage overheads,
- a replay-based evaluation mechanism of different code transformations driven by actual usage patterns,
- a novel prototype iterative compilation system which optimizes programs for specific devices, without any developer effort or impact on the user experience.

This paper is organized as follows. Section 2 describes the motivation behind our proposed approach. It is followed by Section 3 that describes our user-transparent iterative compilation framework. In Section 4 we describe the experimental setup, followed by Section 5 that presents our results. In Section 6 we discuss related work and in Section 7 we offer some concluding remarks.

2 Motivation

Iterative compilation [30] is a compiler optimization technique that can outperform a compiler's optimization presets. It searches for different combinations of code transformations and transformation parameters, evaluates their effect on performance, and at the end keeps the best performing binary. Multiple approaches exist to select optimization sequences: random, through genetic search, statistical models [41], or machine learned models [33]. Despite the significant performance benefits of iterative compilation, it has not been applied in a general way on mobile systems. This section explores why.

Offline Compiler Optimization. Regardless of how we optimize an application offline, we rely on some kind of a representative evaluation system (hardware and software) and a deterministic and representative input. By repeatedly executing the application with a deterministic input under different compiler decisions, we can directly compare decisions and choose the best one. With a representative system and inputs, we can be confident that our choice will work well online too.

This simple methodology has proved hard to adapt to mobile systems. There is no such thing as a representative system where we can optimize the application once for every other system. We need to optimize for each system individually. Even then, it is hard to create inputs without explicit support from the application. Mobile applications tend to have complex inputs, including user events, sensors, configuration files, system state, and network data. Packaging all these for offline usage 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 most mobile developers to put the effort required to support offline performance evaluation is unrealistic.

Online Optimization is Risky. An alternative approach is to evaluate optimization decisions online on the program version the user interacts with. This removes the problem of identifying representative systems and inputs: they are representative by definition. While a workable solution in some cases, online optimization creates a new set of problems.

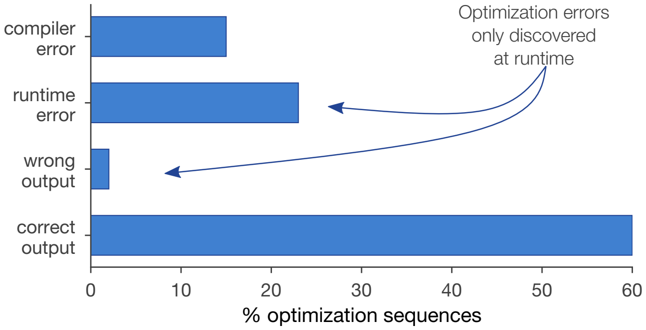

Figure 1. Compilation outcome with LLVM using 100 randomly generated optimization sequences for the `FFT` kernel from *Scimark*. 25% of the sequences result in a binary that does not behave as expected at runtime. In an online optimization system, this would directly affect the user experience and might lead to data corruption.

The first is that there is no hard guarantee that compiler optimizations will not introduce errors. Figure 1 shows what happens when we randomly choose LLVM optimization passes to apply on a benchmark, `FFT` from *Scimark* [42]. Only 60% of the optimization sequences lead to a binary that behaves correctly. 15% cause the compiler to crash or timeout but this is a manageable problem. What is not manageable is the other 25% that leads to compilation errors which only become apparent at runtime, either with a program crash, a program timeout, or a wrong output. In offline optimization such optimizations are just rejected with no other side-effect. But in an online setup, broken optimizations are visible to the user and affect the user experience. Even worse, silent errors that change the program behavior can result in corrupted application data, either local or remote, with long-term consequences for the user. This is an unacceptable risk.

Online Optimization Affects the User Experience. Broken optimizations are just one part of the wider problem of our search being visible to the user. Even for compiler optimizations that lead to valid binaries, there is a good chance that the binary will be noticeably slower than normal. Figure 2 shows the performance of 50 correct `FFT` binaries generated by applying random sequences of LLVM optimizations. Performance is relative to that achieved 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 choices improve performance, the user will still have experienced unacceptable levels of slowdown. As we will show in Section 5, 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.

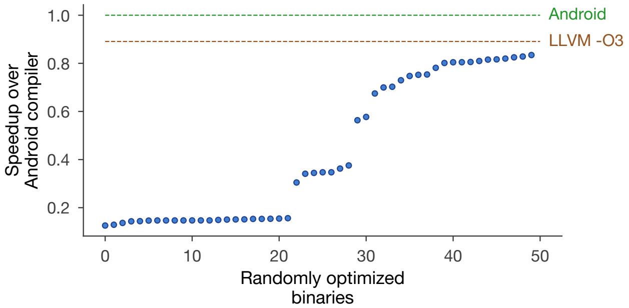

Figure 2. Speedup over the Android compiler for 50 randomly generated LLVM optimization sequences applied on the FFT kernel. Sequences that break the compiler or the application are discarded. All of them slow down the application relative to both the Android compiler and LLVM -O3, up to 8x slower. Evaluating these optimization sequences online would have an unacceptable impact on user experience.

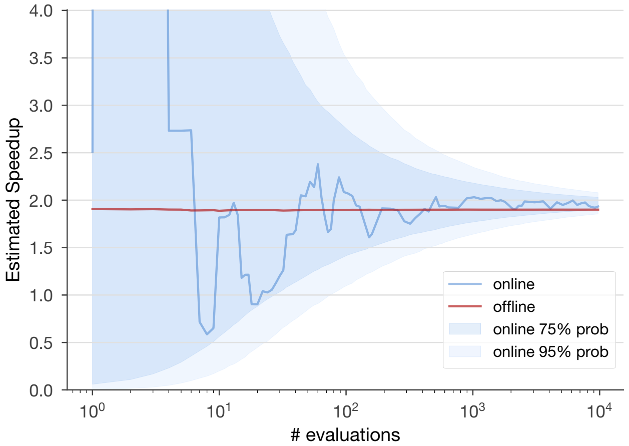

Figure 3. Estimation of the speedup of LLVM -O1 over -O0 for the FFT benchmark as the number of evaluations increases. Online search performs each evaluation with a different randomly selected input between FFT_SIZE and FFT_SIZE_LARGE. Offline search always uses the largest input. Lines represent speedup estimation for a single sequence of evaluations. Areas represent 75% and 95% bootstrapped confidence intervals. Online approaches require a much larger number of evaluations to estimate speedup.

Online Optimization is Slow. Even if we managed to work around these limitations, a fundamental problem remains: we have no control over the context in which optimizations are being evaluated, especially input. We have to use whatever input happens to be fed to the program. This is not always a problem. But in the general case, we cannot directly compare optimizations that have been evaluated on different inputs that take different amounts of time to process. The only safe way to do it is to evaluate each optimization a large number of times. If all optimizations are evaluated on a similar sample of inputs, direct comparisons of average execution times will be statistically meaningful. The problem is this can be a very lengthy process.

Figure 3 shows such an approach where we try to estimate the speedup of LLVM -O1 over -O0 for FFT with an increasing number of evaluations each with an input drawn from a uniform distribution. The evolution of a single speedup estimation is presented as a line. Areas are ranges of likely outcomes. While -O1 is almost 2x better than -O0, our online estimation varies wildly, from almost 2x slowdown to 8x speedup. It is not until the 22nd evaluation when we decide that -O1 is better than -O0. It takes another 20 evaluations for the estimation to start stabilizing. This behavior is not an outlier. The 75% probability area shows a similar evolution: 25 evaluations to decide that -O1 is better and more than 1000 evaluations to have less than 10% uncertainty in the speedup estimation. In contrast, there is very little variation with offline evaluation. Even a single measurement might be enough (though not statistically safe).

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 our optimization decisions. This is very likely an underestimation. Other sources of experimental noise that we cannot control online, such as frequency scaling, thermal throttling, and resource contention affect our confidence. Skewed distributions of input processing times may reduce our confidence even further. Seen in the context of an iterative compilation system like the one in Section 5, this would translate into millions of 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.

Beyond Online and Offline Optimization. What we ideally need is something that combines the best of both worlds. 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.

3 Replay-based Offline Optimization

In this section we describe how we aggressively optimize Android applications, with an offline optimization search driven by user-transparent input captures, taken online, requiring no developer effort. Section 3.1 describes how we detect *hot regions* of Android applications that are worth optimizing. Section 3.2 introduces our lightweight online captures, followed by Section 3.3 which describes our replay mechanism. Section 3.4 explains how we extract information from captures for optimization and automatic code correctness verification. In Section 3.5, we detail our LLVM backend implementation for Android that enables us to use LLVM to optimize Android code. It is followed by our genetic search engine which explores the space of compiler optimization decisions in Section 3.6. Finally, we show how we integrate all these components to build an iterative compilation system that is transparent to the user and the developer but still manages to produce highly optimized program binaries.

3.1 Hot Regions

Interactive applications are composed of multiple asynchronous tasks, some of them multi-threaded, interacting with each other and I/O. Out of this, we need to detect code regions that are both accurately replayable and worth optimizing. The former translates into their behavior being fully determined by their memory state and having no observable effects outside their memory space. The latter means that they take most of the execution time and are compilable by our LLVM backend.

```
input: methods from a sample-based profile
output: method with biggest compilable region
def estimateRegionRuntime(method):
    if not replayable(method) then
        return $-\infty$
    end
    sum $\leftarrow$ 0
    compilable $\leftarrow$ compilableRegion(method)
    foreach $c \in$ compilable do
        sum += runtimeExclusive($c$)
    end
    return sum
def compilableRegion(method):
    def inner(m, l):
        if $m \notin l$ and IsCompilable(m) then
            add(l, m)
            foreach $c \in$ callees(m) do
                inner(c, l)
            end
        end
    list $\leftarrow \emptyset$
    inner(method, list)
    return list
sort(methods, estimateRegionRuntime)
return methods.first
```

Algorithm 1: Detecting an application's hot region.

Through static bytecode analysis, we first identify methods that might interfere with replaying: methods with I/O and methods with sources of non-determinism. Replaying I/O without special infrastructure is either impossible or leads to inconsistent replays. Writing information out is even worse since it can corrupt the application's permanent state. Emulating such I/O is doable but we judged that the overhead of doing so is prohibitive for latency sensitive interactive applications. The sources of non-determinism that we excluded were calls to clocks and pseudo-random number generators. Additionally, we aggressively blocklist almost all Java Native Interface (JNI) calls, regardless of whether they can be accurately replayed or not. Determining that their low level code does not do I/O and it is deterministic requires a significant amount of engineering work that we leave as future work. The only JNI calls we do not blocklist are the ones we will replace with intrinsics, as described in Section 3.5. Finally, we blocklist exceptions. They are the only kind of code that is allowed to access previous stack frames in Java. Optimizations that alter the stack layout can interfere with code that accesses previous stack frames, since that code might assume a different layout than the one in the stack.

Then, we detect regions worth optimizing, which we call *hot regions*, as outlined by Algorithm 1. We use Android's sample-based profiler to find the most frequently executed methods, with a sampling period of 1ms. The profiler is lightweight and, being executed infrequently, it has little effect on performance. We ignore any methods that the Android compiler cannot process or we cannot replay, as discussed above. Finally, we estimate the cumulative time spent in each method and its compilable callees. The most significant becomes our *hot region* that our capture mechanism targets.

3.2 Online Captures

We design our approach upon the idea of capturing the application's behavior while it is being used and accurately replicating that behavior later offline under different optimization decisions. Given that we exclude methods with I/O and non-deterministic behavior, capturing the application's behavior is the same as capturing its memory state just before the hot region. This could be as simple as saving all processor registers and copying every page in the memory space of the app. Our approach aims to be more efficient than that. It uses the kernel's memory protection subsystem to capture only what is needed with the least amount of overhead. With a fork right before the start of the hot region, we create a full copy of the virtual memory space of the app. The Copy-on-Write mechanism will duplicate modified pages leaving our copy of the memory space in its original state. By read-protecting the app's memory pages, we offload identifying accessed pages to the kernel. When the hot region's execution is complete, we only need to store the original state of any pages marked read. This process is transparent both to the user and developer, without any kernel modifications. In more detail, our capture mechanism, shown in Figure 4, is composed of the following steps:

1) Capture initiated. We add code to the entry point of a hot region to check whether a capture should occur. We only require a single capture to drive optimizations and there are plenty of opportunities to get one, so we generally set the frequency of capturing to a low value. This is adjustable both per region and per application run. If a Garbage Collector (GC) run is imminent, we postpone any scheduled capture as the GC will touch memory not normally used by the *hot region*, which might increase snapshot sizes.

2) Forking the child. To keep a pristine copy of the parent's memory we `fork` a child process. Initially both processes point to the same set of physical pages. When the

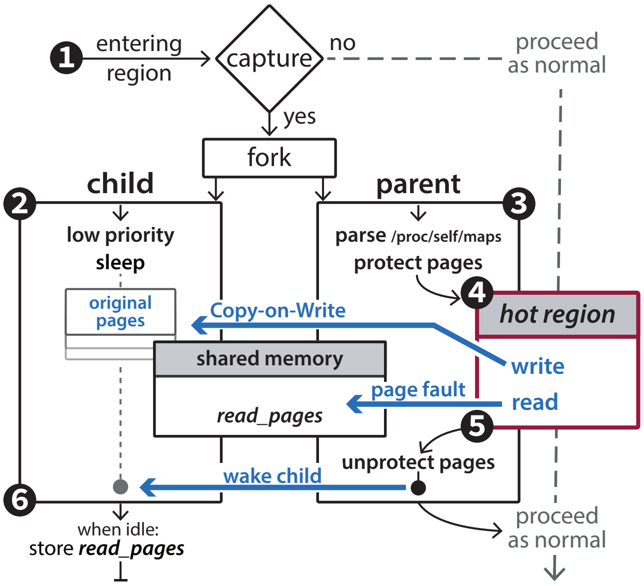

Figure 4. Our capture mechanism stores pages accessed by the *hot region* to enable replaying at any future point in time. With `fork` and Copy-on-Write we keep a copy of the original state in a child process. With read protection and fault-handling we identify the set of read pages.

parent modifies a page, the kernel's efficient Copy-on-Write automatically creates a copy of the page's original state for the child. This is faster than taking a full snapshot of the memory state before the hot region runs [48] or manually copying each page just before it is first used [10]. We then minimize the child's priority and send it to sleep.

3) Memory pages protection. Saving all pages of the process to disk is inefficient. With the help of the kernel, we identify and store only the pages the *hot region* uses instead. We get the full list of pages for the parent by parsing the `/proc/self/maps` file. We read-protect most of them to cause deliberate page faults on read attempts. We also install a page fault handler that performs two actions. First it stores the offending page address in a memory buffer shared with the child. Then it restores access permissions for that page so that any future reads will not cause a fault.

4) Hot region execution. The parent now executes the *hot region* as normal. Other than fault handling when pages are first read and Copy-on-Write when they are first written, there is no further overhead. Since most applications display high levels of locality [17], we expect the number of these events to be low.

5) Ending the hot region. When the parent has finished executing the *hot region*, it wakes up the child, restores access permission to any remaining read-protected pages, and uninstalls the fault handler. After this the parent continues executing as usual.

6) Saving the memory state. The child begins spooling out pages marked as read to disk. Since the child has the lowest priority possible, this will happen only when the system has unused processing and I/O capacity, causing no inconvenience to the user.

There is some additional state that we capture, mainly the architectural state of the processor. Also, for some pages we cannot use the read-protection mechanism to find out whether they are accessed. Read-protecting memory belonging to the Android runtime, its library dependencies, or GC auxiliary structures could crash the process. We choose to always store these pages to disk.

A significant chunk of the used memory pages is not process specific. It contains immutable objects of the Android runtime that only change across device boots, which we capture only once per boot. It also contains several memory-mapped system files that only change when the system is updated, like any executable code pages. Those are never captured. Instead, we log the relevant file paths and offsets, and we memory-map them directly during replays.

3.3 Replaying Android Code Regions

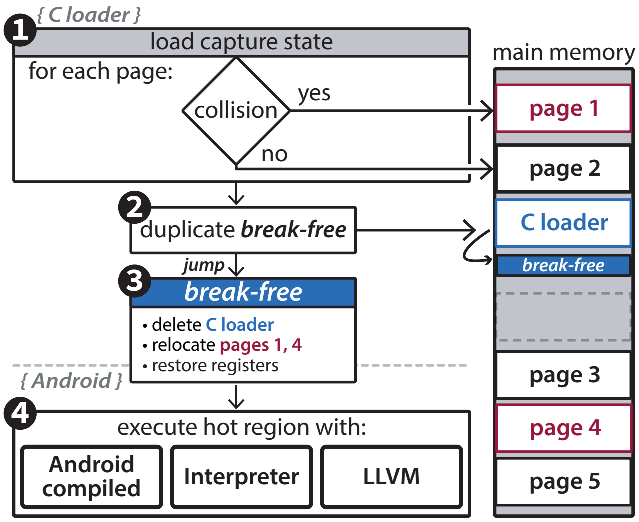

Figure 5. Our replay starts with a `C` loader program that transforms itself into a partial Android process whose state is the previously captured one. The loader can swap different code versions for executing the *hot region*. It works alongside `ASLR`, a memory-shuffling security mechanism.

With all the state used by the *hot region* captured, the next step is to use that state to recreate offline the behavior the 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 [48]. 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 and the globally visible results of the execution should be the same.

In practice, reloading the captured state is a bit more complicated than that. The whole operation, called a *replay*, is shown in Figure 5 and is composed of the following steps:

1) Load captured state. With a loader program, written in C, we load the captured pages into memory. We place most of them directly in the virtual addresses they originally had. Some may map to locations already occupied by the loader's pages. These are collisions and we handle them by placing colliding pages in temporary locations. We choose these locations so that we do not introduce more collisions. This means consulting the captured memory layout and choosing locations not used by it. This approach does not make any assumptions about the address layout of the loader, so it can work in the presence of ASLR, a security mechanism that randomly shuffles processes' memory. We also read the captured architectural state into a temporary location.

2) Duplicating the `break-free` method. To resolve collisions caused by the loader, we must first discard all of its state before we can relocate captured pages. Simply put, the loader must both delete itself and keep setting up the replay. This cannot be done directly, so we duplicate the binary code of a special position-independent method, named `break-free`, to a non-colliding area.

3) Becoming an Android process. We then jump to `break-free` which becomes self-contained by switching to its own stack and data segments. We can now complete the transition to a partial Android process by first releasing the original loader pages and then moving colliding pages to their final locations. Finally, we complete the transition by restoring the architectural state of the processor.

4) Choose and execute code. The final step is to choose the version of the *hot region* that we will execute. We support three different code types. The first is replaying the original Android code for our evaluation baseline. The second is calling the Android interpreter. We use the interpreter to extract capture information for correctness verification and optimization, as we describe in Section 3.4. The final one is calling a new optimized binary in which case the binary is also loaded into memory before we jump into its entry point.

3.4 Interpreted Replay to Verify and Optimize

An added benefit of being able to replay a captured execution offline is that we can collect information about the execution that would be too costly to collect online. In our prototype, we use this capability to record the externally visible behavior of the hot region and the runtime types of virtual calls. In this section we describe how we use an interpreted replay to extract this information. While this is slow, it happens offline without affecting in any way the user experience.

As discussed in Section 2, there is no guarantee that an optimization sequence will not cause the program to crash, freeze, or do something different than what it was supposed to do. Identifying the last case is critical because it might lead to silent data corruption. To that end, we use an interpreted replay of the hot region to build a *verification map*. The map stores key-value pairs of memory locations and values for the externally visible object fields or array elements that were modified, as well as the region return value. Together, these data represent the externally observable behavior of the region. After each iterative compilation replay, we verify correctness by comparing the memory of the process against the verification map. This does not require any manual effort or online instrumentation.

To improve the quality of our optimized code, we collect some profiling data through our interpreted replays. In particular, we record at each virtual or interface call-site the frequency histogram of actual dispatch types. With this information, we are able to reduce call overheads and aggressively inline them using *speculative devirtualization.* We implemented our own pass to that aim but it is not particularly novel, devirtualization is a well-known optimization. What is novel is the information that drives the pass. Finally, we also tune branch prediction on each speculated type according to the frequency histogram.

3.5 LLVM Backend

The only complete compiler toolchain for Dalvik code is the Android compiler one. It is designed to be safe rather than highly optimizing. It only applies a handful of safe optimizations [25] which are guaranteed to have only positive or no impact at all. It lacks more aggressive code transformations found in established compilers like LLVM [32]. The potential for better compiler optimization strategies is limited.

To overcome this limitation, we have developed a new Android compiler backend based on LLVM. The core of our implementation focuses on transforming `HGraph` [6] nodes, the Android compiler internal Intermediate Representation (IR), into LLVM bitcode. This new transformation pass is to our knowledge the only way to pass code from the Android compiler to LLVM. It represents a significant engineering effort with more than 25k lines of code.

We still rely on the Android compiler to apply some target-specific optimizations and generate the `HGraph`. After we transform it into LLVM bitcode, we use existing LLVM facilities to optimize and compile the code. Additionally, we implemented two Android-specific optimizations of our own.

The first is a GC optimization which runs after loop-restructuring passes. We do not always know at compile time when a loop might exit, so `HGraph` inserts a check call in each loop body to transfer control to the Android runtime and allow it to perform GC if needed. A single check per loop is enough but passes like loop unrolling do not have this knowledge so they unroll the whole loop body including the check. Our optimization pass removes these duplicate checks from the loop body. The second optimization replaces JNI calls that implement particular math library methods with LLVM IR implementations, either LLVM intrinsics or ones we implemented. This not only avoids the overhead of JNI calls but it also increases the amount of code we can optimize and compile, making the rest of our optimization passes more effective.

This backend is a work in progress. Despite that it can already produce better code than the default Android backend, as we will see in Section 5. With more engineering work to improve the backend, our optimization strategy will only become more successful. To give more researchers the option of producing highly optimized code for Android applications, we are releasing our backend as open source software.

3.6 Searching Compilers' Space with a GA

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 and subsequently applied passes [13]. 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. We need a way to quickly explore the rewarding areas of this complex optimization space.

Our approach can work with different search strategies but the one implemented was a Genetic Algorithm (GA). It is a well established strategy that has worked really well in the past for similar problems [14, 20, 31, 34]. Genomes encode the sequence of passes, the parameters, and the flags. They may vary in length to account for different numbers of optimization passes. We employ three different mate selection pipelines: elites only, fittest only, and tournament selection. Once mates are selected, we cross them over with a single random point, ensuring that the resulting genome length is higher than a predefined minimum. We have several mutation operators for different types of genes: enable/disable a pass, modify a parameter, or introduce new passes. The GA 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, we perform a hill-climbing step to reach the local maximum.

Our fitness function is primarily performance, measured by replaying the hot region (see Section 3.3), but if the performance of two binaries is sufficiently close, then we prefer the smaller binary. We provide the GA parameters in Section 4.

3.7 Iterative Compilation through Replaying

Our goal is to optimize applications by comparing the effect of different optimization decisions on performance. For sound comparisons, the application needs to perform the same work each time we evaluate an optimization and that work should be representative of actual usage. Since performance evaluations can affect user experience, they should be performed only offline and when the device is idle. Additionally, we should cope with environmental noise and optimize per application and device. In this section we outline our solution that brings together the individual components from previous sections.

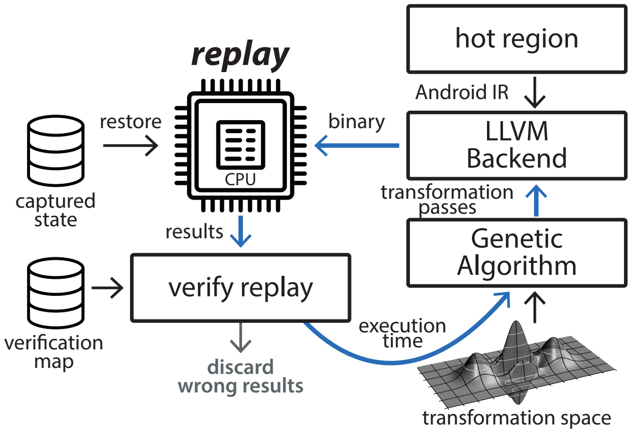

Figure 6. The Replay-based Iterative Compilation main loop (highlighted). The GA (lower right) selects interesting optimization decisions from the LLVM transformation space. It passes them to the LLVM backend which applies them on the hot region. We replay the captured execution using the newly optimized binary multiple times and we report the timings back to the GA. Any binary producing wrong results is discarded. This loop continues until the GA converges.

In Figure 6 we show a high level overview of our approach. Once we have identified *hot regions* and transparently captured real user inputs to them, we can apply offline iterative compilation. With a GA we quickly search the compiler's space by constructing optimization sequences. We pass them to the LLVM `opt` and `llc` tools to first transform and then compile the region's IR. Then we replay the captured execution with the generated binaries to evaluate their performance. Finally, we construct the next generation and continue the process. Each replay has the same input since we restore the same captured state. We ensure soundness in our comparisons by performing multiple replays per binary and using statistical methodologies, as described in Section 4. We also fix the processor frequency in the otherwise idle device environment to reduce random performance variation. Using a previously generated verification map, we automatically discard any transformations that lead to externally observable wrong behavior.

4 Experimental Setup

We have implemented a prototype system and evaluated our approach with a set of experiments on a recent *Google Pixel 4* running *Android 10*. Its processor unit is a *Qualcomm SDM855 Snapdragon 855*. It consists of eight *Kryo 485* cores each configured with a different maximum clock frequency, from 1.78 GHz up to 2.84 GHz. To reduce measurement noise

Table 1. Android applications used in our experiments.

Type	Name	Description
Scimark [42]	FFT	Fast Fourrier Transform
	SOR	Jacobi Successive Over-relaxation
	MonteCarlo	Estimates π value
	Sparse matmult	Indirection and addressing
	LU	Linear algebra kernels
Art	Sieve [39]	Lists prime numbers
	BubbleSort [3]	Simple sorting algorithm
	SelectionSort [3]	Simple sorting algorithm
	Linpack [18]	Numerical linear algebra
	Fibonacci.iter [27]	Fibonacci sequence iterative
	Fibonacci.recv [27]	Fibonacci sequence recursive
	Dhrystone [49]	Representative general CPU performance
Interactive	MaterialLife [46]	Game of life
	4inaRow [7]	Puzzle Game
	DroidFish [51]	Chess Game
	ColorOverflow [36]	Strategic Game
	Brainstonz [35]	Board Game
	Blokish [44]	Board Game
	Svarka Calculator [38]	Generates odds for a card game
	Reversi Android [21]	Board Game
	Poker Odds (Vitosha) [37]	Statistical analysis for poker cards

during replays we keep all cores online and fix their frequency to the maximum. We only replay while the phone is fully charged, so the increased power consumption does not affect device autonomy. At any other time, the OS has full control of the cores' operating states and frequencies.

Our compiler toolchain uses the Android 10 compiler to generate `HGraph` nodes, the LLVM backend from Section 3.5 to transform it into LLVM IR, and LLVM 10 `opt` and `llc` to optimize and compile the code respectively. For `opt`, we consider a total of 197 passes and 710 parameters and flags. For `llc`, we control 90 CPU-specific and 569 general options.

The GA searches through this enormous space over 11 generations, the first randomly generated, the other 10 driven by genetic search. Each generation has 50 genomes. In the first one, we may try up to three times to replace each genome that leads to worse performance than both LLVM `-O3` and Android. This biases the algorithm towards the more profitable areas of the transformation space. In the first generation, we also remove redundant passes to keep the genomes short. The probability that we mutate a genome and the mutation probability per gene are both 5%. The maximum number of identical binaries before halting the algorithm is 100. Each tournament selection round considers seven candidates with a 90% probability.

For our experiments we use Android applications that fall into three types, as listed in Table 1. The first is the *Scimark* benchmark suite. The second, named *Art*, contains benchmarks that have been used in the past by Google or 3rd parties for the evaluation of the Android compiler. The third, named *Interactive*, contains 9 regular Android applications.

We collect execution times for the final reported speedups outside the replay environment to guarantee that our results are not an artifact of the replay environment. We use the best performing binary discovered by the GA earlier using replay but we otherwise execute the application interactively. We disable any random initialization so that applications start in approximately the same state throughout the experiments and perform similar work. 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, we try to follow the logic of each application to bring its state to a similar point each time, including manually interacting with the application.

For the execution time measurements of interactive applications deciding the start and the end point of the measurement is a balancing act. We want a fair and representative measurement which includes the code surrounding the *hot region*, i.e. code we have not optimized. At the same time we want all time measurements to 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, we measure a certain number of iterations of the conceptual main loop of the application. This number is different for each application and was selected empirically. For games this conceptual loop was one player round and it generally ranged from 0.7s to 3s.

Even for a small number of applications, this was a very tedious process. It only reinforced our conviction that deterministic and repeatable evaluations without replaying are non-trivial and require significant effort.

To ensure the statistical soundness of our results we use rigorous methodologies. During search, each transformation is evaluated 10 times through replay. We perform outlier removal on the collected runtime measurements using *median absolute deviation*. Then, we determine the relative merit of two sets of transformations with a *two-side student's t-test*. Reported speedups are also based on 10 evaluations per transformation but without outlier removal. Finally, our plots include 95% confidence intervals where applicable.

5 Results and Analysis

To evaluate our approach we performed four sets of experiments. The first shows our performance gains on the apps of Table 1 and demonstrates the potential of our approach. The second shows why we need an intelligent search of the optimization space and why we should not do it online. The third establishes that the online overhead of captures is low, making them transparent to the user. The final set shows that storage overheads are similarly low and that our approach should be applicable even on low-end devices.

5.1 Speeding-up Android Applications

Our approach readily outperforms the Android compiler with aggressive optimization through offline, replay-based iterative compilation. In this section we show our performance gains and the runtime code breakdown for 21 applications.

Figure 7 shows the speedups we observe outside the replay environment for the binaries selected by the GA. We effectively use two baselines. The default Android compiler, against which all other speedups are measured, represents the performance the user gets out of the box. The other one,

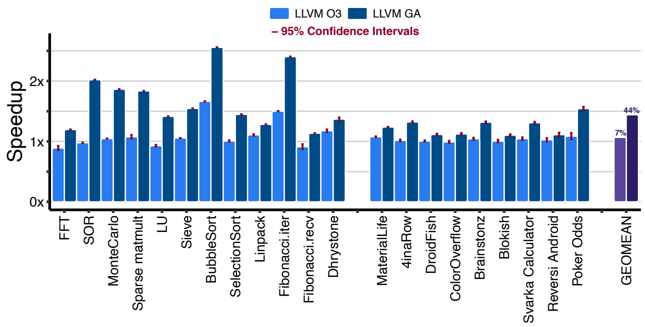

Figure 7. Speedup relative to the Android compiler for 12 benchmark (left) and 9 interactive applications. LLVM −O3 performance ranges from a 0.89x to a 1.66x. LLVM GA, which uses a replay-based iterative compilation, produces from 1.10x to 2.56x speedups and an average of 44%.

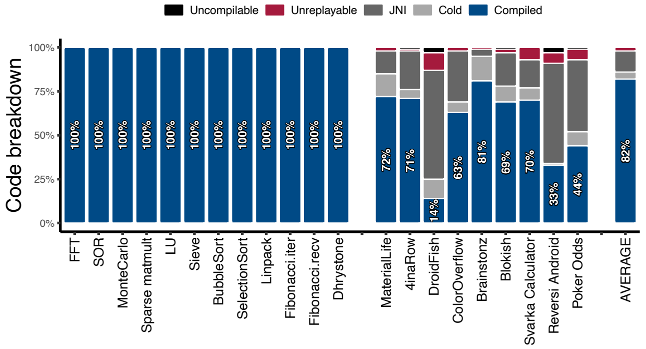

Figure 8. Runtime code breakdown, measured online with a sample-based profiler, for 12 benchmark (left) and 9 interactive applications (right). Compiled is code that we can optimize. Cold falls outside the *hot region*. *JNI* is C/C++ code. Unreplayable cannot be replayed. Uncompilable cannot be processed by the Android backend.

LLVM −O3, is an aggressive optimization setting for LLVM. It represents the best LLVM can do without any application-specific information.

Our optimization framework, shown as LLVM GA, clearly outperforms both Android and LLVM −O3 by a significant margin. The GA performs a relatively quick search: 11 generations at most, each one with 50 genomes. Despite visiting a tiny fraction of the optimization space, we improve performance over both baselines for all programs, achieving an impressive speedup average of 44% over the Android compiler and 35% over LLVM −O3. When considering benchmarks exclusively, speedup ranges from 1.14x for *Fibonacci.recv* to 2.56x for *Bubblesort*, which was the highest overall.

For interactive applications, we had similar results when considering the speedup of *hot regions* in isolation (see Figure 9). When we evaluate our approach on the wider regions that surround the code we optimized, speedups range from 10% to 54%. This is impressive given that only part of the code benefits from iterative compilation. Figure 8 shows how much of the program's execution we can compile and optimize. These breakdowns are collected during normal operation, outside the replay or the evaluation environment, while the user interacts with the application. On average only 57% is *Compiled* and it ranges from 14% to 81% depending on the application. The rest is *Cold* which is not executed enough times to be worth compiling, *JNI* which falls outside the scope of our toolchain, *Unreplayable* which is code we have deliberately chosen not to support in our captures (see Section 3.1) and it includes all code that performs I/O, or *Uncompilable* which the Android compiler, and therefore our backend, cannot process [5]. Most of the code we do not compile is *JNI* code, representing on average 29% of the interactive applications execution and as much as 62%. While the interactive applications naturally perform I/O, the amount of code that actually performs the calls is not as much as one would expect. *Unreplayable*, which includes other sources of non-determinism too, takes only a 4% on average. This is quite small when compared to *Compiled*, which is essentially the code that processes that I/O.

While LLVM −O3 improves performance as much as 66% and 7% on average, this is roughly only a sixth of the performance our GA unlocks. Even worse, there are cases where −O3 hurts performance versus the more conservative Android compiler. The most notable example is *FFT* where an increased amount of heap-related operations, e.g. checks for GC, slows down the program by 10%. The GA learns to solve this problem with loop unrolling combined with our post-loop optimization, while the −O3 heuristics decide not to unroll these loops. This shows clearly that using a powerful optimizing compiler is not enough on its own. We also need a sophisticated way to guide compiler optimization decisions.

The reported speedups in Figure 7 underestimate the potential of our approach, as we are held back by the limitations of our LLVM backend, which is a work in progress. The transformation from HGraph to LLVM bitcode is not as efficient as it can be, while many *JNI* math library calls could be easily replaced with LLVM intrinsics. Despite that we still manage to improve performance significantly, including for programs like DroidFish where we affect only a small part of the execution. With a more mature compiler toolchain, we would expect these gains to grow even further. In Section 7, we discuss how we will work towards that direction.

5.2 Using a GA for Offline Optimization Search

With a GA search over the vast space of LLVM optimization decisions, we are able to discover code transformations that significantly outperform the Android compiler. In this section, we illustrate this by visualizing how the best genomes evolve over time. We also show that GA, like any other self-adapting algorithm, will inevitably attempt to evaluate sub-optimal configurations during search. Any optimization approach that tests optimization online is simply not feasible.

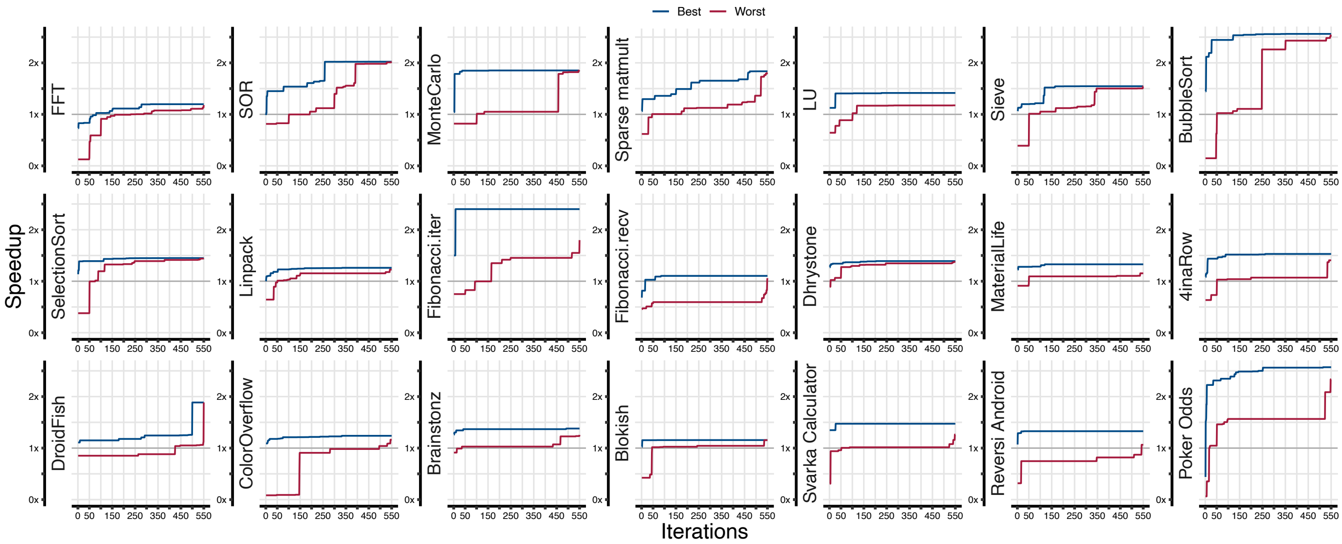

Figure 9. Range of speedups over the Android compiler for the LLVM transformation sequences selected by the GA. Speedups are estimated through replay for the *hot regions* only. The two lines represent the evolution of the best and worst genomes over time. Vertical grid lines indicate the change from one generation to the next. All programs benefit from the search. Program versions worse than the baseline are common across all programs, in some cases even after multiple generations.

Figure 9 shows the evolution of our 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 our applications require multiple generations to discover their best binary, while a few might have benefited by an even longer search, e.g. Sparse Matmult. Overall, the ability to search the optimization 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 they were evaluated online, they would have a dramatic effect on the user experience. For the whole first random generation of *FFT* and a significant portion of it for *Fibonacci.recv*, even the best genomes are worse than the baseline. This is not limited to the early stages of the search. Nine applications were still picking sub-optimal transformations even after three generations, while a few kept testing sub-optimal genomes to the very end. This does not take into consideration the even slower genomes that we evaluated but discarded in the first generation, as explained in Section 4. In any case, only a handful of sub-optimal evaluations would have been enough to degrade the user experience, rendering online approaches impractical.

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. Most applications have used CPU-specific optimizations like address, literal, arithmetic, or logical fusing, while more than half of the applications have used full or partial inlining, with or without custom thresholds. Other common optimizations include global value numbering, sub-expression elimination, and instruction/function/return merging.

5.3 Transparent Capturing

Any optimization approach that might temporarily deteriorate the user experience is hard to justify. The only stage of our approach that happens online and could affect the user is capture. We have to establish that its runtime overhead is low enough to have no negative effect on the user.

Figure 10 shows a detailed breakdown of the overheads introduced by capturing a *hot region*. There are three sources of overhead. Fork is the time it takes to call `fork` and return. It ranges roughly from 1ms to 6ms depending on the application and the state that needs to be replicated for the child. Preparation includes everything else that happens before we start executing the *hot region*. Most of this time is 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. The rest of the overhead is incurred during the execution of the *hot region* due to the additional page faults and Copy-on-Writes. 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 modified pages leading to a high Copy-on-Write overhead.

The lowest overall overhead was 5.7ms while the average was 14.5ms. Much of this can be attributed to the cost of communicating with the kernel: parsing page mappings

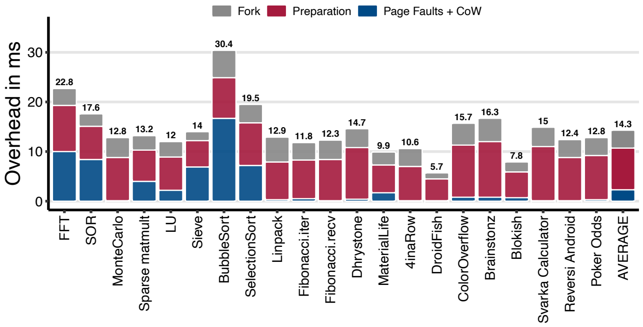

Figure 10. Breakdown of online overheads caused by our user-space capture mechanism. Preparation costs and much of the page-faults can be avoided with an in-kernel implementation. Regardless, with a maximum of ~30ms overhead and an average of 14.5ms, our infrequent captures remain unnoticeable by the users.

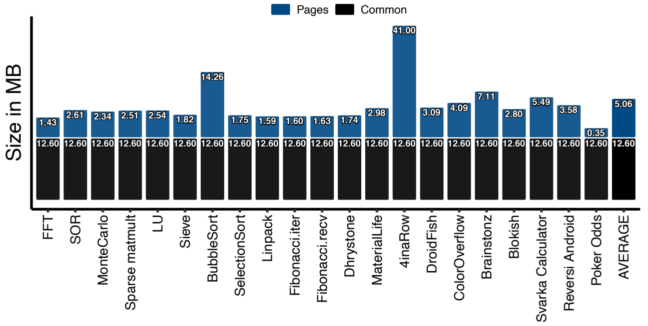

Figure 11. Storage overhead for capturing all pages used by the *hot region* vs capturing only the process-specific pages. While the overhead of capturing all pages is not unreasonable, we can reduce it even further by storing pages that are common across all applications only once. This brings the storage overhead down to 5.06MB on average.

through the slow */proc*, read-protecting pages one-by-one, jumping between user and kernel space to handle page faults. In-kernel solutions could eliminate most of it but they are beyond the scope of our work. User-space solutions could tackle some of the page fault overhead by exploiting spatial locality, e.g. using a coarser granularity to determine read pages. In any case, even the highest overhead of around 30ms is low enough to not have a significant effect on the user.

5.4 Capture Storage Overheads

It is important that our capture mechanism does not hog the phone'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.

Figure 11 shows the storage overheads for all applications. The average size of the pages captured for each application is less than 18MB. More than two thirds of that though is not process-specific. It represents Android runtime instances of immutable objects that are identical across all processes created during the same device boot. A single capture of these pages is enough for all applications. The remaining 5.06MB on average are the pages we actually need to save for each hot region capture. The smallest size of these program-specific pages is 356KB for *Poker Odds*, while the largest was 41MB for *4inaRow*. In the majority of cases, it is between 1MB and 5MB. For most programs, the captured information is in the same order of magnitude as the data the application actively uses in the *hot region*. When considering the heap allocated data alone (which represent the majority of the application's dynamic state), our approach captures on average only 6% of the heap data. Depending on the case, that percentage can be as low as 0.3%, for example capturing only 0.3MB out of the 88.4MB of *Poker Odds*'s heap space.

With our capture mechanism, we only need to store a small fraction of the data a brute force approach would store. That level of storage is manageable even for low-end devices, especially given that this overhead is transient. Once we have optimized the application, we can discard the captured data and release space back to the user.

6 Related Work

Iterative compilation is a well established technique, studied for more than two decades [30], with its early applications focused mainly on embedded systems [1, 9]. Early work in this area relied on exhaustive search of a simple optimization space. Later approaches aim at avoiding the cost of an exhaustive search through random search [15], genetic search [14], and other self-adapting techniques [2, 4, 11, 20, 23, 33]. While all of them improve the quality of the search, either through better optimizations or faster search, they assume offline evaluation of optimizations with a fixed input in a controlled environment.

Fursin et al. [22] has developed an online approach that evaluates transformation online by exploiting performance stability. This approach, however, will adversely impact users as it is susceptible to significantly slower evaluations, runtime crashes, or erroneous outputs. On top of that, the variable input and the noisy online environment would require significantly more evaluations per transformation to confidently rank them, to the point that it is no longer practicable. Our approach allows bulk-evaluation of transformations without affecting the user, while inherently tackling execution noise, by operating at idle times and imposing a specific execution environment for each replay.

Capture and replay frameworks are also well studied. Many approaches have relied on instrumentation [28, 29, 40, 50] in order to reduce the amount of the captured data. By capturing fine-grained events like variable reads or writes, they can minimize the needed state to a minimum. However, this results in very high overheads since each event has to be instrumented. Other approaches [26, 47] avoid this low level instrumentation, however they either modify the Java API or require the application to be instrumented with external tools in order to use their custom-developed APIs. CRIU [48] is a checkpoint and restore system based on Linux that is language-agnostic avoiding the overheads and limitations of previous approaches. Its main drawback is that it captures the whole application state. This fits CRIU's intended usage well, migrating processes in data centers, but it is a poor match for replaying hot regions.

The closest existing approach is CERE [10]. It identifies codelets, distinct computationally important areas of code, and it uses a page-based mechanism to capture each codelet's working set so that it can replay it in isolation. It differs from our approach 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 stops until CERE copies the page to a temporary buffer, regardless of whether the page will be modified or not. Copy-on-Write keeps this process in kernel space and only copies pages that are modified, allowing us to capture interactive applications online without affecting the user experience.

7 Conclusion and Future Work

In this paper we introduced an optimization approach for mobile devices, transparent to developers and users. An offline approach would rely on representative inputs, which are hard to reconstruct, especially for interactive applications. An online approach, while it could use real user inputs, will expose the user to suboptimal or even erroneous executions. On top of that, input variability makes the robust comparison of optimizations an extremely lengthy process.

Our approach is able to tune interactive applications using real inputs but without disrupting user experience. With sporadic online captures, we store the input for targeted code regions with negligible overheads. Later, we can re-execute each region on-demand with the exact same context but different optimization strategies to compare them and discover the best ones. We do this while the device is idle and charged to avoid affecting the user in any way.

We have implemented a prototype replay-based iterative compilation system on Android, including a novel LLVM backend for the Android compiler that significantly increases the optimization space. We capture inputs once per optimization run, with less than 15ms overhead on average per capture. For this almost imperceptible effect on user experience, we achieve average speedups of 44% over the Android compiler and 35% over the LLVM -O3 optimization level.

This work has produced some exciting results but there are more untapped opportunities for optimization. The code produced by the LLVM backend is not as efficient as it could be. Also, there are still unimplemented intrinsics which would allow us to replace JNI methods with LLVM IR. This can increase the amount of code we compile and therefore optimize. Our post-loop optimization could be expanded to other slow check operations, e.g. cases where not all array bounds checking is necessary. Finally, we aim to extend LLVM to support reserving and re-purposing registers, similar to what the Android compiler does for accessing runtime structures.

Acknowledgments

This work was supported by the Royal Academy of Engineering under the Research Fellowship scheme.

References

[1] Bas Aarts, Michel Barreteau, François Bodin, Peter Brinkhaus, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John Gurd, Jan Hoogerbrugge, Ping Hu, et al. 1997. OCEANS: Optimizing compilers for embedded applications. In *Euro-Par'97 Parallel Processing*. Springer, 1351–1356. https://doi.org/10.1007/BFb0002894

[2] Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. 2006. Using machine learning to focus iterative optimization. In *Proceedings of the international symposium on code generation and optimization*. IEEE Computer Society, 295–305. https://doi.org/10.1109/CGO.2006.37

[3] The Algorithms. 2020. *Sorting Algoirthms*. https://github.com/TheAlgorithms/Java

[4] Lelac Almagor, Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven W Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. 2004. Finding effective compilation sequences. *ACM SIGPLAN Notices* 39, 7 (2004), 231–239. https://doi.org/10.1145/998300.997196

[5] Android. 2020. *Android compiler: pathological cases it cannot compile*. https://android.googlesource.com/platform/art/+/refs/tags/android-10.0.0_r11/compiler/compiler.cc#48

[6] Android. 2020. *HGraph: Android compiler's Intermediate Representation (IR)*. https://android.googlesource.com/platform/art/+/refs/tags/android-10.0.0_r11/compiler/optimizing/nodes.h#312

[7] Quarzo Apps. 2020. *4 in a row*. https://play.google.com/store/apps/details?id=com.quarzo.fourinarow&hl=en&gl=US

[8] GSM Association. 2020. *The Mobile Economy 2020*. https://www.gsma.com/mobileeconomy/wp-content/uploads/2020/03/GSMA_MobileEconomy2020_Global.pdf

[9] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. 1998. Iterative compilation in a non-linear optimisation space.

[10] Pablo De Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. 2015. CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Trans. Archit. Code Optim.* 12, 1 (April 2015). https://doi.org/10.1145/2724717

[11] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael FP O'Boyle, and Olivier Temam. 2007. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization (CGO'07)*. IEEE, 185–197. https://doi.org/10.1109/CGO.2007.32

[12] Yang Chen, Shuangde Fang, Lieven Eeckhout, Olivier Temam, and Chengyong Wu. 2012. Iterative optimization for the data center. *ACM SIGARCH Computer Architecture News* 40, 1 (2012), 49–60. https://doi.org/10.1145/2248487.2150983

[13] K Cooper, Timothy Harvey, Devika Subramanian, and Linda Torczon. 2002. *Compilation order matters*. Technical Report. Technical Report, Rice University.

[14] Keith D Cooper, Philip J Schielke, and Devika Subramanian. 1999. Optimizing for reduced code space using genetic algorithms. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 1–9. https://doi.org/10.1145/315253.314414

[15] Keith D Cooper, Devika Subramanian, and Linda Torczon. 2002. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing* 23, 1 (2002), 7–22. https://doi.org/10.1023/A:1015729001611

[16] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 95–105. https://doi.org/10.1145/3213846.3213848

[17] Peter J Denning. 2006. The locality principle. In *Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe*. World Scientific, 43–67. https://doi.org/10.1145/1070838.1070856

[18] Jack J Dongarra, Cleve Barry Moler, James R Bunch, and Gilbert W Stewart. 1979. *LINPACK users' guide*. SIAM. https://doi.org/10.1137/1.9781611971811

[19] Juan Durillo and Thomas Fahringer. 2014. From single-to multi-objective auto-tuning of programs: advantages and implications. *Scientific programming* 22, 4 (2014), 285–297. https://doi.org/10.1155/2014/818579

[20] Deji Fatiregun, Mark Harman, and Robert M Hierons. 2004. Evolving transformation sequences using genetic algorithms. In *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*. IEEE, 65–74. https://doi.org/10.1109/SCAM.2004.11

[21] FelipeRRM. 2020. *Reversi Android*. https://github.com/FelipeRRM/AndroidReversi

[22] Grigori Fursin, Renato Miceli, Anton Lokhmotov, Michael Gerndt, Marc Baboulin, Allen Malony, Zbigniew Chamski, Diego Novillo, and Davide Del Vento. 2014. Collective mind: Towards practical and collaborative auto-tuning. *Scientific Programming* 22, 4 (2014), 309–329. https://doi.org/10.1155/2014/797348

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

[24] Google. 2020. *Android Optimizing compiler backend*. https://android.googlesource.com/platform/art/+/refs/tags/android-10.0.0_r11/compiler/optimizing/optimizing_compiler.cc

[25] Google. 2020. *Android Optimizing compiler backend: code transformations*. https://android.googlesource.com/platform/art/+/refs/tags/android-10.0.0_r11/compiler/optimizing/optimization.h#68

[26] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet lightweight record-and-replay for Android. *ACM SIGPLAN Notices* 50, 10 (2015), 349–366. https://doi.org/10.1145/2814270.2814320

[27] Takanori Ishikawa. 2020. *Fibonacci*. https://gist.github.com/ishikawa/16670

[28] Ajay K Jha and Woo J Lee. 2013. Capture and Replay Technique for Reproducing Crash in Android Applications. In *Proceedings of the 12th IASTED International Conference in Software Engineering*. 783–790. https://doi.org/10.2316/P.2013.796-025

[29] Shrinivas Joshi and Alessandro Orso. 2007. SCARPE: A technique and tool for selective capture and replay of program executions. In *2007 IEEE International Conference on Software Maintenance*. IEEE, 234–243. https://doi.org/10.1109/ICSM.2007.4362636

[30] Toru Kisuki, Peter M Knijnenburg, M O'Boyle, François Bodin, and Harry A Wijshoff. 1999. A feasibility study in iterative compilation. In *High Performance Computing*. Springer, 121–132.

[31] Sameer Kulkarni and John Cavazos. 2012. 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*. 147–162. https://doi.org/10.1145/2384616.2384628

[32] Chris Lattner and Vikram Adve. 2004. 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, 75. https://doi.org/10.1109/CGO.2004.1281665

[33] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. 2009. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 81–91. https://doi.org/10.1145/2536688

[34] San-Chih Lin, Chi-Kuang Chang, and Nai-Wei Lin. 2008. Automatic selection of GCC optimization options using a gene weighted genetic algorithm. In *2008 13th Asia-Pacific Computer Systems Architecture Conference*. IEEE, 1–8. https://doi.org/10.1109/APCSAC.2008.4625477

[35] Velbazhd Software LLC. 2020. *Brainstonz*. https://f-droid.org/en/packages/eu.veldsoft.brainstonz

[36] Velbazhd Software LLC. 2020. *ColorOverflow*. https://f-droid.org/en/packages/eu.veldsoft.colors.overflow

[37] Velbazhd Software LLC. 2020. *PokerOdds (Vitosha)*. https://f-droid.org/en/packages/eu.veldsoft.vitosha.poker.odds

[38] Velbazhd Software LLC. 2020. *Svarka Calculator*. https://f-droid.org/en/packages/eu.veldsoft.svarka.odds.calculator

[39] NIH. 2020. *Sieve*. https://imagej.nih.gov/nih-image/java/benchmarks/sieve.html

[40] Alessandro Orso and Bryan Kennedy. 2005. Selective capture and replay of program executions. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 1–7. https://doi.org/10.1145/1082983.1083251

[41] Eunjung Park, Sameer Kulkarni, and John Cavazos. 2011. 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, 65–74. https://doi.org/10.1145/2038698.2038711

[42] Roldan Pozo and Bruce Miller. 2004. *SciMark 2.0*. https://math.nist.gov/scimark2/

[43] Chromium Project. 2020. *Telemetry*. https://source.chromium.org/chromium/chromium/src/+/master:third_party/catapult/telemetry/

[44] scoutant. 2020. *Blokish*. https://f-droid.org/en/packages/org.scoutant.blokish

[45] Volker Seeker, Pavlos Petoumenos, Hugh Leather, and Björn Franke. 2014. Measuring qoe of interactive workloads and characterising frequency governors on mobile devices. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 61–70. https://doi.org/10.1109/IISWC.2014.6983040

[46] Juanky Soriano. 2020. *MaterialLife*. https://play.google.com/store/apps/details?id=com.juankysoriano.materiallife&hl=en&gl=US

[47] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. 2000. *jRapture: A capture/replay tool for observation-based testing*. Vol. 25. ACM. https://doi.org/10.1145/347324.348993

[48] Virtuozzo. 2020. *Checkpoint and Restore In Userspace*. https://www.criu.org

[49] Reinhold P Weicker. 1984. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM* 27, 10 (1984), 1013–1030. https://doi.org/10.1145/358274.358283

[50] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. 2007. 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, 85–94. https://doi.org/10.1145/1287624.1287638

[51] Peter Österlund. 2020. *DroidFish Chess*. https://play.google.com/store/apps/details?id=org.petero.droidfish&hl=en&gl=US