Codestitcher: Inter-procedural Basic Block Layout Optimization*

Rahman Lavaee[†]
University of Rochester
Rochester, NY, USA
r.lavaee@google.com

John Criswell
University of Rochester
Rochester, NY, USA
criswell@cs.rochester.edu

Chen Ding
University of Rochester
Rochester, NY, USA
cding@cs.rochester.edu

ABSTRACT

Modern software executes a large amount of code. Previous techniques of code layout optimization were developed one or two decades ago and have become inadequate to cope with the scale and complexity of new types of applications such as compilers, browsers, interpreters, language VMs and shared libraries.

This paper presents Codestitcher, an inter-procedural basic block code layout optimizer which reorders basic blocks in an executable to benefit from better cache and TLB performance. Codestitcher provides a hierarchical framework which can be used to improve locality in various layers of the memory hierarchy. Our evaluation shows that Codestitcher improves the performance of the original program (already optimized with O3 and link time optimizations) by 3% to 25% (on average, by 10%) on 5 widely used applications with large code sizes: MySQL, Clang, Firefox, PHP server, and Python. It gives an additional improvement of 4% over LLVM's PGO and 3% over PGO combined with the best function reordering technique. For profiling, Codestitcher does not need instrumentation. Instead it uses branch history samples which are collected during the execution of the original program. Codestitcher's profiling and trace processing together incur an average overhead of 22.5%, compared to an average overhead of 90% from LLVM's PGO.

CCS CONCEPTS

• **Software and its engineering** → *Incremental compilers*;

KEYWORDS

Optimization, Compilers, Code layout, Locality, Profiling, Inter-procedural layout

ACM Reference Format:

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 (CC '19)*, February 16–17,

*This work was partially funded by the National Science Foundation (Contract No. CCF-1717877, CCF-1629376, CNS-1319617, and CCF-1116104), IBM CAS Faculty Fellowships, and a grant from Huawei Technologies. Any conclusion or recommendations expressed in this paper are those of the authors and do not necessarily reflect those of the funding organizations.

[†]Now at Google

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CC '19, February 16–17, 2019, Washington, DC, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6277-1/19/02...\$15.00

<https://doi.org/10.1145/3302516.3307358>

2019, Washington, DC, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3302516.3307358>

1 INTRODUCTION

For large applications, instruction misses are the main culprit for stalled cycles at the processor front-end. They happen not just in the instruction cache, but also in the unified cache at lower levels and in the TLB. Increasing the cache and TLB capacities or their associativities results in lower miss rates, but also increases their access latencies. In particular, the on-chip L1 instruction cache and TLB are required to have very low hit latencies (1–4 cycles). As a result, they have experienced only slight increase in capacity and associativity. For example, as reported by Intel [10], from the Pentium II to the Nehalem micro-architecture, the L1 instruction cache doubled in capacity (16KB to 32KB), but it retained its associativity of 4. From Nehalem to Haswell, it has retained its capacity, but doubled in associativity (4 to 8). Since Haswell, and until the recent micro-architecture, Coffee-Lake), the L1 instruction cache has not seen any improvement in capacity, associativity, or the block size.

In the meantime, modern software is growing in code size and complexity. Table 1 shows the increase in code size in terms of lines of code (LOC) and the program's binary size, for two applications: MySQL and Firefox. Assuming a fixed yearly rate in code growth, MySQL has grown by 10% in lines of code and by 16% in binary code size (text size) each year. For Firefox, this yearly growth has been 9% in lines of code and 14% in text size.

For code, longer cache lines seem more profitable as code is immune to false sharing and is more likely to be sequential. However, using a longer cache line only for code is not practical because of the inclusive, unified caches at lower levels. In modern architectures, hardware prefetching and stream buffers exploit the spatial locality beyond a single cache line.

Considering the slow rate of improvement in capacities of on-chip caches, it is natural to wonder how we can utilize them in an optimal way. An important factor is code layout. It affects the instruction cache performance in several ways. First, cache lines that are shared between unrelated code segments (functions or basic blocks which do not execute together) may result in lower

Table 1: Code size growth for MySQL and Firefox, in terms of LOC (lines of C/C++ code), and text size.

Application	Version	Release Year	LOC	.text Size
MySQL	6.2.16	2008	0.9M	3.6MB
	7.5.8	2017	2.1M	13.6MB
Firefox	1.0.1	2004	2.4M	7.2MB
	52.0	2017	7M	41.7MB

utilization of the cache. Second, when these code segments are stored consecutively in memory, prefetching may fill the cache with useless code. Third, in a set-associative cache, code blocks that are mapped to the same set may result in conflict misses. Finally, unrelated code segments which reside in the same page may inflict pressure on instruction TLB.

Profile-guided code layout optimization has been carefully studied in the past. An influential solution was proposed by Pettis and Hansen [22] (called PH in short). PH consists of function splitting, intra-function basic block chaining, and global function ordering. For Function splitting, PH separates the hot code from the cold code by splitting every function into these two parts. Next it performs basic block chaining to coalesce the hot part of every function. These function parts are then passed to the function reordering stage where PH finds an optimal ordering based on a greedy heuristic.

For over two decades, code layout optimization techniques have mainly followed the three-step framework suggested by PH. The research has primarily been focused on suggesting new heuristics for the function reordering stage. The main two shortcomings of prior techniques are as follows:

- (1) The coarse-grained hot-cold function splitting limits the scope of the layout optimization.
- (2) Function reordering heuristics are not based on precise spatial distance between related code segments.

To overcome these shortcomings, we introduce a new framework for inter-procedural basic block reordering named **Codestitcher**¹. Unlike prior techniques, Codestitcher splits functions into as many parts as necessary to expose all opportunities for placing related code segments together in memory. More importantly, Codestitcher uses the layout distance between related instructions as a parameter to incrementally construct the overall layout. Using this parameter, code collocation proceeds hierarchically, maximizing its benefits within successive layers of the memory hierarchy.

The main contributions of this paper are as follows:

- We identify the locality improvement opportunities that an inter-procedural basic block layout can offer, compared to a per-function layout.
- We present a new solution for basic block chaining, based on maximum cycle cover. Unlike prior techniques, our solution offers a theoretical guarantee.
- We define the distance-sensitive code layout problem and present a hierarchical framework which enables us to optimize code locality in all layers of the memory hierarchy, by successively solving the distance-sensitive layout problem for increasing distance parameters.
- We show how branch prediction is affected by inter-procedural basic block layout and why branch directions should not hamper locality optimizations.
- Finally, we present the evaluation of Codestitcher on five widely-used programs with large code sizes on a modern hardware platform. We also analyze, separately, the effect of using large pages and code layout granularity (function vs. basic block).

¹Codestitcher is available at <https://github.com/rlavaee/codestitcher>

Figure 1: High-level overview of Codestitcher

The rest of the paper is organized as follows. Section 2 describes the design of Codestitcher and motivates its design. Section 3 discusses our implementation of Codestitcher in LLVM and our profiling framework which is based on the Linux perf utility. Section 4 presents our evaluation on a set of five widely-used programs. Section 5 discusses related work, and Section 6 concludes.

2 DESIGN

2.1 Overview

Figure 1 gives a high-level overview of Codestitcher. The source code is first compiled to obtain LLVM bytecode files. These bytecode files are first linked and compiled to build the baseline program. As we build the program, we emit descriptive symbols at the beginning of each basic block. Then Codestitcher uses the Linux perf utility to sample profiles from the execution of this *symbolized* version of the program. After collecting the profiles, Codestitcher uses the emitted symbols to map these profiles to the control flow graph. It then performs *Basic Block Chaining* based on the profile data, to minimize the number of unconditional jumps that the program executes. The result of this step is a sequence of inter-procedural basic block (BB) chains. A BB chain is a sequence of basic blocks which terminates with an unconditional branch or return, but such instructions cannot happen in the middle of a BB chain. In this paper, functions calls are not considered to be unconditional jumps.

The constructed basic block chains are then passed to the *Hierarchical Code Collocation* stage, where Codestitcher iteratively joins them together to form longer sequences of basic blocks (basic block layouts). Conceptually, in this stage, Codestitcher iterates through a number of code distance parameters. For each distance parameter, Codestitcher collocates code segments in order to maximize spatial locality within that distance. The emitted basic block symbols are used again in this stage to find the size of each basic block.

2.2 Motivations

2.2.1 Inter-Procedural Basic Block Layout. We use a contrived example to show intuitively why an inter-procedural basic block layout can deliver higher improvements compared to a per-function layout. Consider the inter-procedural control flow graph in Figure 2 where function M calls function A 100 times during the program execution. The entry basic block of A (A0) ends with a conditional branch. The branch jumps to A1 80 times and jumps to A2 20 times. A1 calls function B but A2 calls function C.

A per-function layout for this problem requires merging all basic blocks of A before function reordering. PH, for example, performs BB chaining to form the layout $\boxed{A0-A1-A2}$ for A. Next, PH processes the call frequency edges in decreasing weight order. When processing each edge, PH joins together the layouts connected by that edge. A possible final layout is shown below.

LAYOUT 1. $\boxed{M} \quad \boxed{A0-A1-A2} \quad \boxed{B} \quad \boxed{C}$

The quality of a code layout can be evaluated with locality measures. A simple measure for code locality within one function is sequential code execution. Similarly, as a simple inter-procedural metric, we can look at how often control transfers between adjacent basic blocks.

In our example program, a total of 300 control transfers happen between 6 basic blocks and via 5 control flow edges. In layout 1, only two edges ($M \rightarrow A0$ and $A0 \rightarrow A1$) run between neighboring blocks (a total of 180 control transfers). Evidently, forming a single code segment for A has led both calls in A to separate from their callees (B and C).

On the other hand, an inter-procedural basic block layout can benefit from a finer grain splitting for A. In particular, if A is split into hot code paths ($\boxed{A0 \rightarrow A1}$ and $\boxed{A2}$), all function entries can be attached to their caller blocks, as shown in the layout below.

LAYOUT 2. $\boxed{M} \quad \boxed{A0 \rightarrow A1} \quad \boxed{B} \quad \boxed{A2} \quad \boxed{C}$

With Layout 2, all edges except $A0 \rightarrow A2$ run between adjacent blocks. This gives a total of 280 control transfers between adjacent blocks; an increase of about 55% over PH.

2.2.2 Layout Distance Between Related Instructions. Common procedure reordering heuristics follow a bottom-up approach. Starting with an initial set of code segments (hot function parts in PH), they iteratively join code segments together to form longer code segments. At each step, the heuristic makes two choices: which two code segments to join and in which direction. For instance, PH joins the code segments which are most heavily related by call frequency edges. In our example program in Figure 2, PH first joins M with A to form the layout $\boxed{M} \quad \boxed{A0-A1-A2}$. This layout then joins with B to form $\boxed{M} \quad \boxed{A0-A1-A2} \quad \boxed{B}$. Finally, connecting this layout with C gives the optimal PH layout. The three steps are shown in Figure 3.

At the last step, when $\boxed{M} \quad \boxed{A0-A1-A2} \quad \boxed{B}$ and $\boxed{C}$ join together, PH faces a choice between different merging directions (as shown in Figure 3(a)). The PH heuristic focuses on the merging points and chooses the direction that results in the largest number of

Figure 2: Weighted inter-procedural control flow graph of a program, over a hypothetical execution. Every circle represents a function. Control flow edges are represented by dashed lines and call edges are represented by solid lines.

Figure 3: PH function reordering [22], applied on the program in Figure 2

calls between the merging points. In this example, no calls happen between the merging points (C vs. M and B). Therefore, PH treats both directions equally beneficial. However, we notice that the lower orientation (also shown in Layout 1) forms a closer distance between A2 and C, which means a larger improvement in code locality.

Strikingly, the layout distance between related instructions (instructions connected by control flow edges) can guide us in making the best choice at every iteration. With each (ordered) pair of code segments, and every code distance level, we can attribute a value which indicates the number of control transfers which would happen within that distance if those two code segments join with each other. Then rather than solely focusing on calling frequencies, we can maximize the total number of control transfers which happen within close distances.

2.2.3 Basic Block Chaining via Max. Path Cover. Reordering the program's basic blocks may require inserting jump instructions to conserve the control flow. Clearly, this is not the case for a function layout. This means function reordering can easily be implemented in a linker. For instance, the Darwin linker supports function ordering via the command line flag `-order_file`. However, as we argued in Section 2.2.1, inter-procedural basic block layouts disclose new opportunities for improving code locality. The challenge is how to split functions without adding extra jumps.

The execution cost of unconditional jumps in modern processors is minimal thanks to the deep execution pipeline. However, additional jumps increase the instruction working set, which in turn increases the pressure on instruction cache and other layers of the memory hierarchy. As a result, minimizing the number of executed unconditional jumps is the first step towards finding the optimal layout.

Minimizing the number of executed unconditional jump instructions is equivalent to maximizing the number of fall-throughs, i.e., the number of times execution falls through to the next basic block. We formalize the fall-through maximization problem as follows.

DEFINITION 2.1 (THE FALL-THROUGH MAXIMIZATION PROBLEM). Given the control flow graph G for the whole program, along with frequencies of each control flow edge in G , find a decomposition of G

into a set of disjoint chains L which maximizes the total frequencies of edges in L .

This problem is exactly the same as the problem of *maximum weighted path cover* in control flow graphs. The general maximum weighted path cover problem is **NP**-hard as the traveling salesman problem reduces to it [15]. The simple greedy approach used by PH is the most well-known heuristic for solving this problem. This solution does not give any theoretical guarantee, but has a quick running time of $O(m \lg n)$ on a graph with m edges and n vertices. On the other hand, a direct 1/2-factor approximation algorithm exists for this problem. It is described as follows.

Given the weighted directed graph $G(V, E)$, we first remove all self-loops from G . Then we add zero weight edges for all non-existing edges in the graph (including self-edges) and find a *maximum cycle cover* in the resulting graph. A maximum cycle cover is a set of disjoint cycles which covers all vertices and has the maximum weight among all cycle covers. The maximum cycle cover can be reduced to the problem of maximum matching in weighted bipartite graphs. Thus, for a function with n hot basic blocks and m hot edges, we can use the Hungarian algorithm [11] to find the maximum cycle cover in time $O(n^2 m)$.

After finding the maximum cycle cover, we convert it to a path cover by dropping the lightest edge from each cycle. It is easy to verify that the weight of the resulting path cover is at least within a factor 1/2 of the optimal path cover [15].

We further improve the approximate path cover solution using the same idea as the greedy approach, i.e., after finding the approximate path cover, we consider the control flow edges in decreasing weight order, and add them to the path cover if they connect the tail of one path to the head of another.

The approximate solution has a theoretical bound but is not always guaranteed to deliver a better path cover than the greedy solution. In our experiments, we find that although the total number of fall-throughs for the whole program is higher for the approximate solution, the greedy solution outperforms the approximate for some functions. Therefore, we combine the two solutions to generate a hybrid one that beats each individual solution. For example, for one our tests in Section 4.4 (Clang), our approximation algorithm works better on 98% of all profiled functions and yields 0.19% higher total fall-throughs compared to the greedy approach. The hybrid approximation-greedy approach boosts this improvement by a marginal 0.03%, which is another indication for the overall superiority of the approximation approach.

2.3 Hierarchical Merging of Code Segments

Spatial locality is a multi-dimensional concept [8], defined as the *frequent* and *contemporaneous* use of data elements stored in *close storage locations*. Two dimensions are time and space, that is, how close in time the data elements are accessed, and how close in memory the elements are stored. A third dimension is frequency: how frequent those accesses are. The challenge in data/code layout optimization is to respect the importance of all three dimensions at the same time.

For code layout, unraveling the time and frequency dimensions is more tractable, as the program execution precisely overlaps with instruction accesses. Instructions within a basic block respect the

spatial locality. Therefore, code layout optimizations only focus on control transfers across basic blocks (branches, calls, and returns).

The real challenge is the space dimension because it is affected by code layout. The challenge becomes more significant in a finer grain code layout. For example, the distance between branch instructions and their targets is fixed in a function layout, but can vary in a basic block layout. The goal of code layout optimization is to ensure that frequently executed control instructions travel a smaller distance in the program binary.

Let us formally define the concept of layout distance. In our discussions below i and j always refer to basic blocks of the program.

DEFINITION 2.2 (LAYOUT DISTANCE). We denote by $\text{dist}(i, j)$ the number of bytes from i to j in the code layout, including i and j themselves.

For a distance parameter d , and a control transfer $i \rightarrow j$, we call $i \rightarrow j$ a *d -close transfer* if $\text{dist}(i, j) \leq d$. We also denote by $f(i, j)$ the number of times control transfers from i to j . If control never transfers from i to j , $f(i, j) = 0$.

We now define the *d -close Code Packing* problem.

DEFINITION 2.3 (d -CLOSE CODE PACKING). Order the program's basic blocks in a way that maximizes the total number of *d -close transfers*, i.e.,

$$\sum_{\substack{i, j \\ \text{dist}(i, j) \leq d}} f(i, j).$$

Solving this problem for a specific distance d would result in optimal spatial locality, but only in a limited distance. Given the hierarchical design of memory systems, it is important to improve spatial locality within different memory layers.

Codestitcher can follow this hierarchical design by successively solving the *d -close code packing* problem for increasing distance levels. At each distance level, Codestitcher gets as input an initial **partial layout**. A partial layout is a set of BB sequences, where each basic block appears in exactly one sequence. Codestitcher then solves the code packing problem for this distance level and passes the new partial layout to the next level.

The exact formulation of the problem Codestitcher solves at every distance level is a bit different from Definition 2.3. Let us introduce some terms that will help us in describing this problem and its solution.

Partial Layout Distance. In a partial layout L , for any two basic blocks i and j , their partial layout distance, $\text{dist}_L(i, j)$, is the same as in Definition 2.2, except that if i and j belong to different BB sequences, their distance is ∞ .

Super-Layouts and Sub-Layouts. Let L_1 and L_2 be two partial layouts. L_2 is a super-layout for L_1 if it can be derived from L_1 by joining some (or none) of the BB sequences in L_1 . In that case, L_1 is called a sub-layout of L_2 . A proper sub-layout of L is said to be of *finer granularity* than L .

d -close transfers in a partial layout. Let L be a partial layout. We define $t_d(L)$, the total number of *d -close transfers* in L , as

$$t_d(L) = \sum_{S \in L} \sum_{\substack{i, j \in S \\ \text{dist}_L(i, j) \leq d}} f(i, j).$$

In other words, $t_d(L)$ is the sum of d -close control transfers over all BB sequences in L . (The first summation in the equation goes over all the BB subsequences S which form the partial layout L and the second summation goes over all control transfers within every individual BB subsequence S).

We now define the problem of d -close partial layout, the building block of Codestitcher.

DEFINITION 2.4 (d -CLOSE PARTIAL LAYOUT). *Let L_0 be the initial partial layout. Find the finest grain super-layout of L_0 which has the maximum number of d -close control transfers ($t_d(L)$).*

The finest granularity constraint prevents joining BB sequences unless doing so results in additional d -close transfers. Effectively, it helps the next distance levels benefit from a larger number of close transfers.

2.4 Solving the d -Close Partial Layout Problem

The d -close partial layout problem asks for an optimal super-layout for L_0 , i.e., a super-layout with the maximum number of d -close transfers. Let L be a typical super-layout for L_0 . Each BB sequence $S \in L$ is the result of joining some BB sequences in L_0 . Let $I(S)$ be the set of those sequences.

The initial partial layout is fixed. Therefore, maximizing $t_d(L)$ is equivalent to maximizing its additive value with respect to L_0 , which is $\Delta t_d(L) = t_d(L) - t_d(L_0)$. This value can be expressed as follows.

$$\Delta t_d(L) = \sum_{S \in L} \sum_{\substack{Q, R \in I(S) \\ Q \neq R}} \sum_{\substack{i \in Q, j \in R \\ \text{dist}_L(i, j) \leq d}} f(i, j). \quad (1)$$

$\Delta t_d(L)$ is the objective value of the d -close partial layout problem. Its expression is intimidating, but easy to explain. The summation goes over all pairs of basic blocks i and j which didn't belong to the same BB sequence in the initial layout (L_0), but do in L .

We solve the d -close partial layout problem as follows. For each BB sequence S and every basic block $i \in S$, we define $F(i)$, the forward position of i as the number of bytes from the beginning of S to right after i . We also define $B(i)$, the backward position of i , as the number of bytes from the end of S to right before i .

We now define a directed weighted graph G on the set of BB sequences in L_0 . For each two BB sequences $S, T \in L_0$, we set the weight of the (S, T) edge equal to the number of d -close transfers between S and T when T is positioned right after S in the final layout. More formally, we have

$$w(S, T) = \sum_{\substack{i \in S, j \in T \\ B(i) + F(j) \leq d}} f(i, j) + f(j, i). \quad (2)$$

A super-layout for L is equivalent to a path cover for G . However, unlike the fall-through maximization problem, here, non-adjacent BB sequences may contribute to value of $t_d(L)$. (In equation 1, Q and R may be any two BB sequences in $I(S)$, not just adjacent ones.) Therefore, the weight of this path cover is only a lower bound on $\Delta t_d(L)$. Nevertheless, we can still use the graph formulation to compute a greedy solution. We describe the algorithm as follows.

At every step, we find the heaviest edge in G . Let it be (S, T) . We connect the instruction sequences corresponding to S and T , and

replace S and T with a new node representing the joined sequence $S.T$. Then we insert edges connecting this new node to the rest of the graph according to Formula 2. We continue this process until the additive value cannot be further improved. In other words, all edges in G become of zero weight.

Applying the greedy approach as explained above has a disadvantage. The heaviest edges in G are more likely to run between longer instruction sequences. Joining these long instruction sequences together may prevent higher gains in subsequent iterations and distance levels. To solve this problem, we set the weight of each edge (S, T) equal to $\frac{w(S, T)}{\text{size}(S) + \text{size}(T)}$, where $\text{size}(S)$ and $\text{size}(T)$ are respectively the binary sizes of S and T . The new edge weight formulation allows us to join shorter instruction sequences before longer ones.

We implement this algorithm as follows. First, we compute the inter-procedural control flow graph, that is, for each control flow instruction, a list of instructions it can jump to, along with the frequency of each control transfer. We use the control flow graph to build the directed weighted graph G as we explained above. Then we build a max heap of all edges $(S, T) \in G$ which have nonzero weights. This max heap helps us efficiently retrieve the edge with highest weight density at every iteration. At every iteration of the algorithm, we use the control flow graph along with the current location of instructions to update G , while we also update the max heap accordingly.²

2.5 Interaction with Branch Prediction

Traditionally, the performance of branch prediction has had a less significant impact on the design of code layout techniques, compared to code locality. Moreover, the use of more informative branch history tables has helped modern processors improve their branch prediction performance, to the extent that code layout can only impact the static branch prediction, i.e., when branch history information is not available.

We ran a microbenchmark to demystify the details of Intel's static branch prediction, which Intel does not disclose. Our microbenchmark consists of one test for every type of conditional branch (forward-taken, forward-not-taken, backward-taken, and backward-not-taken). For each type, we ran 1000 consecutive distinct branch instructions and counted the number of correctly predicted branches. Table 2 shows the result, averaged over 10 separate runs, on three CPU micro-architectures: Nehalem, Haswell, and Kaby-Lake.

The results suggest that while the old processor (Nehalem) used direction-based branch prediction, the newer processors (Haswell and Kaby-Lake) predict most branches as not-taken: a prediction strategy which favors spatial locality. In particular, backward branches usually belong to loops, and therefore, are more frequently taken. But Haswell often statically predicts them as not-taken. This strategy can potentially hurt branch prediction, but has the advantage that with good code locality, the wrong prediction path (not-taken) is fall-through and incurs a smaller penalty. Overall, for Haswell, code layout does not influence the branch prediction rate but better code locality can result in lower penalties.

²We use a tree-based implementation of the max heap for efficient update of the weights. Please see [5] for more information about max heaps.

Table 2: Static branch prediction rate for 1K branch instructions, each executed once

Branch Type	Kaby-Lake	Haswell	Nehalem
backward taken	76.3%	2.1%	99.9%
backward not-taken	96.1%	99.8%	0%
forward taken	0.6%	15.8%	0%
forward not-taken	95.3%	99.1%	99.3%

Interestingly, Kaby-Lake has significantly improved the prediction rate of backward branches (from 2.1% in Haswell to 76.3%) at the cost of a slight degradation in the prediction rate of the other three branch types (total of 22.7% reduction in prediction rate).

For direction-based branch prediction (as used in Nehalem), code layout can influence the branch prediction rate. For instance, frequently taken branches can enjoy a higher prediction rate if they face backward. Similarly, infrequently taken branches are better predicted if they face forward. PH imposes these orderings after BB chaining and before function reordering, by joining the basic block chains of every function in a specific direction. This can potentially result in joining code segments which are not strongly related. Furthermore, the larger code segments can limit the benefit in function reordering. On the other hand, as we argued above, static branch prediction is of lower priority than locality. Therefore, for Codestitcher, we decided to ignore this strategy and only focus on code locality.

3 IMPLEMENTATION

We implement Codestitcher using LLVM [13] version 3.9. It can optimize ELF binaries (supporting other formats would be straightforward) and supports multi-threaded code and multi-process applications. Shared libraries that are profiled and compiled with Codestitcher are optimized as well. Our implementation involves a compiler, a linker, and perf-based profiler. As a profile-guided optimization tool, Codestitcher has four stages: compiling the program to generate a symbolized version, running the program while sampling profiles with the Linux perf utility, constructing the new layout, and compiling the program to generate the optimized program. Next, we explain our profiling and layout construction framework in more detail.

3.1 Profiling With Linux Perf

During compilation, we emit unique symbols at the beginning of every basic block. For each basic block, its symbol string includes its function, its number, and its successors in the control flow graph. In short, these basic block symbols provide the necessary and sufficient information to build the control flow graph at the basic block level.

During the execution of this symbolized program, we leverage the Linux **perf** utility to gather profiles by sampling the *last branch record* (LBR) stack. The LBR stack records the most recent instructions which have caused a control transfer (conditional branches, jumps, calls, and returns). Thus it includes all control transfers between basic blocks except fall-throughs. Fall-throughs can be inferred from the LBR stack; for every two consecutive branches in the LBR stack, all instructions between the target address of the first branch and the source address of the second branch are executed

contiguously, as no other branches can execute in between. These contiguous instruction sequences include all fall-throughs between basic blocks.

3.2 Layout Construction And Code Reordering

We developed the Linux perf tool to build the weighted control flow graph based on the LBR trace. The weighted CFG and the static CFG information are passed on to the layout construction procedure. The layout construction procedure also gets as input a list of distance levels. At each distance level d , the layout construction algorithm solves the d -close partial layout problem for every shared library and executable, and passes the generated partial layout on to the next distance level. The emitted symbols are used to compute the size of basic blocks when they join to form longer instruction sequences.

At the end of this stage, Codestitcher will have generated a partial layout of all the hot code, consisting of instruction sequences which are not related enough to be joined together. Codestitcher generates a full layout by sorting these instruction sequences in decreasing order of their execution density. The execution density of an instruction sequence S is defined as $\frac{\sum_{i \in S} f(i)}{\text{size}(S)}$, where $f(i)$ is the number of times the instruction i has executed and $\text{size}(S)$ is the total size of S in bytes. This allows hotter instruction sequences to appear closer to the beginning of the text section in the executable.

4 EXPERIMENTAL EVALUATION

In this section, we evaluate Codestitcher by comparing it against LLVM's default profile-guided optimization (PGO) as well as function reordering techniques applied on top of PGO. We also compare against our implementation of Pettis-Hansen's global basic block reordering technique within the Codestitcher's framework. Finally, we discuss the overheads of Codestitcher and compare them with LLVM's PGO.

4.1 Experimental Setup

Our test machine runs Ubuntu 16.04 and is powered by two quad-core i7-7700 Kaby-Lake processors running at 3.60 GHz. Using the `getconf` command, we determined that each core has a 32 KB L1 instruction cache and a 256 KB L2 unified cache, each with associativity of 8. The shared last level cache is 8 MB and 16-way set associative. The instruction TLB includes 64 entries for regular pages, with associativity of 8, and 8 fully-associative entries for large pages. The shared L2 TLB contains 1536 with associativity of 6. We compile all our benchmark programs with Clang 3.9 with second optimization level (O3) and link time optimizations (LTO) enabled with the `-flto` flag.

4.2 Benchmarks

We test a set of widely-used programs that have large code sizes: a database server (MySQL cluster 7.4.12), a browser (Firefox 50.0), a compiler (Clang 3.9), a complete PHP server (httpd-2.4 along with PHP-7.0), and a python interpreter (Python 2.7.15). In this section, we first discuss the code size and cache characteristics of these programs and then describe the testing setup for each.

Table 3 shows the characteristics of these applications. The programs contain between 15 MB (MySQL) and 81 MB (Firefox) in the

Table 3: Code size and performance characteristics (MPKI means misses per thousand instructions)

Applications & Tests	I-cache MPKI	I-TLB MPKI	Binaries & Their .text Size (MB)
MySQL	62.44	9.35	mysqld (15)
Clang	8.14	1.01	clang (50), gold (4)
Firefox	9.16	1.54	libxul.so (81)
PHP Server	7.63	0.96	libphp7.so (14) httpd (1)
Python	3.40	0.19	python (2)

text section of their executables, except for Python whose text size is 2 MB (all code sizes are rounded up to 1 MB). Firefox and Python have multiple binaries, but the table shows the largest one for these two programs.

Table 3 also shows the I-cache and I-TLB miss rates for each application when they run on their test input. We measure the I-cache and I-TLB performance because their frequent misses lead to frequent stalls at the CPU front-end and limit the ability of out-of-order execution.³ The applications in Table 3 are ordered by their I-cache MPKI (misses per thousand instructions), from the highest, 62 MPKI for MySQL, to the lowest, 3.4 MPKI for Python. The cache performance does not always correlate with code size directly. Firefox, the largest program on our benchmark, trails MySQL in I-cache miss rate.

The cache and TLB performance depend on the size and usage of the executed code, which change in different applications and in different runs of the same application. Another important factor is the degree of multi-threading. MySQL and two PHP tests are multi-threaded, and more threads results in higher pressure on the I-cache and I-TLB.

Although all applications suffer from poor instruction performance, there is a large variation among them. The highest and lowest MPKIs differ by a factor of 18 times for the instruction cache and 49 times for TLB.

In a recent study [20], the authors measured the instruction cache performance of 4 server applications used internally at Facebook, including the execution engine that runs the Facebook web site [20]. These commercial applications are not available for our study. When we downloaded the open-source version of one of their applications, HHVM, we found that the program text after compilation is 24 MB, much lower than 133 MB, as reported by them. The baseline compilation of HHVM with Clang resulted in a broken binary. Therefore, we were unable to test this program.

Compared to our test suite, these commercial applications have larger code sizes, between 70 MB and 199 MB (compared to 2 MB and 81 MB among our applications). They show similar ranges of I-cache MPKIs, from 5.3 to 67 (compared to 3.4 to 62), and I-TLB MPKIs, from 0.4 to 3.1 (compared to 0.19 to 9.4), as the programs in our test suite do. Our MySQL test is an outlier with its 9.4 I-TLB MPKI, 3 times the highest number in the Facebook study.

³We use the ICACHE_64B:ITAG_MISS and ITLB_MISSES:MISS_CAUSES_A_WALK hardware performance counter events to measure I-cache and I-TLB misses.

Next we describe the testing setup for each program including the inputs that we use for profiling and testing. Notably, the profiling and testing inputs that we use for every program are completely disjoint, although they may come from the same benchmark suite.

Python. For testing the Python interpreter, we use the *apps* benchmark group from Google’s *Unladen Swallow* benchmark suite [24] as input scripts. The *apps* benchmark group consists of 6 “applicative” benchmarks: 2to3, chameleon_v2, html5lib, rietveld, spambayes, and tornado_http. The Unladen swallow benchmark provides a test harness which reports the average running time along with the standard deviation. We report the improvement over total wall-clock runtime of the 6 inputs. For profiling, we use the combination of *etree* and *template* scripts from the benchmark suite.

Firefox. For Firefox, we use the *tp5o* benchmark from Talos [18], a performance testing framework used at Mozilla. The *tp5o* benchmark tests the time it takes for Firefox to load 49 common web pages stored on the local file system. For each web page, Talos measures its load for 25 times, discards the first 5, and takes the median of the remaining 20. It then reports a single score as the geometric mean of these numbers over all web pages. For profiling, we use a combination of tests from Talos (*a11yr*, *tsvgx*, *tscrollx*, *tart*, *cart*, *kraken*, *ts_paint*, *tpaint*, *tresize*, and *sessionrestore*).

MySQL. For testing the MySQL server, we use the non-transactional read-only test (*oltp_simple*) from the Sysbench benchmark suite. We run both the Sysbench client and the MySQL server on the same machine. For both profiling and testing, we use 4 client threads to read from a database pre-populated by 8 tables each with 100,000 records. Sysbench reports the total throughput (requests per second) along with the average latency and tail latency (95 percentile) of requests. We report the mean improvement in average latency over 10 runs. For profiling, we use the combination of *select*, *select_random_points*, *update_index*, and *insert*.

PHP server. Our setup uses the Apache web server along with the PHP7 interpreter module; we therefore have Codestitcher optimize both binaries. We set up an isolated network between a single client and a single server using a 1 Gbps Ethernet cable. Our test data is WP-Test [17], an exhaustive set of WordPress [25] test data. For testing, we run Apachebench on the client side to make a total of 10,000 http requests to the server over 50 concurrent connections, to fetch the home page of our test Wordpress blog. Apachebench reports the throughput along with service time quantiles. We report the 90th percentile of service times. For profiling, we use a clean installation of Drupal [6] and follow the same setup as above.

Clang. Our Clang experiment optimizes both the Clang compiler and the Gold linker. We test Clang by compiling the programs in LLVM’s *multisource benchmarks* test suite, which consists of 158 programs. For each compilation, the test suite reports separate user times of compilation and linking. We measure the total user time. We use a single compilation job (*make -j 1*). We used the programs in the *Applications* sub-directory of the multisource benchmarks (which consists of 25 applications) for profiling.

Figure 4: Comparison of PGO, PGO.PH, and PGO.C3 against BB.PH and CS in terms of performance improvement over baseline

4.3 Comparison

We compare the performance improvements from Codestitcher against LLVM’s standard profile-guided optimization (PGO) framework, as well as function-reordering techniques applied on top of PGO-optimized programs.

LLVM’s PGO works in multiple stages. First, the compiler instruments the program for edge-profiling. When the instrumented program runs, it generates basic block execution count profiles. In the third stage, these profiles are fed into another compilation phase to guide intra-function basic block placement and inlining heuristics. To apply PGO, we first use the PGO-instrumented program to perform a single run of each profiling input and then use the emitted profiles to build the PGO-optimized programs.

Since LLVM’s PGO does not perform function reordering, we complement its affect by separately applying two function reordering techniques on the PGO-optimized programs; classical Pettis-Hansen function reordering (PH) [22] and call chain clustering (C3) as proposed by Ottoni and Maher [20]. C3 and PH have both been implemented as part of HHVM, an open-source PHP virtual machine, developed at Facebook. The two function reordering heuristics rely on the Linux **perf** utility to collect profile data. The **perf** tool interrupts the processes at the specified frequency and records the stack trace into a file.

To apply PH and C3 on top of PGO, we perform a second stage of profiling on PGO-optimized programs using HHVM’s **perf**-based profiling framework. We use the *instructions* hardware event to sample stack traces at the sampling rate of 6250 samples per second, similar to our own **perf**-based profiling framework. The obtained profiles are then used to compute the new PH and C3 function orderings and rebuild the programs with such orderings. Effectively, these optimized programs achieve intra-function basic-block reordering via PGO and function reordering via PH and C3.

We evaluate Codestitcher using our **perf**-based profiling framework and the layout construction procedure as we described in Sections 3.1 and 3.2. For a fair comparison with PGO.C3 and PGO.PH, we use the same **perf**-sampling parameters as we use for PH and C3 (i.e., using the *instructions* event and the sampling rate of 6250 samples per second). We apply Codestitcher on two layout distance parameter: 4 KB and 1 MB. The 4 KB parameter is the most important and will include most of the control transfers in the optimized code. The 1 MB distance parameter serves to reduce the interference at the L1 and shared TLBs besides reducing internal fragmentation for large pages.

Finally, we implement Pettis-Hansen’s full basic block reordering (consisting of function reordering, function splitting, and basic block reordering) within Codestitcher’s profiling and layout construction framework. For this technique, we directly use the profiles generated by Codestitcher.

In summary, we evaluate the following five techniques:

- **PGO**: LLVM’s standard PGO (consisting of function inlining and basic block reordering).
- **PGO.PH**: Pettis-Hansen’s function reordering adopted from HHVM [9], applied on top of PGO.
- **PGO.C3**: Function placement based on call chain clustering, adopted from HHVM and applied on top of PGO.
- **PH.BB**: Our implementation of Pettis-Hansen’s full basic block reordering technique, consisting of function splitting, basic block chaining, and function reordering, applied on top of baseline.
- **CS**: Codestitcher using our **perf**-based profiling work, applied on top of baseline.

We also evaluate each of the the four techniques (PGO.PH, PGO.C3, PH.BB, and CS) when using large pages for the hot code (resulting in the new techniques PGO.PH+LP, PGO.C3+LP, PH.BB+LP, and CS+LP). Similar to Ottoni and Maher [20], our implementation of large pages utilizes anonymous large pages for the hot code.

As we mentioned before, we use disjoint sets of profiling and testing inputs for each program. The same profiling inputs are used across all profiling stages for different techniques. We generate profiles over a single run for the full-trace PGO profiling, and over five identical runs for the other two **perf**-sampling-based profiling stages. For testing each technique, we report the average improvement over 10 identical runs of the test input, relative to the baseline program. We also report the standard deviation of the improvement.

4.4 Results

Figure 4 shows the performance improvement by 9 optimization techniques on all 5 programs.

For Python and PHP server, the hot code in each of their executing binaries will not exceed a single MB. Therefore, we do not evaluate the techniques with large pages on these two programs. The other three applications are optimized using each of the 9 optimization techniques.

The programs are shown from left to right in decreasing order of I-cache and I-TLB MPKI (Table 3). It is evident that the higher I-cache and I-TLB pressure (from left to right) a program exhibits,

Table 4: Geometric mean improvement across all 5 tests in Figure 4

	PGO	PGO.PH	PGO.C3	PH.BB	CS
regular pages	5.8	6.1%	6.7%	8.4%	9.9%
large pages	NA	6.0%	6.4 %	9.0%	10.1%

the more effective code layout optimization is, and the greater improvements we see in performance.

The speedups from CS range between 2.9% (for PHP server) and 24.8% (for MySQL), with three of the five tests gaining more than 10% speedup. We observe that CS significantly outperforms the other techniques in the largest three applications (MySQL, Clang, and Firefox). All techniques perform similarly well on PHP and Python.

For an overall comparison, Table 4 reports the geometric mean improvements of all 9 techniques on the 5 tests. As we mentioned above, Python and PHP server are only tested with regular pages because their hot code does not exceed a single MB. Therefore, when calculating the geometric mean of the improvements, we use the regular page results as the large page results for these two programs.

PH.BB is the second best technique after CS. For the three largest programs (MySQL, Clang, and Firefox) CS outperforms PH.BB both with regular pages and with large pages, and by a margin of between 0.7% and 4.8%. For Firefox, PH performs similarly well as the function reordering techniques. For smaller programs (PHP and Python) PH performs marginally better than the other techniques (outperforms CS by 0.1% and 0.2%, respectively).

Table 4 shows the geometric mean improvements across all tests, for each of the 9 optimization techniques. We observe that CS dominates other techniques both when using regular pages and large pages.

The relative I-cache miss rates are shown in Figure 5. The improvement in I-cache miss rate is mostly governed by the code layout granularity (basic block ordering vs. function ordering). Function reordering (PGO.PH and PGO.C3) adds little improvement to the I-cache miss rates of the PGO-optimized programs. Across all the 5 tests, the function reordering techniques yield a maximum of 4.1% I-cache miss rate reduction on top of PGO (which happens for MySQL). Surprisingly, for Python, both function reordering techniques increase the I-cache miss rates achieved by PGO alone. On the other hand, CS gives highest improvements across all the tests. It adds between 8.8% (Clang) and 27.0% (MySQL) improvement on top of PGO.

We also observe that regardless of the code layout technique being used, using large pages almost always increases the I-cache miss rate (especially for Firefox, using large pages proves detrimental across all techniques). The average increase in the miss rate is 0.6% for PGO.PH, 0.5% for PGO.C3, 2.0% for PH.BB, 1.7% for CS. Interestingly, Ottoni and Maher [20] discover the same phenomenon with PH and C3, although they report bigger increases.

Figure 6 shows the I-TLB miss rates relative to the baseline. First we observe that unlike I-cache miss rate, using large pages consistently reduces the I-TLB miss rate for each individual code layout technique. In particular, using large pages results in 13.5%

Table 5: Geometric mean of relative L1 instruction cache, instruction TLB, L2, and LL cache MPKI, and branch misprediction rate, across all tests

	PGO	PH.BB	PH.BB+LP	CS	CS+LP
rel. I-cache MPKI	66.0%	54.5%	55.5%	49.7%	50.6%
rel. I-TLB MPKI	87.6 %	89.6%	77.7%	79.8%	71.9%
rel. branch misprediction rate	97.2%	89.8%	89.4%	87.1%	86.3%
rel. L2-cache MPKI	87.6	81.0%	82.0%	77.9%	78.7%
rel. L3-cache MPKI	100.1%	96.8%	99.1%	94.9%	97.3%

additional improvement for PGO.PH, 13.6% for PGO.C3, 20.2% for PH.BB, and 11.6% for CS.

Second, we find that reordering code at basic block granularity does not have a consistent impact on the I-TLB miss rates compared to function granularity. While CS strongly outperforms the other techniques on MySQL both with regular and large pages, the function reordering techniques result in significantly lower I-TLB miss rates on Firefox (with large pages), Python, and PHP server.⁴

Finally, it is interesting to see that the I-TLB improvements from plain PGO are generally comparable to the other techniques with regular pages (we only compare PGO with regular page techniques since it does not do hot-cold splitting). Python is the only case where PGO performs significantly poorer than all the other techniques.

For an overall comparison of the performance counters, we report in Table 5, the geometric mean of relative I-cache and I-TLB miss rates across all 5 tests, along with level 2 (L2) and last level (L3) cache miss rates, and branch prediction. On average, CS gives the lowest miss rates on all three cache levels. CS+LP is the winner in I-TLB performance, beating CS by an absolute difference of 7.9%. Considering that CS+LP gives 0.2% higher overall improvement than CS (Table 4), we infer that its superior I-TLB performance neutralizes the effect of its slightly higher cache miss rates. CS+LP also dominates the other techniques with regards to branch prediction.

4.5 Overhead

Each of the code layout optimization techniques have their own costs and overheads. These overheads are categorized into three types, each corresponding to one stage of the profile-guided optimization framework:

- Profiling overhead: program slowdown during profiling
- Trace processing overhead: extra processing time to compute the total node/edge counts from profiles
- Layout construction and code reordering overhead: additional processing and build time to compute the optimal layout and reorder the program's binary

⁴We believe that the inferior performance of function reordering techniques is due to mispredicted branches. Specifically, in a basic block layout, where function fragments are stored far from each other, speculative long jumps across distant function fragments may result in higher I-TLB miss rates. Notably, this does not happen in a function layout where every function is stored contiguously in memory.

Figure 5: L1 instruction cache miss per 1K instructions, relative to the baseline.

Figure 6: L1 instruction TLB miss per 1K instructions, relative to the baseline.

As Ottoni and Maher [20] explain, today’s data center applications are profiled in production mode and therefore profiling overhead is the most important overhead since it impacts the underlying program’s performance. All other overheads (trace storage overhead, trace processing, layout construction, and excess build time) do not directly impact the application’s execution.

We report the profiling overhead of Codestitcher along with LLVM’s PGO. In addition, we report the trace processing overhead and the build overhead of Codestitcher.

Table 6 shows the overhead results. The profiling overheads are measured differently, and in accordance with our experimental setup: 90% tail latency increase in PHP, average latency in MySQL, increase of total compilation and link time for Clang, increase of the score reported by the *talos* benchmark suite for Firefox, and increase of total wall-clock time for Python.

The trace processing overheads for Codestitcher are reported as the processing time relative to the length of the profiling run. This processing reads all the sampled LBR stack traces and builds the weighted inter-procedural CFG. We have implemented this inside the Linux perf utility.

The layout construction and linking overheads are reported as the excess build time relative to the link time of the program (given all the LLVM bitcode files). For Codestitcher, our layout construction library is prototypical and is implemented outside the compiler and in Ruby.

The overhead results indicate that the combined profiling and trace processing overhead of Codestitcher (22% average overhead) is much less than LLVM’s PGO which incurs an average overhead of 90%. Moreover, we note that Codestitcher requires no instrumentation for profiling, except for emitting symbols in the program. Two major drawbacks of our technique are the higher overhead

Table 6: Overheads relative to baseline program, for Codestitcher and LLVM’s PGO.

		MySQL	Clang	Firefox	PHP	Python
CS	profile sampling	6.7%	5.3%	3.7%	7.2%	5.5%
	trace processing	21%	10%	24%	23%	6%
	layout construction & code reordering	329%	54%	161%	91%	456%
PGO	profiling	45%	149%	113%	142%	4%

of layout construction and its storage overhead for traces. Our prototypical layout construction library can be significantly optimized by re-implementation in the compiler or in the perf tool. The storage overhead can be reduced by performing trace processing in parallel with trace collection.

5 RELATED WORK

Code layout is a form of cache-conscious data placement, which has been shown to be NP-hard, even for an approximate solution [1, 3, 14, 21]. However, code layout is more tractable for two main reasons. First code access patterns can be precisely captured from program execution. Second, the less structured format of code allows code reordering to be done with higher flexibility.

Ramirez et al. [23] specifically studied code layout optimizations for transaction processing workloads. They implemented PH in the Spike binary optimizer [4], and studied the impact of different stages of PH. Their implementation mostly follows PH, but uses a fine grain function splitting, analogous to us.

More recently, Ottoni and Maher [20] introduced call-chain clustering, a new heuristic for function reordering. Call-chain clustering follows a similar bottom-up approach as PH, but aims at reducing the expected call distance, by merging function layouts in the direct of the call. They evaluate their technique on four data-center applications including HHVM. Our work gives an independent evaluation of their heuristic on programs which are more widespread, but have smaller code sizes.

Focusing on conflict misses, Gloy and Smith [7] developed a solution based on a *Temporal Relation Graph* (TRG). They defined the temporal relation between two code blocks as the number of times two successive occurrences of one block are interleaved by at least one occurrence of the other. The solution works best on direct-mapped caches. It looks at TRG edges running between code blocks mapped to the same cache lines. Considering these edges in sorted order, it tries to remove conflict misses by remapping procedures to other cache lines. Once all mappings are obtained, it orders the procedures to realize those mappings while minimizing the total gap between procedures.

The direct extension of TRG for k -way set-associative caches requires storing temporal relations between every code block and every k -sized set of code blocks. It is evident that as programs become larger and cache associativity goes higher, storing this information becomes more expensive. Liang and Mitra [16] introduced the *intermediate blocks profile*, a compact model for storing this information, which enables them to evaluate the approach on up to 8KB associative caches, and on programs with up to about 500 procedures. Besides this, their optimization is not portable across different cache configurations.

Finally, while in this paper we used edge-profiling to optimize code layout, path profiling [2] gives more precise information about the control flow in a program. Whole program path profiling [12] extend path profiling to analyze the program's entire control flow, including loop iterations and inter-procedural paths. We believe that unless we allow for code duplication, path profiling is too excessive of information for code layout. With code duplication, different hot paths which are executed in different contexts (and may possibly share code fragments), could be laid out in different parts of the code [19]. However, because of the expensive compulsory misses in lower level caches and TLBs, we expect that code duplication harms the performance more than it benefits.

6 SUMMARY

In this paper, we introduced a new technique for inter-procedural basic block layout including path-cover based BB chaining and distance-based BB collocation. Our technique achieves a performance improvement of 10% over five widely-used programs. The improvement is primarily due to the finer grain splitting of functions and the optimal code collocation, enabled by our distance-sensitive code collocation framework. Compared to LLVM's PGO combined with the best function reordering technique, our technique results in about 3% more improvement while it incurs less than a quarter of the profiling overhead of PGO alone.

REFERENCES

- [1] Hagit Attiya and Gili Yavneh. 2017. Remote Memory References at Block Granularity. In *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017*. 18:1–18:17.
- [2] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 29, Paris, France, December 2-4, 1996*. 46–57.
- [3] 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, Article 53 (Jan. 2019), 53:1–53:28 pages.
- [4] Robert S. Cohn, David W. Goodwin, and P. Geoffrey Lowney. 1998. Optimizing Alpha Executables on Windows NT with Spike. *Digital Technical Journal* 9, 4 (1998), 3–20.
- [5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, 3rd Edition*. MIT Press.
- [6] Drupal. 2000. Drupal. <https://www.drupal.org/>. Online; accessed 10 April 2018.
- [7] Nicholas C. Gloy and Michael D. Smith. 1999. Procedure placement using temporal-ordering information. *ACM Transactions on Programming Languages and Systems* 21, 5 (1999), 977–1027.
- [8] Saurabh Gupta, Ping Xiang, Yi Yang, and Huiyang Zhou. 2013. Locality principle revisited: A probability-based quantitative approach. *J. Parallel Distrib. Comput.* 73, 7 (2013), 1011–1027.
- [9] HHVM 2011. HHVM: A virtual machine designed for executing programs written in Hack and PHP. <https://github.com/facebook/hhvm>. Online; accessed 19 November 2016.
- [10] Intel Corporation. 2016. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation. Available at <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>.
- [11] Harold W. Kuhn. 2010. The Hungarian Method for the Assignment Problem. In *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*. 29–47.
- [12] James R. Larus. 1999. Whole Program Paths. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*. 259–269.
- [13] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization, CGO, San Jose, CA, USA, 75–88*.
- [14] Rahman Lavaee. 2016. The hardness of data packing. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. St. Petersburg, FL, USA, 232–242.
- [15] Moshe Lewenstein and Maxim Sviridenko. 2003. A 5/8 Approximation Algorithm for the Maximum Asymmetric TSP. *SIAM J. Discrete Math.* 17, 2 (2003), 237–248.
- [16] Yun Liang and Tulika Mitra. 2010. Improved procedure placement for set associative caches. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*. Scottsdale, AZ, USA, 147–156.
- [17] Michael Novotny. 2013. WP Test. <https://github.com/poststatus/wpctest>. Online; accessed 19 November 2017; acquired by Post Status Labs Project in 2016.
- [18] Mozilla Foundation. 2007. Talos. <https://wiki.mozilla.org/Buildbot/Talos>. Online; accessed 19 November 2017.
- [19] Frank Mueller and David B. Whalley. 1995. Avoiding Conditional Branches by Code Replication. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 56–66.
- [20] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing function placement for large-scale data-center applications. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. 233–244.
- [21] Erez Petrank and Dror Rawitz. 2002. The hardness of cache conscious data placement. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, OR, USA, 101–112.
- [22] Karl Pettis and Robert C. Hansen. 1990. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. White Plains, NY, USA, 16–27.
- [23] Alex Ramirez, Luiz André Barroso, Kourosh Gharachorloo, Robert S. Cohn, Josep-Lluís Larriba-Pey, P. Geoffrey Lowney, and Mateo Valero. 2001. Code layout optimizations for transaction processing workloads. In *Proceedings of the International Symposium on Computer Architecture*. Göteborg, Sweden, 155–164.
- [24] Unladen 2009. Unladen Swallow Benchmarks. <https://code.google.com/p/unladen-swallow/wiki/Benchmarks>. Online; accessed 13 September 2016.
- [25] WordPress Foundation. 2003. Wordpress. <https://wordpress.org/>. Online; accessed 10 April 2018.