Object Intersection Captures on Interactive Apps to Drive a Crowd-Sourced Replay-Based Compiler Optimization

PASCHALIS MPEIS, University of Edinburgh, UK
PAVLOS PETOUMENOS, University of Manchester, UK
KIM HAZELWOOD, Facebook AI Research, USA
HUGH LEATHER, Facebook AI Research, USA

Traditional offline optimization frameworks rely on representative hardware, software, and inputs to compare different optimizations 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 erroneous code. Therefore, our mobile code is poorly optimized, resulting in wasted performance and 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. Our previous work [46], captures the state accessed by any targeted code region during its online stage. 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. In this paper, we propose a technique that further decreases the storage sizes without any additional overhead. It captures only the intersection of reachable objects and accessed heap pages. We compare this with another new approach that has minimal runtime overheads, at the cost of higher capture sizes. Coupled with a search heuristic for the compiler optimization space, our capture and replay mechanism allows us to discover optimization decisions that improve performance without testing these decisions directly on the user. Finally, with crowd-sourcing we split this offline evaluation effort between several users, allowing us to discover better code in less time.

We implemented a prototype system in Android based on LLVM combined with a genetic search engine and a crowd-sourcing architecture. We evaluated it on both benchmarks and real Android applications. Online captures are infrequent and introduce ~5ms or 15ms on average, depending on the approach used. For this negligible effect on user experience, we achieve speedups of 44% on average over the Android compiler and 35% over LLVM -O3. Our collaborative search is just 5% short of that speedup, which is impressive given the acceleration gains. The user with the highest workload concluded the search 7x faster.

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

Additional Key Words and Phrases: iterative compilation, capture, replay, interactive

Extension of Conference Paper [46]. This paper introduces two novel capture mechanisms. The first minimizes the capture sizes by intersecting the reachable heap objects with the memory pages that were identified as input. The second favors a faster runtime operation at the cost of increased capture sizes. Furthermore, the optimization space exploration was accelerated by incorporating a joint, crowd-sourced genetic search among different users.

Authors' addresses: Paschalis Mpeis, p.mpeis@ed.ac.uk, University of Edinburgh, Edinburgh, UK; Pavlos Petoumenos, pavlos.petoumenos@manchester.ac.uk, University of Manchester, UK; Kim Hazelwood, kimhazelwood@fb.com, Facebook AI Research, Menlo Park, USA; Hugh Leather, hleather@fb.com, Facebook AI Research, Menlo Park, USA.

ACM Reference Format:
Paschalis Mpeis, Pavlos Petoumenos, Kim Hazelwood, and Hugh Leather. 2022. Object Intersection Captures on Interactive Apps to Drive a Crowd-Sourced Replay-Based Compiler Optimization. *ACM Trans. Arch. Code Optim.* 1, 1, Article 1 (January 2022), 25 pages. https://doi.org/10.1145/3517338

1 INTRODUCTION

More and more of our computing needs rely on smart mobile devices. With active users now more than 5.2 billions [9], 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 [28]. 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 [35], we need to determine whether each choice will be beneficial or not. We could evaluate all choices on the same representative mobile system, program, and input and then assume that good choices on that setup will always be beneficial. With a multitude of processor designs, system capabilities, software stacks, operating environments, and usage patterns [52], 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.

Evaluating optimizations 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 [22]. Many of these decisions can easily be suboptimal, severely affecting the user experience. Statistical approaches can limit this negative impact at the cost of an even slower process [14] but in a low-latency environment even rare slowdowns that are noticeable by the users are unacceptable [52]. In any case, when tested optimizations lead to erroneous results, something relatively common with optimizing compilers [1], 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. Our work introduces 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.

[1]Optimizing compilers [18] consider not just the typical flags but also more extreme parts of the space.

Traditional capture mechanisms [11, 48, 55, 57] 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. Our input capture mechanisms were developed with this aim. Our previous work, leverages the kernel Virtual Memory subsystem to automate and accelerate the capture. Using memory protection we monitor, with little overhead, the pages accessed while a targeted region runs. Future replays will only need to access these pages, a small fraction of an application's memory. For those pages, we need their original state before they are modified by the application. Instead of copying them preemptively in user space [11], 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. In this work we reduce the capture sizes even further, without requiring any additional overhead. We identify the heap objects that are reachable by the region's code and intersect them with the accessed heap pages. Any unreachable data in those pages are not needed for a replay, so we erase them. We also introduce an alternative approach that favors a simpler and more lightweight runtime operation over capture sizes. All of our approaches have unnoticeable runtime overheads.

We fused capture and replay into a novel iterative compilation system that optimizes mobile applications for the actual environment and context they are being used in. We use genetic search 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 decision over the whole search and we apply it on the online version of the application, the one the user interacts with.

All evaluation 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.

To accelerate the offline evaluations, we have implemented a crowd-sourcing module to this system. It allows performing a collaborative search amongst different users, splitting between them the cost of evaluating candidate optimizations. 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. The more users that collaborate the better, enabling deeper searches with less effort from each user.

This framework is compiler-agnostic: any compiler that can process Dalvik code [6] and generate code for Android devices can be plugged in. It can also be applied to different Operating Systems (OSes) and even for non-mobile environments. For Android, currently there is only one Dalvik toolchain, the default one, and it has a very limited optimization space with 18 distinct optimizations. We overcame this limitation by creating a new Android backend[2] based on LLVM which offers a far more expansive and interesting optimization space.

We used Android apps, both benchmarks and interactive ones, to evaluate our approach. With profiling and static bytecode analysis we identify code that can be accurately replayed and is worth optimizing. Our online captures run a handful of times per day and are imperceptible to the user. Our *Page* and *Intersection* approaches minimize stored data and require an average overhead of less than 15ms, while our capture *Everything* approach requires less than 5ms. Powered by capture and replay, our iterative compilation approach searches for good optimizations without exposing users

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

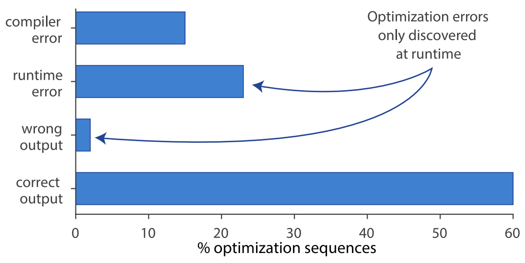

Fig. 1. Compilation outcome with LLVM using 100 randomly generated optimization sequences for the FFT kernel from *Scimark*. 25% of the sequences result in a binary that does not behave as expected at runtime. In an online optimization system, this would directly affect the user experience and might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

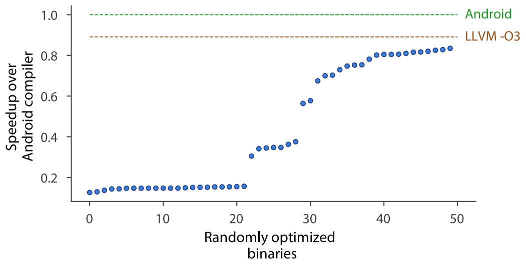

Fig. 2. Speedup over the Android compiler for 50 randomly generated LLVM optimization sequences applied on FFT. Sequences that break the compiler or the app are discarded. All of them slow down the app relative to the Android compiler and LLVM -O3, up to 8x. Evaluating these sequences online would have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

to slow or erroneous binaries. We achieve on average 44% whole program speedups over Android and 35% over LLVM -O3, significantly improving the user experience. Our joint search delivers speedups within 5% of those achieved by individual searches, which is impressive considering that only a fraction of a user's evaluation effort is now required. With 10 users, we were able to accelerate the search by 7x for the user that had the biggest workload and by 12x for the same user when compared to the average. Furthermore, we were able to get near optimal results for new users after as little as a few minutes of participation in the search.
The main contributions of this paper are:

- two code region capture mechanisms for Android that minimize runtime or storage overheads,
- a replay-based evaluation mechanism of different code transformations driven by actual usage patterns,
- a novel prototype iterative compilation system, with a crowd-sourcing module that accelerates the optimization search amongst different users, without any developer effort or impact on the user experience.

This paper is organized as follows. Section 2 describes the motivation behind our work, followed by Section 3 that describes our crowd-sourced iterative compilation framework. The experimental setup is on Section 4, followed by Section 5 that presents our results. Section 6 discusses related work and Section 7 offers some concluding remarks.

2 MOTIVATION

Iterative compilation [35] is an optimization technique that can outperform a compiler's default presets. It searches for different code transformations and their parameters, evaluates them for performance, and at the end keeps the best performing binary. There are multiple techniques for searching: random, through genetic search, and statistical [49] or machine learned models [38]. Despite the significant performance benefits of iterative compilation, it has not been applied in a general way on mobile systems.

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[3]. By repeatedly executing an app with a deterministic input under different

[3]A few benchmarks do not take input so they do not require representative inputs

optimizations, we can directly compare them 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 representative system where we can optimize an app once for every other system. We need to optimize for each one individually. Even then, it is hard to create inputs without explicit support from the app. Mobile apps tend to have complex inputs, including user events, sensors, 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.

There is no hard guarantee that optimizations will not introduce errors. Figure 1 shows what happens when we apply random LLVM optimizations on `FFT` [50]. Only 60% of the cases lead to a correct binary. 15% crash or timeout the compiler but this is manageable. The remaining 25% is not, as it leads to runtime errors: program crashes or timeouts, or even a wrong output. Offline approaches simply reject such binaries 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 data 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 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 app or disabling the optimizer.

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.

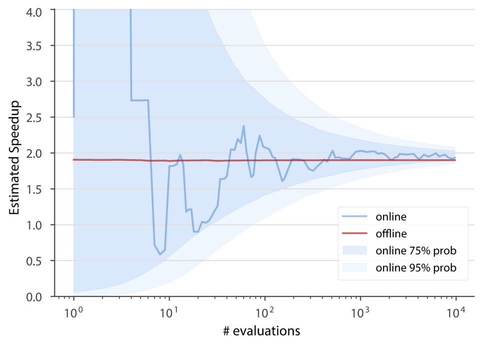

Fig. 3. Estimation of the speedup of LLVM -O1 over -O0 for the FFT benchmark as the number of evaluations increases. Online search performs each evaluation with a different randomly selected input between FFT_SIZE and FFT_SIZE_LARGE. Offline search always use the largest one. Lines represent speedup 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.

```
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
    foreach $c \in$ compilableRegion(method) 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 app's hot region.

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 describes our page capture mechanism, followed by Section 3.3 which introduces our replay mechanism. Section 3.4 explains how we extract information from captures for optimization and automatic code correctness verification, followed by Section 3.5 that describes two additional capture mechanisms. In Section 3.6, 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 3.7. Section 3.8 shows, 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. Finally, Section 3.9 presents a crowd-sourcing module built into our iterative compilation system that enables a deeper search in less time.

3.1 Identify 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 can accurately replay and are worth optimizing. The former translates into their behavior being fully determined by their memory state and having no observable effects outside of it. The latter means that they take most of the execution time and are compilable by our LLVM backend.

Through static bytecode analysis we initially identify methods that might interfere with replaying. These are methods with I/O or other 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 replace with intrinsics, as described in Section 3.6. Finally, we blocklist exceptions. They are the only 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 executes infrequently, with 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. Even for the cases that the region is not in the critical path of a multi-threaded operation, the user would still benefit indirectly from optimizations, i.e., by saving computational cycles.

3.2 Online Page-Capture Mechanism

Our approach captures the application's behavior while it is being used and accurately replicating that behavior later offline under different optimizations. Since we exclude methods with I/O or

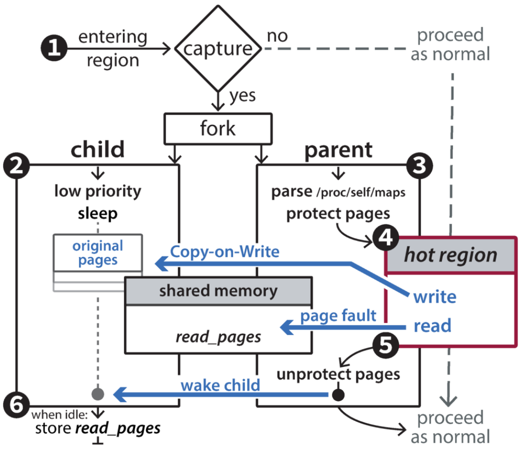

Fig. 4. Our capture mechanism stores pages read by the *hot region* to enable replaying at any future point in time. With fork and Copy-on-Write we copy the original state in a child process. With read protection and fault-handling we identify the read pages.

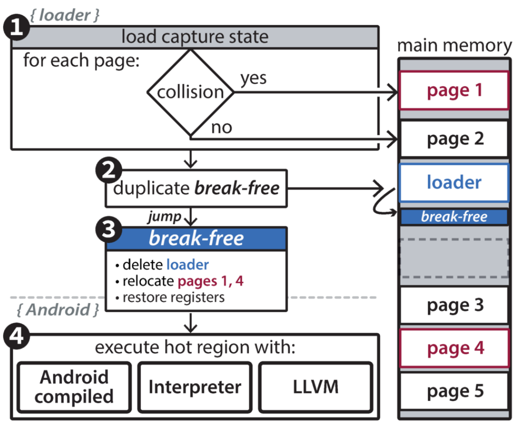

Fig. 5. Our replay starts with a *loader* program that transforms itself into a partial Android process whose state is the previously captured one. The loader can swap different code versions for executing the *hot region*. It works alongside ASLR, a memory-shuffling security mechanism.

non-determinism, 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 memory pages of a process. 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 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 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. Our mechanism is shown on Figure 4 and is detailed below:

1) Capture initiated. We check at the entry point of a hot region whether a capture should occur. A single capture is enough for an optimization search. Since there are plenty opportunities to get one, we set a low capture frequency. 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 region, which might increase capture size.

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 physical pages. When the parent modifies a page, the kernel's efficient Copy-on-Write automatically duplicates the original page to the child, which is faster than storing the all memory before the region runs [55] or manually copying a page just before it is first used [11]. 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 [19], we expect the number of these events to be low.
5) Ending the hot region. After the parent has executed the region, it wakes up the child, restores access permission to any remaining read-protected pages, and uninstalls the fault handler. Finally, it continues executing as usual.
6) Saving the memory state. Once the child has received a synchronization signal from the parent, it begins spooling out pages marked as read to disk. Since it has the lowest priority possible, this happens only when there is 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, we cannot read-protect some of the pages to find out whether they are accessed. Read-protecting memory belonging to the runtime, its library dependencies, or GC auxiliary structures could crash the process. We choose to always capture such pages.

A significant chunk of the used pages is not unique across processes. It contains immutable runtime structures that only change across device boots, which we capture only once per boot. It also contains several memory-mapped system files that only change on system updates, like any executable 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

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 [55]. 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 we put into memory the captured pages. We place most of them directly in their original virtual addresses. 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, by consulting the captured memory layout and choosing locations not used by it. This approach does not make any assumptions about the loader's address layout, so it can work in the presence of memory shuffling security mechanisms, like ASLR. We also read the captured processor's 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 do online. Our prototype uses 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. Despite not providing a guarantee that no errors will occur after the verification process, we have not encountered any such cases. Nevertheless, it might be wise to limit the search optimization space on a production environment.

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 in the optimized code 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 Capturing at different granularities

Our *Page* capture mechanism balances computational effort and storage overheads. In this section, we present two additional mechanisms for capturing inputs that deliver different trade-offs. The first, significantly decreases the storage requirements of our existing approach [46] for a little bit more computational effort but without causing any extra runtime overhead. It achieves this by intersecting the hot region reachable objects with the input memory pages. The second, favors a simpler and faster runtime operation at the cost of increased storage size.

Intersecting live object data with input pages. Our original capture mechanism identifies memory pages that have been read by the hot region and captures them in their entirety. In this section we present a novel approach that stores significantly less data without causing any extra overhead to the users. It operates by intersecting any reachable heap objects with the input pages that our original mechanism can identify.

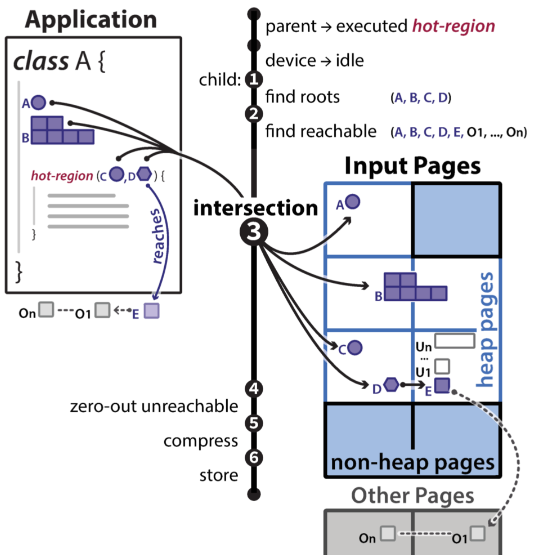

Fig. 6. Reducing capture sizes by intersecting reachable objects with memory pages containing the input. We visit the region's parameters and fields to construct the reachable set. Most reachable objects are not in the input pages ($O_1, ..., O_n$), while much of the input pages contain unreachable objects ($U_1, ..., U_n$). By finding the reachable ones inside the *Input Pages*, we reclaim the space shown as a white area.

```
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 2: Object intersection with input pages.

Figure 6 visualizes the proposed object intersection mechanism. Similarly with the *Page capture* mechanism, it operates in a child process after the hot region has finished executing. As the child process executes with low priority (see Section 3.2), the intersection mechanism will run only when there are idle computing resources, so that it does not interfere with any unrelated user activities. Therefore, it does not cause any extra noticeable overheads. Initially, the child creates a set of root objects that can be accessed directly from the 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 reducing the capture size. 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 [20]. Lastly, it stores the reduced input data to permanent storage.

Algorithm 2 outlines in greater detail the intersection mechanism. Initially, the hot region's *root set* is calculated. This includes all the classes as well as the parameters of the region's entry point method. Then, each object in the *root set* is visited using a breadth first search traversal. Each visited object is added to a *reachable set* and its references are scheduled (in *workQueue*) to be visited afterwards. Once we find all reachable objects, we sort them based on their memory addresses before iterating over them. For each object, we check whether it *resides* inside the *input* *pages* and if so we calculate its memory footprint and mark it as potentially reachable by the region. This allows us to discover the unreachable area so we can then zero it out. Once we do that, we compress the pages, which now only contain reachable data. As we will see in the results, this leads to impressive space savings.

Capturing the full state. We also introduce an alternative input capture mechanism, named *Everything*, that stores all data present in physical memory. We use it in our evaluation to better quantify the costs and benefits of reducing the capture size with the *Page* and *Intersection* mechanisms. The main benefits of this approach are simpler implementation and operation, which translate into lower 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 *Pages* and *Intersection* mechanisms rely upon. Then it stores all of the pages in the program's memory space (enumerated in the `proc/maps` Linux pseudofle [40]). As we will demonstrate later, this significantly decreases the online operation overheads, at the cost, however, of increased storage size.

3.6 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 [29] which are guaranteed to have only positive or no impact at all. It lacks more aggressive code transformations found in established compilers like LLVM [37]. The potential for better compiler optimization strategies is limited.

To overcome this limitation, we have developed a new Android compiler backend based on LLVM [45]. The core of our implementation focuses on transforming `HGraph` [7] 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 that 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. The second optimization replaces JNI calls that implement particular math library methods with either LLVM intrinsics or our own custom LLVM IR implementation. This not only avoids the overhead of JNI calls but it also increases the amount of code we can compile and optimize, making the rest of our optimization passes more effective.

This backend is a work in progress. Yet, it already produces better code than the default Android backend, as we will see in Section 5. With more engineering work to improve it, our optimization strategy will only become better.

3.7 Searching Compilers' Space with a GA

LLVM has a huge optimization space with ~200 passes that can be applied multiple times with a different effect each time that depends on previously and subsequently applied passes [15]. On

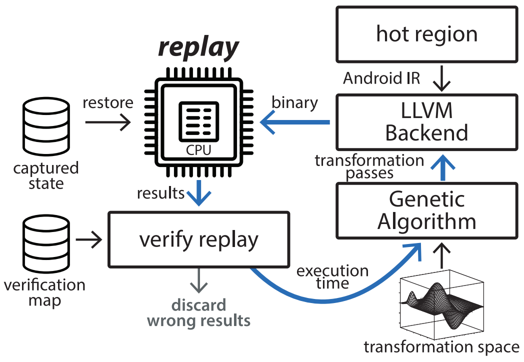

Fig. 7. The Replay-based Iterative Compilation main loop (highlighted). The GA (lower right) selects interesting optimization decisions from the LLVM transformation space. It passes them to the LLVM backend which applies them on the hot region. We replay the captured execution using the newly optimized 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.

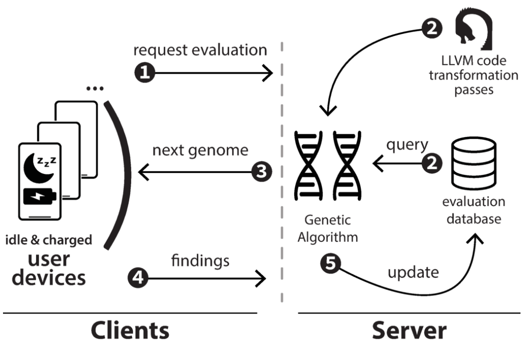

Fig. 8. Accelerating the GA search with crowdsourcing. We utilize idle and charged devices to spread the evaluation effort amongst different users. Several client devices communicate with a server which instructs them on what genome to evaluate next, based on the code transformation space and the previous evaluation data. The device then evaluates the next point and reports the findings to the server, which finally updates its database.

top of that, it has more than 1300 parameters and flags. Applying a pass or changing a flag might improve performance or 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 we implemented was a Genetic Algorithm (GA). It is a well established strategy that has worked really well in the past for similar problems [16, 23, 36, 39]. 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 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 region (see Section 3.3), but if the performance of two binaries is close enough, then we prefer the smaller binary. We provide the GA parameters in Section 4.

3.8 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 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. This section outlines our solution that utilizes the individual components from previous sections.

In Figure 7 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.

3.9 Using crowd-sourcing to accelerate the genetic search

By pushing iterative compilation offline we hide from the users the significant overheads that come with it. These include compilations and repeated evaluations, which are necessary for statistically meaningful comparisons but can be very slow [46]. This section describes a scalable, crowd-sourced architecture to accelerate this offline effort. It performs a collaborative search over several users. As the evaluation data are now jointly produced, we can both perform a more extensive search and almost immediately deliver tangible performance benefits to new users joining the search.

Figure 8 shows how we perform a crowd-sourced collaborative search. Once several users have had their inputs captured, they can participate in a joint search, coordinated by the server, as long as their devices remain idle and charged. Initially, a device requests the next genome to evaluate. The server then queries the database for existing evaluation data, as well as the code transformation space of the 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 our replay-based mechanism. Finally, the device reports the results to the server, which updates its database. These results include compilation time and size, any runtime or compilation errors, and the evaluation timings. All this data are used to tune the search for the best genome of a hot region amongst several users. This approach, as we will show 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.

4 EXPERIMENTAL SETUP

We have implemented a prototype system and evaluated our approach with a set of experiments on *Google Pixel 4* devices 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 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 3.6 to transform it into LLVM IR, and LLVM tools. LLVM was based on version 10. In particular, we use `opt` to optimize with a total of 197 passes and 710 parameters and flags, and `llc` to compile with 90 CPU-specific and 569 generic 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%. At 100 identical binaries we halt the algorithm. Each tournament selection round considers 7 candidates with a 90% probability.

Fig. 9. Android apps used in our experiments.

Type	Name	Description
Scimark [50]	FFT	Fast Fourier Transform
	SOR	Successive over-relaxation
	MonteCarlo	Estimates π value
	Sparse MatMult	Indirection and addressing
	LU	Linear Algebra kernels
Art	Sieve [47]	Lists prime numbers
	BubbleSort [3]	Simple sorting algorithm
	SelectionSort [3]	Simple sorting algorithm
	Linpack [21]	Numerical linear algebra
	Fibonacci.iter [32]	Fibonacci sequence iterative
	Fibonacci.recv [32]	Fibonacci sequence recursive
	Dhrystone [56]	General CPU performance
Interactive	MaterialLife [53]	Game of life
	4inaRow [8]	Puzzle Game
	DroidFish [58]	Chess Game
	ColorOverflow [42]	Strategic Game
	Brainstonz [41]	Board Game
	Blokish [51]	Board Game
	Svarka Calculator [44]	Odds for a card game
	Reversi Android [24]	Board Game
	Poker Odds [43]	Poker Statistical analysis

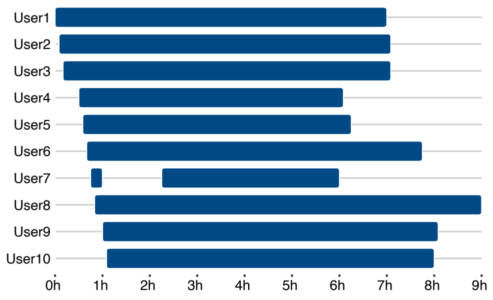

Fig. 10. User availability for a joint optimization search.

For our experiments we use Android applications that fall into three types, as listed in Table 9. 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. In our future work (see Section 7) we aim to test the proposed approach on a wider-range of source codes, as there might be some artificiality on this application list.

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 apps, this was tedious process. It only reinforced our conviction that deterministic and repeatable evaluations without replaying require significant effort.

During search, each transformation is evaluated 10 times through replay, unless it is substantially slower than the baseline. For those cases we perform only two evaluations. 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

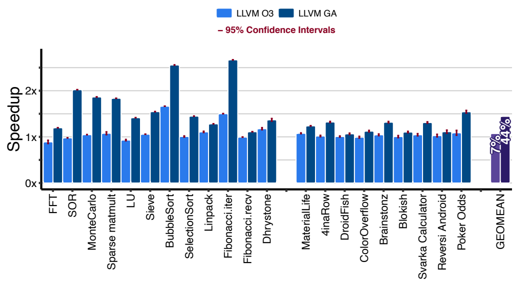

Fig. 11. Speedup relative to the Android compiler for 12 benchmark (left) and 9 interactive applications. LLVM -O3 performance ranges from a 0.89x to a 1.66x. LLVM GA, which uses a replay-based iterative compilation, produces from 1.10x to 2.56x speedups and an average of 44%.

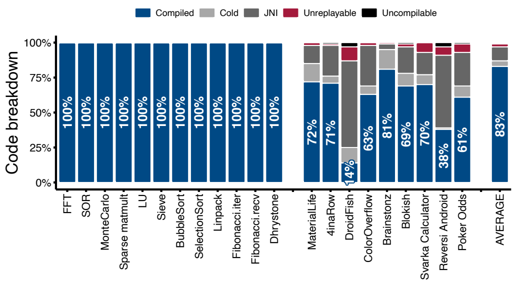

Fig. 12. Runtime code breakdown, measured online with a sample-based profiler, for 12 benchmarks (left) and 9 applications (right). *Compiled* is the code we optimize. *Cold* falls outside the *hot region*. *JNI* is C/C++ code. *Unreplayable* cannot be replayed. *Uncompilable* cannot be processed by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

speedups are also based on 10 evaluations per transformation but without outlier removal. Finally, our plots include 95% confidence intervals where applicable.

For our joint GA search we used a fixed initial population of 200 to ensure that the same amount of points in space are visited on each run. The algorithm runs remotely and stores the evaluation data in a document store. We evaluated this architecture using ten *Google Pixel 4* devices. Each device represented a different user and each user had a different input. For the interactive apps we generated new inputs using manual interactions, and for the benchmarks we altered the region's problem sizes. Once we captured all user inputs for each app, we programmed the phones to participate collaboratively, in a realistic way. Studies have shown that most users charge their phones overnight [25], when devices are also conveniently idle. Using sleep data [13], we created a user availability schedule, as shown by Figure 10 . 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.

5 RESULTS AND ANALYSIS

To evaluate our approach we performed six sets of experiments. The first shows our performance gains on the apps of Table 9 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 compares the online overheads of the proposed input capture mechanisms and the fourth one compares their storage requirements. Two of the approaches focus on minimizing the capture sizes, being applicable even on low-end devices, while an alternative one focuses exclusively on minimizing the runtime overhead. We demonstrate that all approaches have transparent online operation from the users. The last two experiments showcase the offline GA search acceleration when crowd-sourcing the findings from several users.

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 11 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, 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 apps, we had similar results when considering the speedup of *hot regions* in isolation (see Figure 13). When we evaluate our approach on the wider regions that surround the code we optimized, speedups range from 7% to 54%. This is impressive given that only part of the code is optimized. Figure 12 shows how much of the program's execution we can compile and optimize. These breakdowns are collected during normal operation, outside the replay environment, while the user interacts with the application. On average only 60% is *Compiled* and it ranges from 14% to 81%. 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*, representing on average ~27% of the interactive apps execution and as much as 62%. While the interactive apps naturally perform I/O, the amount of code that actually performs the calls is not as much as one would expect. *Unreplayable*, which has other sources of non-determinism too, takes only a 4% on average. This is quite small in comparison 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 one sixth of what our GA unlocks. Even worse, there are cases where -O3 is worse than Android. The most notable case is *FFT* where an increased amount of heap-related operations, e.g. checks for GC, slows down the program by 10%. In contrast with -O3, GA learns to solve this problem with loop unrolling combined with our post-loop optimization. This shows that even a highly optimizing compiler is not enough on its own. We also need a sophisticated way to discover code optimizations.

The reported speedups in Figure 11 underestimate the potential of our approach, as our LLVM backend is a work in progress. Our generated bitcode is not as efficient as it could be, while many *JNI* math library calls could be implemented with LLVM IR. Despite that we still manage to improve performance significantly. With a more mature compiler toolchain, we would expect these gains to grow even further. On the negative side, our approach will not be as effective with IO-bound applications. Some examples include applications that heavily exchange network messages or perform streaming. In our case, this would increase the unreplayable code region sizes. But in general, compiler optimizations would have a limited effect on such codes anyway.

5.2 Using a GA for Offline Optimization Search

With a GA search over the vast space of LLVM optimizations, we are able to readily outperform the Android compiler. This section illustrates this by showing the best genome evolution. We also show that GA, like any self-adapting algorithm, will inevitably test slow configurations, making it clear that any online approach is simply not feasible.

Figure 13 shows the evolution of best and worst genomes over time. In all cases the best binary outperforms Android. For almost all apps the GA improves performance over time, except for *Blokish* where the initial random search discovers the best performing binary. More than two thirds

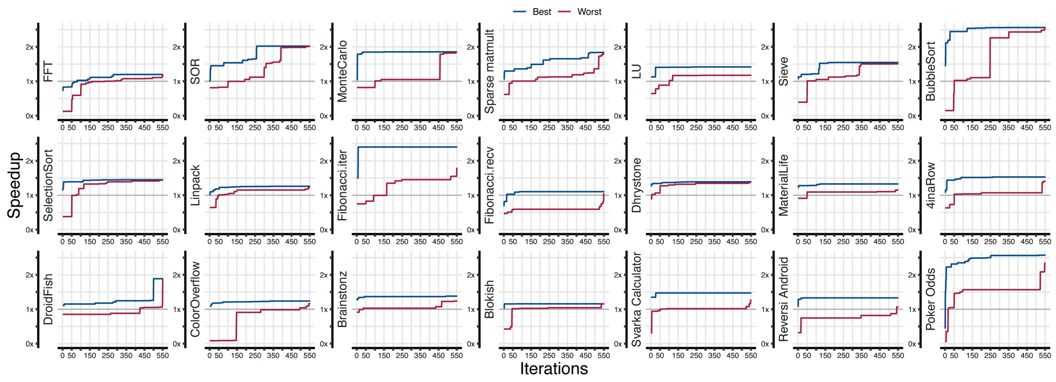

Fig. 13. Range of speedups over the Android compiler for the LLVM transformation sequences selected by the GA. Speedups are estimated through replay for the *hot regions* only. The two lines represent the evolution of the best and worst genomes over time. Vertical grid lines indicate the change from 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.

of our apps require multiple generations to discover their best binary, while a few might have benefited by a longer search, e.g. Sparse Matmult. Overall, the ability to search the optimization space and evaluate transformations in a robust way provides clear benefits.

On the other end, several genomes cause extreme slowdowns, as much as 10x. If they were evaluated online, they would have a dramatic effect on the users. For the first random generation of *FFT* and most 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 apps were still picking sub-optimal genomes even after 3 generations, while a few kept doing this to the very end. This does not even consider the slower evaluations that we discarded in the first generation (see Section 4). In any case, just a few slow 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 peeling 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. We also noticed that the applications that had good speedups for the baselines, did very well in the GA search as well (e.g., *BubbleSort* and *Fibonacci.iter*).

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. This section presents the runtime overheads for the proposed capture mechanisms. *Intersection* incurs the same overheads with *Pages* (our previous work), as the extra operations performed are on a low-priority, child process. There are four sources of overhead that are explained below.

Figure 14 shows a detailed breakdown of the overheads introduced by our infrequent captures. *Fork* is the time it takes to call `fork` and return. It ranges roughly from 1ms to 6ms, with ~3.5ms on average, depending on the application and the state that needs to be replicated for the child. *CoW* is

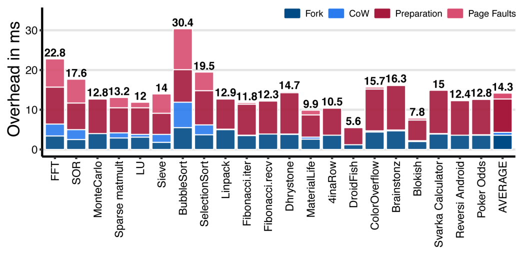
Fig. 14. Breakdown of the online capture overheads. *Fork* and *CoW* (Copy-On-Write) are common for all approaches *Page Faults* and almost all *Preparation* are avoided by our *Everything* approach, which has less than 5ms overhead on average. *Pages* and *Intersection*, despite some higher overhead, are still unnoticeable by the users, with a maximum of ~30ms overhead and a 14.3ms average.

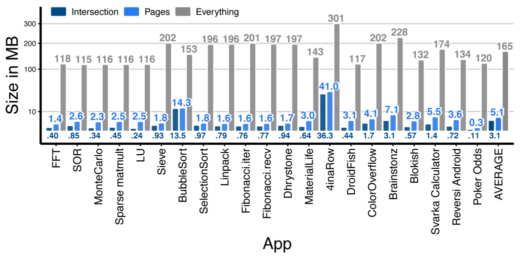
Fig. 15. Storage overhead for capturing user inputs at different granularity. *Everything*, does not make any attempt on minimizing the capture sizes. *Pages* captures only the memory pages read by the *hot region* and decreases sizes by an order of magnitude. By intersecting the reachable heap objects within those read pages the required storage is further decreased, by an additional 64%.

the overhead caused by the Copy-On-Write mechanism and ranges between ~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 we perform before we execute 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 incurs when the region runs, is increased as well for those cases, being around half of the *Page Fault* overheads.

For *Pages* and *Intersection*, the lowest overhead overall was 5.7ms and the average 14.5ms. Much of this is due to the communication with the kernel: parsing page mappings through the slow `/proc`, read-protecting pages them, and handling page faults. In any case, even the highest overhead (~30ms) is low enough to be noticed by the users. As the next experiment will show, *Everything* requires significantly more storage. However, regarding runtime overhead, it is 2.7x faster than the other two approaches, requiring less than 5ms on average.

5.4 Capture Storage Overheads

It is important that our capture mechanism does not require excessive storage. 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. In this section we compare our different approaches: 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 15 shows the storage overheads for all applications using different capture granularity. *Everything*, captures all pages that were in virtual memory right before entering the hot region. It does not attempt to minimize the capture data, requiring 165MB on average with the highest being

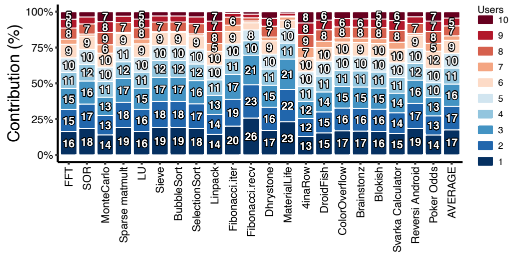

Fig. 16. User participation to the joint optimization search for all apps. Users that joined the search earlier had higher contributions. The least contribution came from *User7*, who was absent at a period where all of the remaining users were actively contributing.

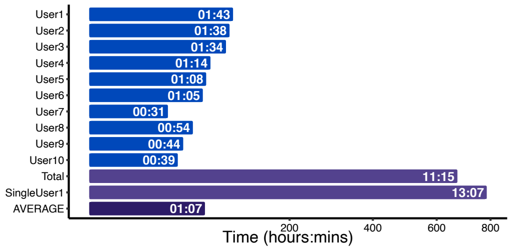

Fig. 17. User search contribution time alongside with the total time and average. *SingleUser1* shows the time required by *User1* to perform a same-length search on its own. On average, the joint search required about an hour, roughly 12x less than *SingleUser1*.

~300MB. As the previous experiment has illustrated, it operates significantly faster, making this approach appealing when the storage is not very limited.

Pages 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 is not unique. They represent immutable runtime instances that are identical across all processes from the same device boot. A single capture of those 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.

When considering the heap allocated data alone (that is the majority of an app's dynamic state), *Pages* stores on average 6% of the heap data and it can be as low as 0.3% (e.g., capturing only 0.3MB out of the 88.4MB for *Poker Odds*). That is a small fraction of what *Everything* would store, as several heap objects are stored outside the input pages. Still, there are many region unreachable objects residing inside those input pages, which are unnecessarily stored by the *Pages* approach. By doing a reachable object *intersection* with the input pages we can reclaim this space and significantly reduce capture sizes. *Intersection* requires 3.1MB of storage on average, 64% less than *Pages*. 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 pages. *Pages* and *Intersection* minimize well the amount of required storage, making them practicable even for low-end devices, especially given that this overhead is transient. Once we have optimized the app, we can discard the capture.

5.5 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 is a natural next step for our optimization approach.

To evaluate our collaborative GA search we have used 10 different mobile devices, as described in Section 4. Figure 16 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 a 30%. The last 4 users have contributed a significantly

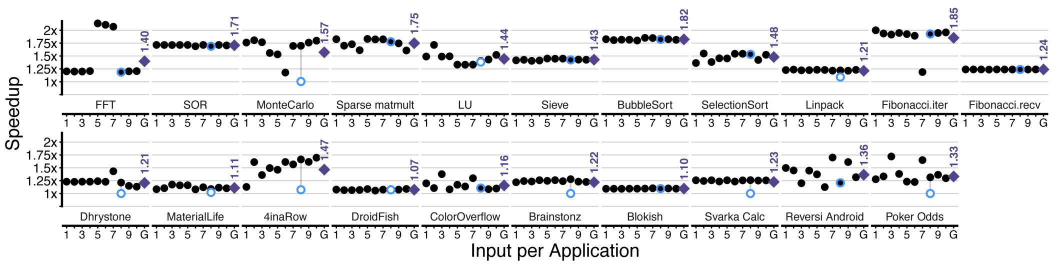

Fig. 18. Showing the best genomes for each application from a joint search, at each user's *exit point*. That is the point in time where a user has exited the collaborative search. *User7* had two such points with the first shown as a circle outline drawn with a lighter color. The average speedup is shown with a rhombus. Our 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 5.1 (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.

less 22%. *User7* had the least contribution as it suspended evaluations when all other 9 users were actively participating.

Figure 17 visualizes the time each user has put into the joint search. Only a fraction of each user's availability was required. *User7* contributed the least, for ~30 minutes. The most was 1h:43m by *User1*, who was the most active and operated on bigger workloads for the benchmarks. That is also evident when comparing the *Total* with *SingleUser1*, which shows a same-length search performed individually. In this particular scenario, *User1* puts more than 7x effort when not collaborating, and 12x more when compared to the *AVERAGE* contribution. When considering the availability time (see Figure 10), users were available for ~7 hours, with only 15% of it 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.

5.6 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 a collaborative search are still effective. This section presents the best genomes of our joint GA search amongst 10 users.

Figure 18 shows the speedups from the joint optimization search of each user's *exit point*. 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 5% less than the individual-input search (see Section 5.1). 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 on 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 had much variation between different users. These results collectively indicate that despite doing a collaborative search amongst different users, we are able to specialize (when needed) for each of them individually. This was not the case for two users from *Fibonacci.iter* and *MonteCarlo*, however, as they deviated a lot from the other ones. In our future work, we will consider either widening or even performing an individual search for such cases.

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 RELATED WORK

Iterative compilation is a well established technique [35]. Early approaches, focused on embedded systems [1, 10], were exhaustively searching simple optimization spaces. Later approaches avoided that, through random [17] or genetic search [16], and other self-adapting techniques [2, 4, 12, 23, 27, 30, 38]. While all of them improve the search quality, either with better optimizations or faster search, they assume offline evaluations, fixed inputs, and a controlled environment.

Fursin et al. [26] exploits performance stability to evaluate transformations online. However, this approach is susceptible to slow or erroneous evaluations. Additionally, the variable input and the operating noise require more evaluations for sound comparisons, to the point that is impracticable. Our approach allows bulk-evaluations without affecting the user, while inherently tackling noise, by operating at idle times and imposing an execution environment.

Capture and replay frameworks are also well studied. Many approaches use instrumentation [33, 34, 48, 57] to reduce the captured data. By capturing fine-grained events like variable reads or writes, they minimize the stored state. This causes significant overheads since each event has to be instrumented. Other approaches [31, 54] avoid this low level instrumentation, however they either modify the Java API or require instrumentation with external tools in order to use their custom APIs. CRIU [55] is a Linux-based system that avoids the overheads and limitations of previous approaches. It fully captures the memory, which fits its intended usage, but it is a poor match for replaying hot regions.

The closest existing approach, CERE [11], aims to accelerate large scientific applications by breaking them down into shorter code regions. It captures those using a page-based mechanism. As latency is not an issue for CERE, overheads are typically over 20%, some reaching 250%. Part of this is due to not using Copy-on-Write. When a page is first accessed, execution stops until a copy is made, regardless of whether the page will be modified or not. Copy-on-Write keeps this process in kernel space and only copies the modified pages, making our online operation unnoticeable from the users.

7 CONCLUSION AND FUTURE WORK

This paper introduced a crowd-sourced optimization approach for mobile devices, transparent to developers and users. Offline approaches rely on representative inputs that are hard to reconstruct, especially for interactive apps. Online approaches use real inputs, but expose users to slow or erroneous executions. On top of that, input variability makes the robust comparison of optimizations an extremely lengthy process.

Our approach tunes interactive apps using real inputs without disrupting the users. With infrequent captures, we store the input for targeted regions with little overhead. Later, we can replay each region on-demand, using the same context but different optimizations to compare them and discover the best ones, while the device is idle and charged.

We implemented a prototype system on Android, including a novel LLVM backend to increases the optimization space. We capture inputs once per optimization run. Our *Page* and *Intersection* mechanisms require less than 15ms overhead on average per capture, while *Everything* requires less than 5ms.

For this almost imperceptible effect on users, we achieve average speedups of 44% over Android and 35% over the LLVM -O3. With crowd-sourcing we achieve just 5% less of that at a fraction of the evaluation 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.

This work has produced some exciting results but there are more untapped opportunities for optimization. In the future we aim to evaluate our approach on more code sources. One problem that might arise, is that some of the computationally-intensive codes of the more complicated apps are re-written in native code (i.e., JNI). While our approach does not currently consider JNI, it can help avoiding it altogether, as it brings fine-tuning (automated or not) to the Java/Kotlin languages directly, reducing development time and increasing productivity.

To address input variation that concerns the same user, we can optimize for multiple captures. When such input changes happen between different days, we can simply perform periodic captures and optimize the code overnight. When those changes happen during the same day, we can follow two approaches. The first is to consider multiple captures and optimize for the average case. The second is to generate multiple binaries and employ multi-versioning. For the latter, we can maintain a set of pre-optimized binaries for the most frequently encountered inputs, and at runtime use a lightweight input-classification algorithm to predict and use the code version that will yield the best performance.

The findings of our joint search suggest that the input was significant for many cases when it came to code optimization. Despite being able to do a joint search and specialize for individual users when needed, we noticed that a few deviated a lot from others. We are considering detecting those scenarios and do non-shared searches (at least to some extent) to address this. We will also extend our GA search 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.

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. *Dalvik Bytecode: distibution format of Android code.* https://source.android.com/devices/tech/dalvik/dalvik-bytecode

[7] 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

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

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

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

[11] 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

[12] 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

[13] CDC. 2021. *CDC Sleep Statistics for the United States.* https://www.cdc.gov/sleep/data_statistics.html

[14] 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

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

[16] 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

[17] 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

[18] 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

[19] 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

[20] Peter Deutsch. 1996. RFC1951: DEFLATE compressed data format specification version 1.3. (1996).

[21] 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

[22] 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

[23] 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

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

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

[26] 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

[27] 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.*

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

[29] 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

[30] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization.* 242–255.

[31] 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

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

[33] 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

[34] 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

[35] 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.

[36] 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

[37] 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

[38] 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

[39] 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

[40] Linux. 2020. `/proc` *pseudo-filesystem on Linux*. https://www.kernel.org/doc/html/latest/filesystems/proc.html

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

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

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

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

[45] Paschalis Mpeis. 2020. *LLVM Backend for the Android compiler*. https://github.com/Paschalis/android-llvm

[46] 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*. 268–281.

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

[48] 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

[49] 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

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

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

[52] 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

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

[54] 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

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

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

[57] 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

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