Optimizing Function Layout for Mobile Applications

Ellis Hoag
ellishoag@meta.com
Meta Platforms Inc.
Menlo Park, CA, USA

Kyungwoo Lee
kyulee@meta.com
Meta Platforms Inc.
Menlo Park, CA, USA

Julián Mestre
julian.mestre@sydney.edu.au
University of Sydney
Australia
julianmestre@meta.com
Meta Platforms Inc.
Menlo Park, CA, USA

Sergey Pupyrev
spupyrev@meta.com
Meta Platforms Inc.
Menlo Park, CA, USA

Abstract

Function layout, also known as function reordering or function placement, is one of the most effective profile-guided compiler optimizations. By reordering functions in a binary, compilers can improve the performance of large-scale applications or reduce the compressed size of mobile applications. Although the technique has been extensively studied in the context of large-scale binaries, no study has thoroughly investigated function layout algorithms on mobile applications.

In this paper we develop the first principled solution for optimizing function layouts in the mobile space. To this end, we identify two key optimization goals: reducing the compressed code size and improving the cold start-up time of a mobile application. Then we propose a formal model for the layout problem, whose objective closely matches our goals. Our novel algorithm for optimizing the layout is inspired by the classic balanced graph partitioning problem. We have carefully engineered and implemented the algorithm in an open-source compiler, LLVM. An extensive evaluation of the new method on large commercial mobile applications demonstrates significant improvements in start-up time and compressed size compared to the state-of-the-art approach.

CCS Concepts: • **Software and its engineering** → *Compilers*; • **Theory of computation** → **Graph algorithms analysis**.

LCTES '23, June 18, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0174-0/23/06…$15.00
https://doi.org/10.1145/3589610.3596277

Keywords: profile-guided optimizations, code layout, function reordering, code-size reduction, graph algorithms

ACM Reference Format:
Ellis Hoag, Kyungwoo Lee, Julián Mestre, and Sergey Pupyrev. 2023. Optimizing Function Layout for Mobile Applications. In *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3589610.3596277

1 Introduction

As mobile applications become an essential part of everyday life, it becomes crucial to improve their speed, size, and reliability. Profile-guided optimization (PGO) is a critical component in modern compilers for improving the performance and size of applications; it enables the development and delivery of new app features for mobile devices that have limited storage and low memory. The technique, also known as feedback-driven optimization (FDO), leverages the program's dynamic behavior to generate optimized applications. PGO is now a standard feature in most commercial and open-source compilers.

Modern PGO has been successful in speeding up server workloads [7, 17, 36] by providing a double-digit percentage boost in performance. This is accomplished through a combination of several compiler optimizations, such as function inlining and code layout. PGO relies on execution profiles of a program, such as the execution frequencies of basic blocks and function invocations, to guide compilers in selectively and efficiently optimizing critical paths of a program. Typically, server-side PGO aims to improve CPU and cache utilization during the steady state of program execution, resulting in higher server throughput. However, applying PGO for mobile applications poses a unique challenge, as mobile applications are largely I/O bound and lack a well-defined steady-state performance due to their user-interactive nature [27]. Instead, the download speed and launch time of an app are crucial to its success, as they directly impact user experience, and therefore, user retention [5, 30].

In this paper we revisit a classic PGO technique, function layout, and show how to successfully apply it in the context of mobile applications. We emphasize that most of the earlier compiler optimizations focus on a single objective, such as the performance or the size of a binary. However, function layout might impact multiple key metrics of a mobile application. We show how to place functions in a binary to simultaneously improve its (compressed) size and start-up performance. The former objective is directly related to the app download speed and has been extensively discussed in recent works on compiler optimizations for mobile applications [5, 27, 28, 30, 40, 42]. The latter receives considerably less attention but nevertheless is of prime importance in the mobile space [10, 18, 19, 32].

Function layout, along with basic block reordering and inlining, is one of the most impactful PGOs. The seminal work of Pettis and Hansen [37] introduced a heuristic for function placement that reduces I-TLB (translation lookaside buffer) cache misses, improving the steady-state performance of large-scale binaries. The follow up work of Ottoni and Maher [35] further improved this placement scheme by considering the performance of the processor instruction cache (I-cache). The two heuristics are utilized in the majority of modern compilers and binary optimizers [25, 35, 36, 44]. However, these optimizations have not been widely used in mobile development and the corresponding layout algorithms have not been thoroughly studied. To the best of our knowledge, the recent work of Lee, Hoag, and Tillmann [27] is the only known study that describes a technique for function placement in native mobile applications. With this in mind, we provide *the first comprehensive investigation of function layout algorithms* in the context of mobile applications. In Section 1.1 we explain how function layout impacts the compressed app size, which eventually affects download speed. Then in Section 1.2 we describe how an optimized function placement can improve the start-up time. Finally, Section 1.3 highlights our main contributions, *a unified optimization model* to tackle these two seemingly unrelated objectives and *a novel algorithm* for the problem based on the recursive balanced graph partitioning.

1.1 Function Layout for App Download Speed

As mobile apps continue to grow rapidly, reducing the binary size is crucial for application developers [21, 27, 30]. Smaller apps can be downloaded faster, which directly impacts user experience [39]. For example, a recent study in [5] establishes a strong correlation between the app size and the user engagement. Furthermore, mobile app distribution platforms may impose size limitations for downloads that use cellular data [5]. For example, in the Apple App Store, users will not receive timely updates that include critical security improvements if an app's size exceeds a certain threshold, unless they are connected to a Wi-Fi network.

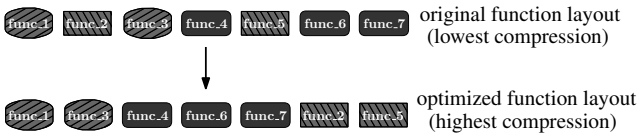


Figure 1. Placing similar (same-patterned) functions nearby in the binary leads to higher compression rates achieved by Lempel-Ziv algorithms. Functions are considered similar when they share common sequences of instructions that can be encoded by short references.

Mobile apps are distributed to users in a compressed form via mobile app platforms. Typically, application developers do not have control over the compression technique used by the platforms. However, Lee et al. [27] observe that modifying the content of a binary can improve its compressed size. Specifically, co-locating "similar" functions in the binary can improve the compression ratio achieved by popular compression algorithms such as ZSTD or LZFSE. A similar technique is used in a bytecode Android optimizer, Redex [20].

Why does function layout affect compression ratios? Most modern lossless compression tools rely on the Lempel-Ziv (LZ) scheme [48]. Such algorithms try to identify long repeated sequences in the data and substitute them with pointers to their previous occurrences. If the pointer is represented using fewer bits than the actual data, then the substitution results in a compressed-size win. That is, the shorter the distance between the repeated sequences, the higher the compression ratio. To make the computation effective, LZ-based algorithms search for common sequences inside a *sliding window*, which is typically much shorter than the actual data. Therefore, function layouts in which repeated instructions are grouped together, lead to smaller (compressed) mobile apps; see Figure 1 for an example.

1.2 Function Layout for App Launch Time

Start-up time is one of the key metrics for mobile applications since a quick launch ensures users have a good first impression [47]. According to a study in [32], 20% of users abandon an app after one use, and 80% of users give poorly performing apps at most three chances before uninstalling them. Start-up time is the time between a user clicking on an application icon and the display of the first frame after rendering. There are several start-up scenarios: cold start, warm start, and hot start [10, 18, 19]. Switching back and forth between different apps leads to a hot/warm start and typically does not incur significant delays. In contrast, starting an app from scratch or resuming it after a memory intensive process is referred to as cold start. Our focus is to improve this cold start scenario, which is usually the key performance metric.

Unlike server workloads, where code layout algorithms optimize the cache utilization [34, 35, 36], start-up performance is mostly dictated by memory page faults [12]. When an app

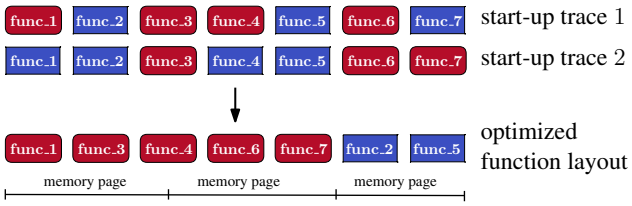


Figure 2. Co-locating hot (round) functions and cold (rectangular) functions nearby in the binary leads to a reduction in page faults. The hotness of the functions and their order of executions might depend on the usage scenario (trace), and the task is to find a single optimized function layout.

is launched, its code is transferred from the disk to the main memory. Function layout can affect the performance because the transfer happens at the granularity of memory pages. As illustrated in Figure 2, interleaving cold functions that are never executed with hot functions results in more memory pages being fetched from disk. While simply grouping hot functions in the binary is an attractive solution, we note that some mobile apps have a user base of billions of daily active users across a wide range of devices and platforms. As a result, optimizing the layout for a particular usage scenario could lead to a suboptimal performance for other scenarios. The challenge is to produce a single function layout that optimizes the start-up performance across all use cases.

1.3 Contributions

We model the problem of computing an optimized function layout for mobile apps as the balanced graph partitioning problem [14]. This approach enables a single algorithm to enhance both app start-up time (which impacts user experience) and app size (which impacts download speed). However, while the layout algorithm is the same for both objectives, it operates with different datasets collected during profiling. For the sake of clarity, we call the optimizations Balanced Partitioning for Start-up Optimization (**bps**) and Balanced Partitioning for Compression Optimization (**bpc**). Algorithm 1 outlines our implementation.

The former optimization, **bps**, is applied to *hot* functions in the binary that are executed during app start-up, while the latter optimization, **bpc**, is applied to all the remaining *cold* functions. In our experiments, we found that approximately 15% of functions are hot, allowing us to improve the overall start-up performance while simultaneously reordering most of the functions in a "compression-friendly" manner. Compared to the prior work [27], we achieved an average start-up time improvement of 4% and a compressed size reduction of up 1%, while speeding up the function layout phase by 30 times faster for `SocialApp`, one of the largest mobile apps in the world. The contributions of the paper are summarized as follows.

- We formally define the function layout problem in the context of mobile applications. To this end, we identify and formalize two optimization objectives, based on the application start-up time and the compressed size.
- Next, we present the Balanced Partitioning algorithm, which takes as input a bipartite graph between *function* and *utility* vertices, and outputs an order of the function vertices. We also demonstrate how to reduce the objectives of **bpc** and **bps** to an instance of the balanced graph partitioning problem.
- Finally, we extensively evaluate the compressed size, the start-up performance, and the runtime of the new algorithms with two large commercial iOS applications, `SocialApp` and `ChatApp`, and experiment with app size on Android native binaries.

The rest of the paper is organized as follows. Section 2 builds an optimization model for compression and start-up performance, respectively. Then Section 3 introduces the recursive balanced graph partitioning algorithm, which forms the foundation for effectively solving the two optimization problems. Next in Section 4, we describe our implementation of the technique in an open-source compiler, LLVM. Section 5 presents an evaluation on real-world mobile applications. We conclude the paper with a discussion of related works in Section 6 and possible future directions in Section 7.

2 Building an Optimization Model

We model the function layout problem with a bipartite graph, denoted $G = (F \cup U, E)$, where F and U are disjoint sets of vertices and E are the edges between the sets. The set F is a collection of all *functions* in a binary, and the goal is to find a permutation (also called an *order* or a *layout*) of F. The set U represents auxiliary *utility* vertices that are used to define an objective for optimization. Every utility vertex $u \in U$ is adjacent with a subset of functions, $f_1, \ldots, f_k \in F$ so that $(u, f_1), \ldots, (u, f_k) \in E$ for some integer $k \geq 2$. Intuitively, the goal of the layout algorithm is to place all functions so that $f_1, \ldots, f_k$ are nearby in the resulting order, for each utility vertex u. That is, the utility vertex encodes a locality preference for the adjacent functions. Next, we formalize the intuition for each of the two objectives.

2.1 Compression

As explained in Section 1.1, the compression ratio of a Lempel-Ziv-based algorithm can be improved if similar functions are placed nearby in the binary. This observation is based on earlier theoretical studies [38] and has been confirmed empirically [13, 29] in the context of lossless data compression. These studies define (sometimes, implicitly) a proxy metric that correlates an order of functions with the compression achieved by an LZ scheme. Suppose we are given some data to compress, e.g., a sequence of bytes that represents the instructions in a binary. We define a *k-mer* to

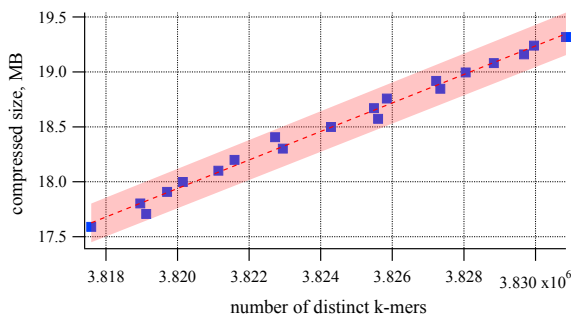


Figure 3. The correlation between the number of distinct k-mers ($k = 8$) in a sliding window of size $w = 64$KB for `ChatApp` and its compressed size after applying a Lempel-Ziv-based (`LZ4`) compression algorithm

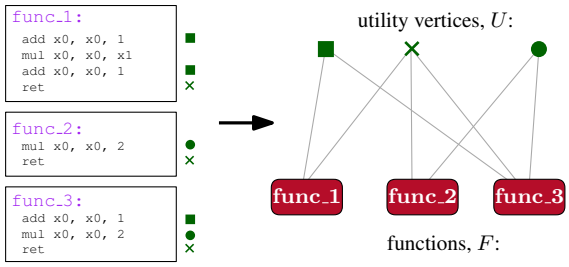


Figure 4. Modeling compression-aware function layout (**bpc**) with a bipartite graph

be a contiguous substring in the data of length k, which is a small constant. Let w be the size of the sliding window utilized by the compression algorithm; typically, w is much smaller than the length of the data. Then the compression ratio attained by an LZ-based data compression algorithm is determined by the number of distinct k-mers in the data within each sliding window of size w. In other words, the compressed size of the data is minimized when each k-mer occurs in as few windows of size w as possible.

To validate the intuition, we computed and plotted the number of distinct 8-mers within 64KB-windows on a set of functions from `ChatApp`; see Figure 3. To obtain a data point for the plot, we fixed a specific layout of functions in the binary and extracted its `.text` section to a string, by concatenating their instructions. Then for every (contiguous) substring of length w, we counted the number of distinct k-mers in the substring. This number serves as the proxy metric for predicting the compressed size of the data. We then applied a compression algorithm to the entire string and measured the compressed size. To get multiple points on Figure 3, we repeated the process by starting with a different function layout, which was achieved by randomly permuting some of the functions. The results in Figure 3 reveal a strong correlation between the actual compression ratio achieved on the data and the predicted value based on k-mers. We recorded a Pearson correlation coefficient of $\rho > 0.95$ between the two quantities. Interestingly, the high correlation was observed for various values of k (in our evaluation, $4 \le k \le 12$), different window sizes (4KB $\le w \le$ 128KB), and various compression tools, including ZSTD (which combines a dictionary-matching stage with a fast entropy-coding stage), `LZ4` (which belongs to the LZ77 family of byte-oriented compression schemes), and `LZMA` (which uses dictionary compression within the xz tool).

Given the remarkable predictive power of the simple proxy metric, we suggest optimizing the layout of functions in a binary based on the metric since it can be easily extracted and computed from the data. To achieve this, we represent each function, $f \in F$, as a sequence of instructions. For each instruction in the binary that occurs in at least two functions, we create a utility vertex $u \in U$. The bipartite graph, $G = (F \cup U, E)$, contains an edge $(f, u) \in E$ if function f contains instruction u; refer to Figure 4 for an illustration of the process. The goal is to co-locate functions that share many utility vertices so that the compression algorithm can efficiently encode the corresponding instructions.

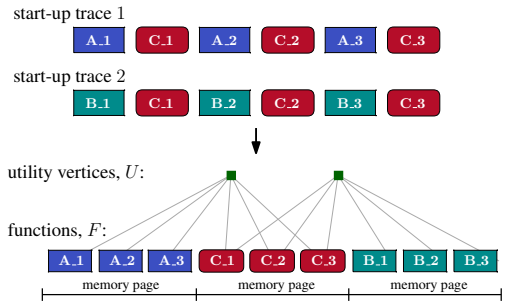
2.2 Start-up

To optimize cold start, we develop a simplified memory model. Initially, we assume that the application code is not present in the main memory. When the application starts, the code needs to be fetched from the disk to the main memory at the granularity of memory *pages*. We assume that the pages are never evicted from the memory. That is, when a function is executed for the first time, its page should be present in the memory to avoid a start-up delay caused by page faults. The goal is to find a function layout that results in the fewest number of page faults possible.

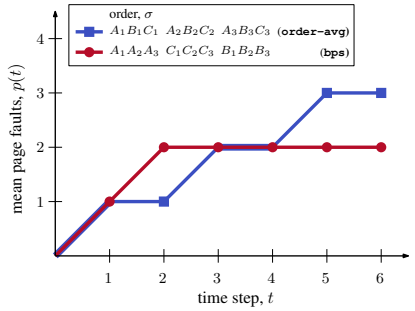
In this model, the start-up performance is affected only by the first execution of a function; all subsequent executions do not result in page faults. Hence, we record the timestamp when each function $f \in F$ was first executed, and collect the sequence, called the *trace*, of functions ordered by the timestamps. The traces vary depending on the user or usage scenario of the application. We assume that we have a representative collection of traces, S.

Given an order of functions, we can determine which memory page each function belongs to, given their sizes and assuming a certain page size. Then for every start-up trace, $\sigma \in S$, and an index $t \le |\sigma|$, we define $p_\sigma(t)$ to be the number of page faults during the execution of the first t functions in σ. Similarly, for a set of traces S, we define the *evaluation curve* as the average number of page faults for each $\sigma \in S$, that is, $p(t) := \sum_{\sigma \in S} p_\sigma(t)/|S|$.

To gain an intuition, consider what happens when there is only one trace $\sigma \in S$. In this case, the optimal layout is to use the order induced by σ, which results in an evaluation curve that is linear in t. On the other hand, a random permutation



(a) Bipartite graph construction with a single threshold



(b) Evaluation curves for two orders: **bps** and **order-avg** ([27])

Figure 5. Modeling start-up-aware function layout (**bps**)

of functions causes most pages to be fetched early in the execution, resulting in an evaluation curve that looks like a step function. Figure 5b provides a concrete example of how different layouts lead to different evaluation curves.

We remark that while traces have the same length if all functions are executed eventually, the length of the prefix of each trace corresponding to the start-up phase of the execution may vary due to diverging execution paths specific to the device and the user. Hence, instead of optimizing the value of $p(t)$ for a particular t, we aim to minimize the area under the curve $p(t)$ by selecting a discrete set of *threshold* values $t_1, t_2, \ldots t_k$, and use the bipartite graph $G = (F \cup U, E)$ with utility vertices

$$U = \{(\sigma, t_i) : \sigma \in S \text{ and } 1 \le i \le k\},$$

and edge set

$$E = \{(f, (\sigma, t_i)) : \sigma^{-1}(f) \le t_i\},$$

where $\sigma^{-1}(f)$ is the index of function f in σ. That way, the algorithm builds an order of F in which the first t_i positions of every $\sigma \in S$ occur, as much as possible, consecutively.

3 Recursive Balanced Graph Partitioning

Our algorithm utilizes the recursive balanced graph partitioning scheme. Recall that the input is an undirected bipartite graph $G = (F \cup U, E)$, where F and U are disjoint sets of functions and utilities, respectively, and E are the edges between them; see Figure 6a. The goal of the algorithm is to find a permutation of F that optimizes a specific objective.

For a high-level overview of our method, refer to Algorithm 1. The algorithm combines recursive graph bisection with a local search optimization at each step. Given an input graph G with $|F| = n$, we apply the bisection algorithm to obtain two disjoint sets of (approximately) equal cardinality, $F_1, F_2 \subseteq F$, where $|F_1| = \lfloor n/2 \rfloor$ and $|F_2| = \lceil n/2 \rceil$. We layout F_1 on the set $\{1, \ldots, \lfloor n/2 \rfloor\}$ and F_2 on the set $\{\lceil n/2 \rceil, \ldots, n\}$. By doing so, we divide the problem into two sub-problems, each with half the size, and recursively compute orders for the two subgraphs induced by vertices F_1 and F_2, adjacent utility vertices, and incident edges. Naturally, when the graph contains only one function, the order is trivial; see Figure 6b.

Every bisection step of Algorithm 1 is a variant of the local search optimization inspired by the popular Kernighan-Lin heuristic [23] for the graph bisection problem. We start by splitting F into two sets, F_1 and F_2, of roughly equal size. Then, we iteratively exchange pairs of vertices between F_1 and F_2 to improve a certain *cost*. To this end, we compute, for every function $f \in F$, the *move gain*, that is, the difference of the cost after moving f from its current set to another one. Then the vertices of F_1 (F_2) are sorted in the decreasing order of the gains to produce list S_1 (S_2). Finally, the lists S_1 and S_2 are traversed in the order, exchanging the pairs of vertices when the sum of the move gains is positive. The process is repeated until a convergence criterion is met or the maximum number of iterations is reached. The final order of the functions is obtained by concatenating the two recursively computed orders for F_1 and F_2.

Optimization objective. An important aspect of our algorithm is the objective to optimize at each bisection step. The goal is to find a layout in which functions sharing many utility vertices are co-located in the order. We capture this with the *cost* of a given partition of F into F_1 and F_2:

$$cost(F_1, F_2) := \sum_{u \in U} cost\big(L(u), R(u)\big), \tag{1}$$

where $L(u)$ and $R(u)$ are the numbers of functions adjacent to utility vertex u in parts F_1 and F_2, respectively; see Figure 6a. Observe that $L(u) + R(u)$ is the degree of vertex u, and thus, it is independent of the split. The objective, which we try to *minimize*, is the summation of the individual contributions to the cost over the utilities. The contribution of one utility vertex, $cost\big(L(u), R(u)\big)$, is minimized when $L(u) = 0$ or $R(u) = 0$, that is, when all functions of u belong to the same part; in that case, the algorithm might be able to group the functions in the final order. In contrast, when $L(u) \approx R(u)$, the cost takes its highest value, as the functions will likely be spread out in the order. Of course, it is easy to minimize the cost for one utility vertex (by placing its functions to one of the parts). However, minimizing $cost(F_1, F_2)$

Algorithm 1: Recursive Balanced Graph Partitioning

Input : graph $G = (F \cup U, E)$
Output: order of F vertices

Function `ReorderBP`
  `/* Initial splitting of the functions into two halves. */`
  for $f \in F$ **do**
    if $random(0,1) < 0.5$ **then** $F_1 \leftarrow F_1 \cup \{f\}$
    else $F_2 \leftarrow F_2 \cup \{f\}$;
  `/* Refinement of the split. */`
  repeat
    for $f \in F$ **do**
      $gains[f] \leftarrow$ `ComputeMoveGain`(f)
    $S_1 \leftarrow$ sorted F_1 in descending order of $gains$;
    $S_2 \leftarrow$ sorted F_2 in descending order of $gains$;
    for $v \in S_1, u \in S_2$ **do**
      if $gains[v] + gains[u] > 0$ **then**
        exchange v and u in the sets;
      else break;
  until *converged* **or** *iteration limit exceeded*;
  `/* Recursively reorder the two parts and concatenate the results. */`
  $Order_1 \leftarrow$ `ReorderBP`(Graph induced by $F_1 \cup U$);
  $Order_2 \leftarrow$ `ReorderBP`(Graph induced by $F_2 \cup U$);
  return *concatenation of* $Order_1$ *and* $Order_2$

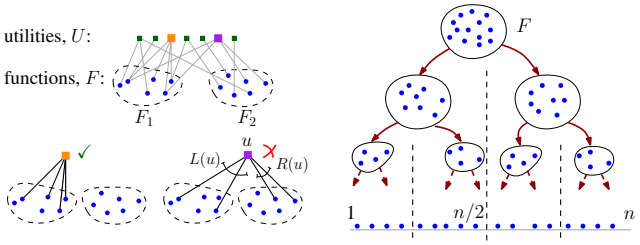
Function `ComputeMoveGain`(f)
  `/* Calculate cost improvement after moving` f `to another part. */`
  $gain = 0$;
  for $(u, f) \in E$ **do**
    if $f \in F_1$ **then**
      $gain \leftarrow gain + cost\big(L(u), R(u)\big) - cost\big(L(u) - 1, R(u) + 1\big)$;
    else
      $gain \leftarrow gain + cost\big(L(u), R(u)\big) - cost\big(L(u) + 1, R(u) - 1\big)$;
  return *gain*

for all utilities simultaneously is a challenging task, due to the constraint on the sizes of F_1 and F_2.

There are multiple ways of defining $cost\big(L(u), R(u)\big)$ that satisfy the conditions above. After an extensive evaluation of various candidates, we identified the following objective:

$$cost := -L(u) \log\big(L(u) + 1\big) - R(u) \log\big(R(u) + 1\big), \quad (2)$$

which is inspired by the so-called *uniform log-gap cost* utilized in the context of index compression [8, 11].



(a) An optimization goal for the bipartite graph

(b) Recursive computation of function orders

Figure 6. The recursive balanced graph partitioning

Computational complexity. To estimate the computational complexity of Algorithm 1 and predict its running time, denote $|F| = n$ and $|E| = m$. At each bisection step, we apply a constant number of refinement steps (referred to as the *iteration limit* in the pseudocode). There are $\lceil \log n \rceil$ levels of recursion, and we assume that every call of `ReorderBP` splits the graph into two equal-sized parts with $n/2$ vertices and $m/2$ edges. Each call of the graph bisection consists of computing move gains and sorting two arrays with n elements. The former can be done in $O(m)$ steps, while the latter takes $O(n \log n)$ steps. Therefore, the total number of steps is expressed as follows:

$$T(n, m) = O(m) + O(n \log n) + 2 \cdot T(n/2, m/2).$$

One can verify that summing over all subproblems yields $T(n, m) = O(m \log n + n \log^2 n)$.

3.1 Algorithm Engineering

While implementing Algorithm 1, we developed a few modifications improving its runtime, space requirements, and the quality of produced layouts.

Improving the running time. Due to the simplicity of the algorithm, it can be implemented to run in parallel. Since the two subgraphs resulting from the bisection step are disjoint, the two recursive calls can be processed concurrently. We use the fork-join computation model, where small enough graphs are processed sequentially, while larger graphs are solved in parallel. To speed up the algorithm further, we set a maximum depth of the recursive tree (16 in our implementation) and limit the number of local search iterations per split (20 in our implementation). If the recursive tree reaches the lowest node and there are still unordered functions, then we fall back to the original relative order of the functions provided by the compiler.

Finally, we observe that our objective cost requires repeated computation of $\log(x + 1)$ expressions for integer arguments. To avoid costly floating-point logarithm evaluations, we pre-computed a table of values for $0 \le x < 2^{14}$, where the upper bound is chosen small enough to fit in the

processor data cache. That way, we replaced most of the logarithm evaluations with a table lookup, saving approximately 10% of the total runtime.

Optimizing the quality. An interesting aspect of Algorithm 1 is the way for exchanging functions between the two sets. Recall that we pair the functions in F_1 with functions in F_2 based on the computed move gains, which are positive when a function should be moved to another set or negative when a function should stay in its current set. We observed that it is beneficial to skip some of the moves. To this end, we introduce a fixed probability (0.1 in our implementation) of skipping the move for a vertex that would otherwise have been moved to a new set. Intuitively, this adjustment prevents the optimization from becoming stuck at a local minimum. It is also helpful in avoiding redundant swapping cycles, which might occur in the algorithm; refer to [31, 46] for a discussion in the context of graph reordering.

Reducing the space complexity. One potential downside to our start-up function layout algorithm is the need to collect full traces during profiling. If too many executions are profiled, then the storage requirements may become impractical. To address the issue, we cap the number of stored traces by a fixed integer ℓ. If the profiling process generates more than ℓ traces, we select a representative random sample of size ℓ using reservoir sampling [45]: When the ith trace arrives, if $i \leq \ell$ we keep the trace; otherwise, with probability $1 - \ell/i$ we ignore the trace, and with complementary probability, we pick uniformly at random one of the stored traces and swap it out with the new one. The process yields a sample of ℓ traces chosen uniformly at random from the stream of traces. We use $\ell = 300$ in our implementation.

4 Implementation in LLVM

Both **bpc** and **bps** use profile data to guide function layout. Ideally, profile data should accurately represent common real-world scenarios. The current instrumentation in LLVM [24] produces an instrumented binary with large size and performance overhead due to added instrumentation instructions, added metadata sections, and changes in optimization passes. This can be particularly problematic for mobile devices, where increased code size can lead to performance regressions and alter the behavior of the application. Profiles collected from these instrumented binaries might not accurately represent our target scenarios. To address these issues, Machine IR Profile (MIP) [27] aims to minimize binary size and performance overhead for instrumented binaries. This is achieved by extracting instrumentation metadata from the binary and using it to post-process the profiles offline.

MIP collects profile data that are relevant for optimizing mobile apps. It records function call counts used to identify functions as either hot or cold. Within each function, MIP can derive coverage data for each basic block. MIP has an optional mode, called return address sampling, which adds probes to callees to collect a sample of their callsites. This can be used to construct a dynamic call graph that includes dynamically dispatched calls. Furthermore, MIP collects function timestamps by recording and incrementing a global timestamp for each function when it is called for the first time. We sort the functions by their initial call timestamp to construct a function trace. To collect raw profiles at runtime, we run instrumented apps under normal usage, and dump raw profiles to the disk, which is uploaded to a database. These raw profiles are later merged offline into a single optimization profile.

4.1 Overview of the Build Pipeline

Figure 7 shows an overview of our build pipeline. We collect thousands of raw profile data files from various uses and periodically perform offline post-processing to generate a single optimization profile. During post-processing, **bps** determines the optimized order of hot functions that were profiled, including both start-up and non-start-up functions. Our apps are built with link-time optimization (LTO or ThinLTO). At the end of LTO, **bpc** orders cold functions to achieve a highly compressed binary size. These two orders of functions are concatenated and passed to the linker which finalizes the function layout in the binary. We have chosen to use two separate optimization passes for **bps** and **bpc**, since applying them jointly at the end of LTO would require carrying large amounts of traces through the build pipeline.

4.2 Hot Function Layout

As shown in Figure 7, we first merge the raw profiles into the optimization profile with instrumentation metadata during post-processing. For the block coverage and dynamic call graph data, we simply accumulate them into the optimization profile as we go along. However, to run **bps**, we need to keep the function timestamps from each raw profile. We encode the sequence of indices to the functions participating in the cold start-up and append them to a separate section of the optimization profile.

The **bps** algorithm, described in Section 2.2, uses function traces with thresholds, to set utility vertices, and produces an optimized order for start-up functions. Once **bps** is completed, the embedded function traces are no longer needed, and can be removed from the optimization profile.

Third-party library functions and outlined functions that appear later than instrumentation in the compilation pass, might not be instrumented. To order such functions, we first check if their call sites are profiled using block coverage data. If so, these functions inherit the order of their first caller. For example, if an uninstrumented outlined function, $f_{outlined}$, is called from the profiled functions, f_A and f_B, and **bps** orders f_A followed by f_B, then we insert $f_{outlined}$ after f_A; this results in the layout $f_A, f_{outlined}, f_B$.

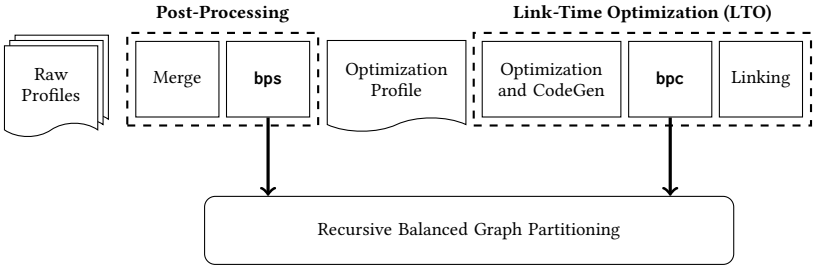


Figure 7. An overview of the build pipeline with the optimized function layout

4.3 Cold Function Layout

We execute **bpc** after optimization and code generation are finished, without using an intermediate representation (IR), as shown in Figure 7. During code generation, each function publishes a set of hashes that represent its contents, which are meaningful across modules. We use one 64-bit *stable hash* for each instruction by combining hashes of its opcode and operands, resulting in an 8-mer, a substring of length 8, for every instruction. When computing stable hashes, we omit hashes of pointers and instead use hashes of the contents of their targets. Unlike outliners that need to match instruction sequences, we do not consider the order or duplicates of the hashes. We only keep track of the unique stable hashes per function as the input to **bpc**.

Since hot functions are already ordered, we filter them out before applying **bpc**. It is worth noting that outliners can optimistically produce many identical functions, which will eventually be folded by the linker. To efficiently handle deduplication, **bpc** groups functions that have identical sets of hashes, and runs with the set of unique cold functions.

5 Evaluation

We evaluated our approach on two commercial iOS applications and one commercial Android application; refer to Table 1 for basic properties of the apps. SocialApp is one of the largest mobile applications in the world, with a total size of over 250MB, and provides a variety of usage scenarios, making it an attractive target for compiler optimizations. ChatApp is a medium sized mobile app with a total size of over 50MB. AndroidNative consists of around 400 shared natives binaries. Unlike the two iOS binaries that are built with ThinLTO, each Android native binary is relatively small, and can be compiled with (*Full*)LTO without a significant increase in build time. Since there is no fully automated MIP pipeline for building AndroidNative, we use the app only to evaluate the compressed binary size.

5.1 Start-up Performance

Here we present the impact of function layout on start-up performance. Our proposed algorithm, referred to as **bps**, is compared with the following alternatives:

Table 1. Basic properties of evaluated applications

	text size (MB)	binary size (MB)	total func.	hot func.	blocks per func. p50	p95	p99
SocialApp	119	259	856K	154K	1	11	29
ChatApp	35	58	202K	44K	3	24	70
AndroidNative	38	62	186K	N/A	3	36	113

- **`baseline`** is the original ordering of functions as dictated by the compiler; the function layout follows the order of object files that are passed into the linker;
- **`random`** is the result of randomly permuting the hot functions;
- **`order-avg`** is a natural heuristic for ordering hot functions suggested in [27] based on the average timestamp of a function during start-up computed across all traces.

To evaluate the impact of function layout in a production environment, we compared the current **`order-avg`** algorithm with the new **bps** algorithm for two different release versions, release N and release $N+1$, and recorded the number of page faults during start-up. Table 2 presents the detailed results for the average and 99th percentile number of page faults observed in millions of samples published in production. Since in the production environment for iOS apps, only a single binary can be shipped, and less effective algorithms (such as **`baseline`** or **`random`**) cannot be utilized without regressing performance, we acknowledge that the improvements observed may result from multiple optimizations shipped simultaneously with **bps**. To account for this, we repeated the alternations three times in consecutive releases, and recorded the overall reduction in page faults. On average, **bps** reduced the number of major page faults by 6.9% and 16.9% for SocialApp and ChatApp, respectively. The improvement translates into 4.2% (average) and 2.9% (p99) reductions of the cold start-up time for SocialApp.

Table 3 presents the results of a similar evaluation of the start-up performance with different function layouts on a specific device, *iPhone*12 *Pro*, during the first 10s of the cold start-up. We repeat the experiment three times and calculated the mean number of page faults in (i) the `.text` segment and (ii) the entire binary, which additionally includes other data

Table 2. The number of major page faults measured for **order-avg** and **bps** shipped in consecutive releases. The relative improvement of **bps** over **order-avg** is shown in parentheses.

		average	p99
SocialApp			
order-avg	release N	3.4K	7.6K
bps	release $N+1$	3.1K (6.9%)	7.2K (4.6%)
ChatApp			
order-avg	release N	1.7K	10.3K
bps	release $N+1$	1.4K (16.9%)	9.3K (9.1%)

Table 3. The relative improvements of major page faults of various function layout algorithms over **baseline** measured on *iPhone12 Pro* during the first 10s of the cold start-ups; negative values indicate regressions.

	Text	Binary
SocialApp		
random	−4.7%	−3.3%
order-avg	14.1%	3.2%
bps	19.9%	6.8%
ChatApp		
random	−144.4%	−65.7%
order-avg	34.9%	15.9%
bps	36.6%	18.3%

segments. Overall, **bps** reduces the total number of major page faults in the binary by 6.8% and 18.3% for SocialApp and ChatApp, respectively, while **order-avg** reduces them by 3.2% and 15.9%, respectively. On the other hand, **random** significantly increases the number of page faults, negatively impacting the start-up performance.

An interesting observation from Tables 2 and 3 is that function layout has a greater impact on the start-up performance of ChatApp than that of SocialApp. This could be due to that SocialApp consists of dozens of native binaries, while function layout is effective within each binary. Therefore, optimizing the function layout across different binaries has a more limited impact on the overall start-up performance of SocialApp. In contrast, ChatApp is composed of only a few native binaries and only one large binary is responsible for the start-up; thus, an optimized function layout can directly impact the performance.

5.2 Compressed Binary Size

We now present the results of size optimizations on the selected applications. In addition to the **baseline** and **random** function layout algorithms, we compare **bpc** with the following heuristic:

- **greedy** is an approach for ordering cold functions discussed in [27]. It is a procedure that iteratively builds an order by appending one function at a time. On each step, the most recently placed function is compared (based on the instructions) with not yet selected functions, and the one with the highest similarity score is appended to the order. To avoid an expensive $O(n^2)$-computation of the scores, several pruning rules is applied to reduce the set of candidates; see [27] for details.

Table 4 summarizes the app size reduction from each function layout algorithm, where the improvements are computed on top of **baseline** (that is, the original order of functions generated by the compiler). The compressed size reduction is measured in three modes: the size of the `.text` section of the binary directly impacted by our optimization, the size of the executables excluding resource files such as images and videos, and the total app size in a compressed package. We observe that **bpc** reduces the size of `.text` by 3% and 1.8% for SocialApp and ChatApp, respectively. Since this section is the largest in the binary (responsible for 2/3 of the compressed ipa size), this translates into overall 1.9% and 1.3% improvements. At the same time, the impact of all the tested algorithm on the uncompressed size of a binary is minimal (within 0.1%), which is mainly due to differences in code alignment. We stress that while the absolute savings may feel insignificant, this is a result of applying a single compiler optimization on top of the heavily tuned state-of-the-art techniques; the gains are comparable to those reported by other recent works in the area [9, 27, 28, 30, 40].

An interesting observation is the behavior of **random** on the dataset, which worsen the compression ratios by approximately 5% in comparison to **baseline**. The explanation is that similar functions are naturally clustered in the source code. For example, functions within the same object file tend to have many local calls, making the corresponding call instructions good candidates for a compact LZ-based encoding. Yet **bpc** can significantly improve the instruction locality by reordering functions across different object files.

Additionally, we assess the reduction in compressed size for AndroidNative. It is important to note that the total app size is measured using Android package kit (apk) which includes not only native binaries, but also Android Dex bytecode. Unlike the aforementioned two iOS apps, the `.text` size of the native binaries is only 1/4 of the total app size. Therefore, the overall app compressed size win is smaller than for the `.text` or the executable sections. We observe that these compressed sizes for AndroidNative are more sensitive to different layouts. This is due to AndroidNative being a traditional C/C++ binary, where the number of blocks per function is significantly higher than that of the iOS apps, as illustrated in Table 1. Function call instructions encode their call targets with relative offsets whose values differ for each call-site. Unlike the iOS apps written in Objective-C having many dynamic calls, AndroidNative has fewer call-sites, making it more compression-sensitive.

Table 4. Compressed size improvements of various function layout algorithms over **`baseline`**; negative values indicate regressions.

	Text	Executables	App Size
SocialApp			
`random`	−5.3%	−4.6%	−3.7%
`greedy`	1.6%	1.3%	1.1%
`bpc`	3.0%	2.3%	**1.9%**
ChatApp			
`random`	−4.9%	−4.4%	−3.8%
`greedy`	1.3%	1.0%	0.8%
`bpc`	1.8%	1.6%	**1.3%**
AndroidNative			
`random`	−10.0%	−8.2%	−3.2%
`greedy`	3.5%	1.9%	0.9%
`bpc`	5.2%	3.0%	**1.3%**

Finally, we discuss the impact of function layout on the build time of the applications. The time overhead by running **`bpc`** is minimal: it takes less than 20 seconds for the larger SocialApp and almost 1 second for the smaller ChatApp. In contrast, using the **`greedy`** approach leads to a noticeable slowdown, increasing the overall build of SocialApp by around 10 minutes, accounting for more than 10% of the total build time of approximately 100 minutes.

6 Related Work

Most compiler optimizations for mobile applications are aimed at reducing the code size. Such techniques include algorithms for function inlining and outlining [9, 28], merging similar functions [41, 42], loop optimization [40], unreachable code elimination, and many others. In addition, some works describe performance improvements for mobiles, by improving their responsiveness, memory management, and start-up time [27, 47]. The optimizations can be applied at the compile time, link time [30], or post-link time [36, 44]. Our approach is complimentary to the works and can be applied in combination with the existing optimizations.

The work by Pettis and Hansen [37] serves as the basis for most modern code reordering techniques for server workloads. The goal of their basic block reordering is to create chains of blocks that are frequently executed together. Many variants of the technique were suggested in the literature and implemented in various tools [25, 33, 34, 35, 36, 43, 44]. Alternative models have been studied in several papers [15, 16, 22, 26], where a temporal-relation graph is considered.

Code reordering at the function-level is also initiated by Pettis and Hansen [37] whose algorithm is implemented in many compilers and binary optimization tools [36, 44]. This approach greedily merges chains of functions and is designed to primarily reduce I-TLB misses. An improvement is proposed by Ottoni and Maher [35], who propose working with a directed call graph to reduce I-cache misses. As discussed in Section 1, the approaches are designed to improve the steady-state performance of server workloads and cannot be applied to mobile apps. The very recent work of Lee, Hoag, and Tillmann [27] is the only study discussing heuristics for function layout in the mobile space; our novel algorithm significantly outperforms their heuristics.

Our model for function layout is based on the balanced graph partitioning problem [2, 14, 23]. There exists a rich literature on the topic from both theoretical and practical points of view [3, 4]. The most closely related work to our study is on graph reordering [11, 31], which utilizes recursive graph bisection for creating "compression-friendly" inverted indices. While our algorithm shares some similarities with these works, our objectives and application area are different.

7 Discussion

In this paper we have presented and evaluated the first function layout algorithm designed for mobile compiler optimizations. The algorithm was carefully designed, making it easy to implement and scalable to process even the largest instances within a matter of seconds. We have successfully applied this optimization to several large commercial mobile applications, resulting in significant improvements in start-up performance and reductions in app size.

An important contribution of the work is a formal model for function layout optimizations. We believe that the model utilizing the bipartite graph with utility vertices is general enough to be applicable in various contexts. In our current implementation, each function is either optimized for start-up or for size, but not for both at the same time. However, it might be possible to relax the constraint and design an approach that unifies the two objectives. Our early experiments show that reordering all functions with **`bpc`** could result in up to 0.3% size reduction, but this may come at the cost of a longer start-up time. Unifying the optimizations is a promising direction for future work.

From a theoretical point of view, our work is related to a computationally hard problem of balanced graph partitioning [2]. While the problem is hard in theory, real-world instances obey certain characteristics, which may simplify the analysis of algorithms. For example, control-flow and call graphs arising from modern programming languages have constant *treewidth*, which is a standard notion to measure how close a graph is to a tree [1, 6, 33]. Many NP-hard optimization problems can be solved efficiently on graphs with a small treewidth, and therefore, exploring function layout algorithms parameterized by the treewidth is of interest.

Acknowledgments

We would like to thank Nikolai Tillmann for fruitful discussions of the problem, and YongKang Zhu for helping with evaluating the approach on AndroidNative.

References

[1] Ali Ahmadi, Majid Daliri, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. 2022. Efficient approximations for cache-conscious data placement. In *International Conference on Programming Language Design and Implementation*, Ranjit Jhala and Isil Dillig (Eds.). ACM, San Diego, CA, USA, 857–871. https://doi.org/10.1145/3519939.3523436

[2] Konstantin Andreev and Harald Räcke. 2006. Balanced graph partitioning. *Theory of Computing Systems* 39, 6 (2006), 929–939. https://doi.org/10.1007/s00224-006-1350-7

[3] Charles-Edmond Bichot and Patrick Siarry. 2013. *Graph Partitioning*. John Wiley & Sons. https://doi.org/10.1002/9781118601181

[4] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. Recent Advances in Graph Partitioning. In *Algorithm Engineering - Selected Results and Surveys*. Springer, Cham, 117–158. https://doi.org/10.1007/978-3-319-49487-6_4

[5] Milind Chabbi, Jin Lin, and Raj Barik. 2021. An Experience with Code-Size Optimization for Production iOS Mobile Applications. In *International Symposium on Code Generation and Optimization*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, Seoul, South Korea, 363–377. https://doi.org/10.1109/CGO51591.2021.9370306

[6] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Nastaran Okati, and Andreas Pavlogiannis. 2019. Efficient parameterized algorithms for data packing. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–28. https://doi.org/10.1145/3290366

[7] Dehao Chen, Tipp Moseley, and David Xinliang Li. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *International Symposium on Code Generation and Optimization*. ACM, New York, NY, USA, 12–23. https://doi.org/10.1145/2854038.2854044

[8] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. 2009. On compressing social networks. In *Knowledge Discovery and Data Mining*. ACM, Paris, France, 219–228. https://doi.org/10.1145/1557019.1557049

[9] Thaís Damásio, Vinícius Pacheco, Fabrício Goes, Fernando Pereira, and Rodrigo Rocha. 2021. Inlining for code size reduction. In *25th Brazilian Symposium on Programming Languages*. ACM, Joinville, Brazil, 17–24. https://doi.org/10.1145/3475061.3475081

[10] Google Developers. 2022. *App Startup Time*. https://developer.android.com/topic/performance/vitals/launch-time

[11] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing Graphs and Indexes with Recursive Graph Bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16)*. ACM, New York, NY, USA, 1535–1544. https://doi.org/10.1145/2939672.2939862

[12] Malcolm C Easton and Ronald Fagin. 1978. Cold-start vs. warm-start miss ratios. *Commun. ACM* 21, 10 (1978), 866–872. https://doi.org/10.1145/359619.359634

[13] Paolo Ferragina and Giovanni Manzini. 2010. On compressing the textual web. In *International Conference on Web Search and Data Mining*. ACM, New York, NY, USA, 391–400. https://doi.org/10.1145/1718487.1718536

[14] Michael R Garey, David S Johnson, and Larry Stockmeyer. 1974. Some simplified NP-complete problems. In *Proceedings of the sixth annual ACM Symposium on Theory of Computing*. ACM, New York, NY, USA, 47–63. https://doi.org/10.1145/800119.803884

[15] Nikolas Gloy and Michael D Smith. 1999. Procedure placement using temporal-ordering information. *Transactions on Programming Languages and Systems* 21, 5 (1999), 977–1027. https://doi.org/10.1145/330249.330254

[16] Amir H Hashemi, David R Kaeli, and Brad Calder. 1997. Efficient procedure mapping using cache line coloring. *SIGPLAN Notices* 32, 5 (1997), 171–182. https://doi.org/10.1145/258915.258931

[17] Wenlei He, Julián Mestre, Sergey Pupyrev, Lei Wang, and Hongtao Yu. 2022. Profile inference revisited. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–24. https://doi.org/10.1145/3498714

[18] Apple Inc. 2022. *Reducing Your App's Launch Time*. https://developer.apple.com/documentation/xcode/reducing-your-app-s-launch-time

[19] Facebook Inc. 2015. *Optimizing Facebook for iOS Start Time*. https://engineering.fb.com/2015/11/20/ios/optimizing-facebook-for-ios-start-time

[20] Facebook Inc. 2021. *Redex: A bytecode optimizer for Android apps*. https://fbredex.com

[21] Facebook Inc. 2021. *Superpack: Pushing the limits of compression in Facebook's mobile apps*. https://engineering.fb.com/2021/09/13/core-data/superpack/

[22] J Kalamationos and David R Kaeli. 1998. Temporal-based procedure reordering for improved instruction cache performance. In *High-Performance Computer Architecture*. IEEE Computer Society, Las Vegas, Nevada, USA, 244–253. https://doi.org/10.1109/HPCA.1998.650563

[23] Brian W Kernighan and Shen Lin. 1970. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal* 49, 2 (1970), 291–307.

[24] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*. IEEE Computer Society, San Jose, CA, USA, 75. https://doi.org/10.1109/CGO.2004.1281665

[25] Rahman Lavaee, John Criswell, and Chen Ding. 2019. Codestitcher: inter-procedural basic block layout optimization. In *Proceedings of the 28th International Conference on Compiler Construction*, José Nelson Amaral and Milind Kulkarni (Eds.). ACM, Washington, DC, USA, 65–75. https://doi.org/10.1145/3302516.3307358

[26] Rahman Lavaee and Chen Ding. 2014. *ABC Optimizer: Affinity Based Code Layout Optimization*. Technical Report. University of Rochester.

[27] Kyungwoo Lee, Ellis Hoag, and Nikolai Tillmann. 2022. Efficient profile-guided size optimization for native mobile applications. In *International Conference on Compiler Construction*. ACM, Seoul, South Korea, 243–253. https://doi.org/10.1145/3497776.3517764

[28] Kyungwoo Lee, Manman Ren, and Shane Nay. 2022. Scalable size inliner for mobile applications (WIP). In *International Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, San Diego, CA, USA, 116–120. https://doi.org/10.1145/3519941.3535074

[29] Xing Lin, Guanlin Lu, Fred Douglis, Philip Shilane, and Grant Wallace. 2014. Migratory compression: Coarse-grained data reordering to improve compressibility. In *USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Santa Clara, CA, USA, 257–271.

[30] Gai Liu, Umar Farooq, Chengyan Zhao, Xia Liu, and Nian Sun. 2023. Linker Code Size Optimization for Native Mobile Applications. In *International Conference on Compiler Construction*. ACM, New York, NY, USA, 168–179. https://doi.org/10.1145/3578360.3580256

[31] Joel Mackenzie, Matthias Petri, and Alistair Moffat. 2022. Tradeoff Options for Bipartite Graph Partitioning. *IEEE Transactions on Knowledge and Data Engineering* (2022), 1–15. https://doi.org/10.1109/TKDE.2022.3208902

[32] Nezar Mansour. 2020. *Understanding Cold, Hot, and Warm App Launch Time*. https://blog.instabug.com/understanding-cold-hot-and-warm-app-launch-time/

[33] Julián Mestre, Sergey Pupyrev, and Seeun William Umboh. 2021. On the Extended TSP Problem. In *32nd International Symposium on Algorithms and Computation (LIPIcs, Vol. 212)*, Hee-Kap Ahn and Kunihiko Sadakane (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Fukuoka, Japan, 42:1–42:14. https://doi.org/10.4230/LIPIcs.ISAAC.2021.42

[34] Andy Newell and Sergey Pupyrev. 2020. Improved Basic Block Reordering. *IEEE Transactions in Computers* 69, 12 (2020), 1784–1794. https://doi.org/10.1109/TC.2020.2982888

[35] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing Function Placement for Large-scale Data-center Applications. In *International Symposium on Code Generation and Optimization*. IEEE Press, Austin, USA, 233–244. https://doi.org/10.1109/CGO.2017.7863743

[36] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: a practical binary optimizer for data centers and beyond. In *International Symposium on Code Generation and Optimization*. IEEE, Washington, DC, USA, 2–14. https://doi.org/10.1109/CGO.2019.8661201

[37] Karl Pettis and Robert C Hansen. 1990. Profile guided code positioning. *SIGPLAN Notices* 25, 6 (1990), 16–27. https://doi.org/10.1145/989393.989433

[38] Sofya Raskhodnikova, Dana Ron, Ronitt Rubinfeld, and Adam Smith. 2013. Sublinear algorithms for approximating string compressibility. *Algorithmica* 65, 3 (2013), 685–709. https://doi.org/10.1007/s00453-012-9618-6

[39] Peter Reinhardt. 2016. *Effect of Mobile App Size on Downloads*. https://segment.com/blog/mobile-app-size-effect-on-downloads/

[40] Rodrigo CO Rocha, Pavlos Petoumenos, Björn Franke, Pramod Bhatotia, and Michael O'Boyle. 2022. Loop rolling for code size reduction. In *International Symposium on Code Generation and Optimization*, Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman (Eds.). IEEE, Seoul, Republic of Korea, 217–229. https://doi.org/10.1109/CGO53902.2022.9741256

[41] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. 2021. HyFM: Function merging for free. In *International Conference on Languages, Compilers, and Tools for Embedded Systems*, Jörg Henkel and Xu Liu (Eds.). ACM, Virtual Event, Canada, 110–121. https://doi.org/10.1145/3461648.3463852

[42] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Effective function merging in the SSA form. In *International Conference on Programming Language Design and Implementation*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, London, UK, 854–868. https://doi.org/10.1145/3385412.3386030

[43] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. 2001. PLTO: A link-time optimizer for the Intel IA-32 architecture. In *Workshop on Binary Rewriting*. 1–7.

[44] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. 2023. Propeller: A Profile Guided, Relinking Optimizer for Warehouse-Scale Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Vancouver, BC, Canada, 617–631. https://doi.org/10.1145/3575693.3575727

[45] Jeffrey Scott Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (1985), 37–57. https://doi.org/10.1145/3147.3165

[46] Qi Wang and Torsten Suel. 2019. Document reordering for faster intersection. *Proceedings of the VLDB Endowment* 12, 5 (2019), 475–487. https://doi.org/10.14778/3303753.3303755

[47] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. 2012. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, Ambleside, United Kingdom, 113–126. https://doi.org/10.1145/2307636.2307648

[48] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343. https://doi.org/10.1109/TIT.1977.1055714

Received 2023-03-16; accepted 2023-04-21