Optimizing Function Placement for Large-Scale Data-Center Applications

Guilherme Ottoni   Bertrand Maher

Facebook, Inc., USA

{ottoni,bertrand}@fb.com

Abstract

Modern data-center applications often comprise a large amount of code, with substantial working sets, making them good candidates for code-layout optimizations. Although recent work has evaluated the impact of profile-guided intra-module optimizations and some cross-module optimizations, no recent study has evaluated the benefit of function placement for such large-scale applications. In this paper, we study the impact of function placement in the context of a simple tool we created that uses sample-based profiling data. By using sample-based profiling, this methodology follows the same principle behind AutoFDO, i.e. using profiling data collected from unmodified binaries running in production, which makes it applicable to large-scale binaries. Using this tool, we first evaluate the impact of the traditional Pettis-Hansen (PH) function-placement algorithm on a set of widely deployed data-center applications. Our experiments show that using the PH algorithm improves the performance of the studied applications by an average of 2.6%. In addition to that, this paper also evaluates the impact of two improvements on top of the PH technique. The first improvement is a new algorithm, called C^3, which addresses a fundamental weakness we identified in the PH algorithm. We not only qualitatively illustrate how C^3 overcomes this weakness in PH, but also present experimental results confirming that C^3 performs better than PH in practice, boosting the performance of our workloads by an average of 2.9% on top of PH. The second improvement we evaluate is the selective use of huge pages. Our evaluation shows that, although aggressively mapping the entire code section of a large binary onto huge pages can be detrimental to performance, judiciously using huge pages can further improve performance of our applications by 2.0% on average.

1. Introduction

Modern server workloads are large and complex programs that have been highly tuned over the course of their development. As a result, many such applications lack obvious "hot spots" that an engineer can optimize to deliver large overall performance improvements. Instead, the sheer volume of code that must be executed can be a bottleneck for the system. As a result, code locality is a relevant factor for performance of such systems.

While the large size and performance criticality of such applications make them good candidates for profile-guided code-layout optimizations, these characteristics also impose scalability challenges to optimize these applications. Instrumentation-based profilers significantly slow down the applications, often making it impractical to gather accurate profiles from a production system. To simplify deployment, it is beneficial to have a system that can profile unmodified binaries, running in production, and use these data for feedback-directed optimization. This is possible through the use of *sample-based profiling*, which enables high-quality profiles to be gathered with minimal operational complexity. This is the approach taken by tools such as AutoFDO [1], and which we also follow in this work.

The benefit of feedback-directed optimizations for some data-center applications has been evaluated in some previous work, including AutoFDO [1] and LIPO [2]. Chen et al. [1] evaluated the impact of intra-module feedback-directed optimizations, while Li et al. [2] evaluated the impact of some cross-module optimizations, in particular inlining and indirect-call promotion. However, no recent work has evaluated the benefit of function placement for large-scale data-center applications.

In this paper, we demonstrate the benefit of optimizing function placement for these large-scale server applications. By default, the linker places functions according to the order the object files are specified in the command line, with no particular order within each object file. This arbitrary layout disperses the hot code across the text section, which reduces the efficiency of caches and TLBs. The potential to improve the function order and thus the performance of a binary was demonstrated by Pettis and Hansen [3]. In this work, we first evaluate the performance impact of their technique on a set of widely deployed data-center applications, and then show the impact of two improvements on top of this traditional technique.

Our study is conducted in the context of `hfsort`, which is a simple tool we created to sort the functions in a binary. Our methodology was designed to be simple enough to be applied to large-scale production systems with little friction. Like AutoFDO [1], this is achieved by leveraging sample-based profiling.

Overall, this paper makes the following contributions:

- it evaluates the impact of Pettis and Hansen's traditional function-ordering algorithm on a set of widely deployed data-center applications;
- it identifies an opportunity for potential improvement over Pettis and Hansen's algorithm, and then describes a novel algorithm based on this insight;
- it describes a simple, user-level approach to leverage huge pages for a program's text section on Linux;
- it experimentally evaluates the aforementioned techniques, demonstrating measurable performance improvements on our set of data-center applications.

This paper is organized as follows. We start by describing the applications studied in this paper and some key performance characteristics in Section 2. Then Section 3 presents an overview of our methodology for improving code layout, followed by a description of techniques for building a dynamic call graph (Section 4) and for sorting the functions (Section 5). Section 6 then describes our technique for leveraging huge pages for the text section on Linux. A thorough evaluation of our techniques on four widely deployed server applications is presented in Section 7. Finally, related work is discussed in Section 8 and Section 9 concludes the paper.

2. Studied Applications

In order to demonstrate the importance of both code locality and the proposed techniques for server applications, this paper focuses on four systems that account for a large portion of the computing cycles spent to run some of the most popular websites in the world. The first of these systems is the HipHop Virtual Machine (HHVM) [4], which is the PHP and Hack execution engine powering many servers across the Internet, including three of the top ten websites in the world: Facebook, Wikipedia, and Baidu [5]. The second system evaluated in this paper is TAO [6], a highly distributed, in-memory, data-caching service used at Facebook. The other two systems are AdIndexer, an ads-related service, and Multifeed Aggregator, a service used to determine what is shown in the Facebook News Feed.

These four applications contain between 70 and 199 MB of program text. In the case of HHVM, which includes a JIT compiler, there is an even larger portion of dynamically generated code. Note, however, that the function-ordering techniques studied in this paper are applied statically and thus do not impact the dynamically generated code in HHVM. For running Facebook, the breakdown of HHVM's execution time is 70% in static code and 30% in dynamic code. For all these applications, while the majority of the time is spent in smaller, hot portions of the code, there is an enormous tail of lukewarm code that executes with moderate frequency. This long tail competes for I-TLB, I-cache, and LLC space with the hot code. The resulting cache pressure from this large code footprints leads to high miss rates at all levels.

Application	.text (MB)	IPC	I-TLB MPKI	I-Cache MPKI
AdIndexer	186	0.61	0.48	9.84
HHVM	133	0.53	1.28	29.68
Multifeed	199	0.87	0.40	5.30
TAO	70	0.30	3.08	67.42

Table 1. Code size and performance characteristics of Facebook server applications. (MPKI means misses per 1000 instructions.)

Frequent misses stall the processor front end and limit opportunities for out-of-order scheduling, leading to low instructions-per-cycle (IPC). Table 1 shows the binary size and cache performance of the studied applications. For example, without the optimizations studied in this paper, HHVM suffers 29.7 I-cache misses and 1.3 I-TLB misses per thousand instructions, and processes only 0.53 instructions per cycle. These high miss rates indicate that the processor's resources are significantly underutilized due to frequent front-end stalls.

3. System Overview

This section gives an overview of the methodology used in this paper to improve binary layout. One of the main design goals of this methodology was to be practical enough to be used in real, large-scale production systems. Figure 1 illustrates the steps and components of this methodology.

The first step in our methodology is to collect profile data. To do so, we use production servers running unmodified binaries. We select a small set of loaded servers for profiling, and we use a sampling-based tool (the Linux `perf` utility) to gather profile data. The `perf` tool uses hardware performance counters and interrupts the process at the specified intervals to collect profile data. As described in Section 4, we use the *instructions* `perf` event to collect profile data (either last-branch records or stack traces) at regular intervals measured in number of dynamic instructions executed.

The profile data is fed into a tool that generates an optimized list of hot functions. The tool we built to sort the hot functions is called `hfsort`, and it is available as open source [7]. This tool starts by processing the profile data to build a *dynamic call graph*, as described in Section 4. This profile-based call graph is the input to the layout algorithm, which uses the node and edge weights to determine an appropriate ordering for the functions. The layout algorithm proposed in this paper is described in detail in Section 5.2. To enable the actual function reordering by the linker, the program is compiled using `gcc -ffunction-sections`, which places each function in an appropriately named ELF section [8]. The `hfsort` tool then generates a customized linker script, which directs the linker to place sections (i.e. functions) in a specific order. Because the linker may employ identical code folding to reduce code size by aliasing functions with identical bodies, the generated linker script should list all aliased functions to ensure proper placement.

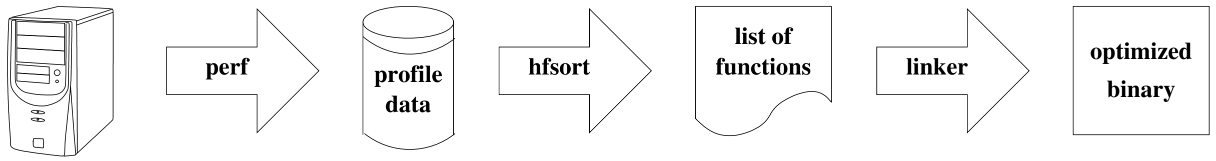


Figure 1. System overview.

This methodology is attractive from a deployment perspective for two main reasons. First, it does not require a special profiling build. Instead, profile data is collected with negligible overhead on unmodified binaries running in their regular production environments. Second, this methodology uses simple, off-the-shelf tools to perform the optimization. `perf` is a standard utility, and so are linkers such as `gold` [9] with support for a custom script. The function ordering can be performed by an external utility such as `hfsort`, and need not be explicitly supported by the linker or the compiler.

4. Building the Call Graph

Application binaries are normally constructed around the concept of functions (or procedures) from a higher-level programming language. The binary code for each function is normally generated contiguously in the binary, and transitions between functions are performed via function calls and returns. In this work, we focus on improving the locality of these transitions among the functions at runtime. To achieve this goal, it is natural to use a *call graph* representation of the binary. A call graph $G = (V, A)$ contains a set of nodes V, each associated with a corresponding function f in the binary, and also a set of arcs A, where each arc $f \to g$ represents the fact that function f calls function g. In order to represent the dynamic occurrence of transitions between functions, we use a *weighted call graph*. That is, associated with each arc $f \to g$, there is a weight $w(f \to g)$ representing the number of times that function f calls g at runtime.

Although a non-weighted call graph for a program can be built statically, obtaining a weighted call graph requires some sort of profiling. The straightforward profiling approach is to instrument the program with counters inserted at every call site, and then to run the program on representative profiling inputs. For binary layout optimizations, which are static transformations, this profiling data is then fed into a pass that rebuilds the binary using the data. This approach is commonly used for profile-guided optimizations, including in the seminal code-layout work by Pettis and Hansen [3].

Overall, there are two main drawbacks with this approach based on instrumentation, which complicate its use in production environments. First, it requires intrusive instrumentation of the program and an extra, special build of the application. Furthermore, to instrument the whole application, including libraries, this instrumentation should be done either at link-time or on the final binary. Second, instrumentation incurs significant performance and memory overheads that are often inadequate to be used in real production environments. As a result, a special, controlled environment is often needed to execute the profiling binary, which then limits the amount of profiling data that can be collected. Together, these issues result in many production environments completely opting out of profile-guided optimizations, despite their potential performance benefits.

The alternative to overcome the drawbacks of instrumentation is to rely on sampling techniques to build a weighted call graph. Compared to instrumentation, sampling-based techniques are intrinsically less accurate, although this inaccuracy can be limited by collecting enough samples. Furthermore, efficient sampling techniques enable the collection of profiling data in actual production environments, which has the potential to be more representative than instrumentation-based profiles collected in less realistic environments.

We have experimented with two sampling-based techniques that have negligible overheads and thus can be used to obtain profiling data on unmodified production systems. The first one is to use hardware support available on modern Intel x86 processors, called *last branch records* (LBR) [10]. This is basically a 16-entry buffer that keeps the last 16 executed control-transfer instructions, and which can be programmed to filter the events to only keep records for function calls. The caller-callee addresses from the LBR can be read through a script passed to the Linux `perf` tool. The second approach we have experimented with is based on sampling stack traces instead of flat profiles, which is also used in other work, e.g. the `pprof` CPU profiler. This can be done very efficiently, in particular for programs compiled with frame pointers. Stack traces can be obtained via `perf`'s `--call-graph` option. From a stack trace, a relatively accurate weighted call graph can be computed by just looking at the top two frames on the stack.[1] More precisely, the weight $w(f \to g)$ of arc $f \to g$ can be approximated by how many sampled stack traces had function g on the top with f immediately below it. Our experiments with these two sampling-based approaches revealed that they lead to weighted call graphs with similar accuracy.

Figure 2 illustrates a dynamic call graph that can be built with either of these sampling-based approaches. For example, the weight $w(B \to C) = 30$ means that, via hardware counters, there were 30 call entries for B calling C in the LBR, or, alternatively via stack traces, that 30 sampled stack traces had function C at the top with function B immediately below it.

[1] We have found that looking at lower frames on the stack can distort the weights.

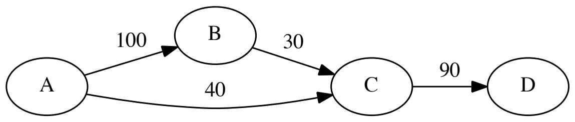


Figure 2. Example of a dynamic call graph.

5. Function-Ordering Heuristics

Petrank and Rawitz [11] demonstrated that finding an optimal data or code placement that minimizes cache misses is a NP-hard problem. Furthermore, they also showed that this problem is unlikely to have an efficient approximate solution either. Besides that, server applications also typically have a very large set of functions, in the order of hundreds of thousands or more, which render applying an optimal, exponential-time solution impractical. Therefore, in practice, heuristic solutions are applied to these problems.

This section describes two heuristics to obtaining a binary layout. Section 5.1 describes a prior heuristic by Pettis and Hansen [3], while Section 5.2 describes the novel heuristic proposed in this paper. Section 7 presents an experimental evaluation comparing the performance impact of these techniques on the applications described in Section 2.

5.1 Pettis-Hansen (PH) Heuristic

Pettis and Hansen [3] studied various aspects of code layout, including reordering functions through the linker to improve code locality (Section 3 in [3]). Their function-ordering algorithm is a commonly used technique in practice, having been implemented in compilers, binary optimizers, and performance tools [3, 12–15]. We describe their heuristic for this problem in this section, which we call the *PH heuristic*, and illustrate how it operates in a simple example.

The PH heuristic is based on a weighted dynamic call graph. However, the call graph used by PH is *undirected*, meaning that an arc between functions F and G represents that either function F calls function G, or function G calls F, or both. Although subtle, this indistinction between callers and callees in the call graph can lead to sub-optimal results as illustrated in Section 5.2.

Once the call graph is constructed, PH processes each edge in the graph in decreasing weight order. At each step, PH *merges* the two nodes connected by the edge in consideration. When two nodes are merged, their edges to the remaining nodes are coalesced and their weights are added up. During the algorithm, a linear list of the original nodes associated with each node in the graph is maintained. When merging two nodes a and b, careful attention is paid to the original connections in the graph involving the first and last nodes in the lists associated with a and b. Reversing of either a or b is evaluated as a mechanism for increasing the weight of the new adjacent nodes that will result from the merge, and the combination that maximizes this metric is chosen. The process repeats until there are no edges left in the graph.

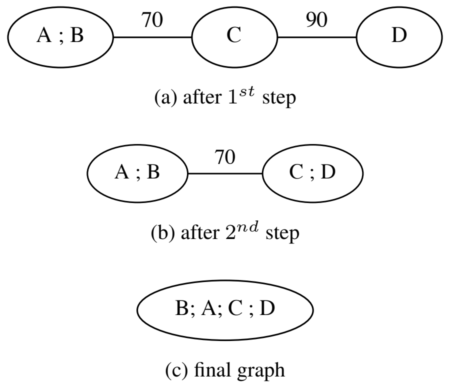


Figure 3. PH heuristic processing example from Figure 2.

We illustrate the operation of the PH heuristic in the example in Figure 2. In the first step, PH processes the heaviest-weight edge, $A \rightarrow B$, merging nodes A and B and obtaining the graph in Figure 3(a). In the second step, the heaviest edge in Figure 3(a), connecting C and D, is selected. In the final step, the only edge remaining is used to merge nodes $A; B$ and $C; D$. At this point, four different options are considered, corresponding to either reversing or not each of the nodes. The edges in the original graph (Figure 2) are analyzed, and the choice to make A and C adjacent is made because they are connected by the edge with the heaviest weight. To realize this decision, the nodes in the merged node $A; B$ are reversed before making the final merge. The final ordering is illustrated in Figure 3(c).

5.2 Call-Chain Clustering (C^3) Heuristic

In this section, we describe a new call-graph-based heuristic, which we named *Call-Chain Clustering* (C^3). We first present a key insight that distinguishes C^3 from the PH heuristic, and then describe C^3 and illustrate its operation with an example.

Unlike PH, C^3 uses a *directed* call graph, and the role of a function as either the caller or callee at each point is taken into account. One of the key insights of the C^3 heuristic is that taking into account the caller/callee relationships matters. We illustrate this insight on a simple example with two functions, F and G, where function F calls G. In order to improve code layout, compilers typically layout a function so that the function entry is at the lower address. During the execution of a function, instructions from higher address are fetched and executed. Representing the size of function F by $|F|$, the average distance in the address space of any instruction in F from the entry of F is $|F|/2$. So, assuming this average distance from the entry of F to the call to G within F and the layout where G follows F in the binary, the distance to be jumped in the address space when executing this `call G` instruction in F is $|F|/2$. This is illustrated in Figure 4(a). Now consider the layout where G is placed before F. In this case, the distance to be jumped by the `call G` instruction is $|G|+|F|/2$. This is illustrated in Figure 4(b). The

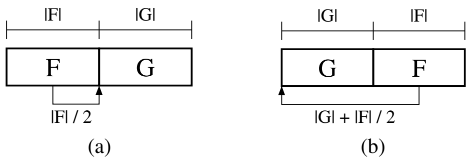


Figure 4. Two possible layouts for functions F and G, where F calls G.

distance in this second case can be arbitrarily larger than in the first case depending on the size of G. And, the larger the distance, the worse the locality: there is a higher probability of crossing a cache line or a page boundary.

The C^3 heuristic operates as follows. It processes each function in the call graph, in decreasing order of profile weights. Initially, each function is placed in a *cluster* by itself. Then, when processing each function, its cluster is *appended* to the cluster containing its most likely predecessor in the call graph. The intuition here is that we want to place a function as close as possible to its most common caller, and we do so following a priority from the hottest to the coldest functions in the program. By following this order, C^3 effectively prioritizes the hotter functions, allowing them to be placed next to their preferred predecessor. The only thing that blocks the merge of two clusters is when either of them is larger than the *merging threshold*. The merging threshold that C^3 uses is the page size because, beyond this limit, there is no benefit from further increasing the size of a cluster: it is already too big to fit in either an instruction cache line or a memory page.

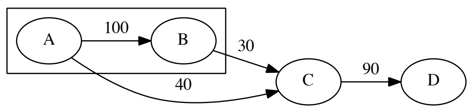
C^3's last step is to sort the final clusters. In this step, the clusters are sorted in decreasing order according to a *density metric*. This metric is the total amount of time spent executing all the functions in the cluster (computed from the profiling samples) divided by the total size in bytes of all the functions in the cluster (available in the binary):

$$density(c) = \frac{time(c)}{size(c)}$$

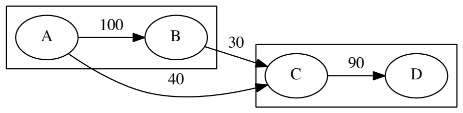
The intuition for using this metric is to try to pack most of the execution time in as few code pages as possible, in order to further improve locality. A large, hot function puts more pressure on the cache hierarchy than an equally hot but smaller function. Therefore, preferring the latter will minimize the number of cache lines or TLB pages required to cover most of the program's execution time.

Note that, although we limit the cluster sizes through the merging threshold, we still place consecutive clusters adjacently in memory, with no gaps between them.

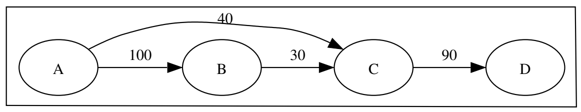
We now illustrate how C^3 processes the example from Figure 2. For simplicity, let us assume that the amount of time spent in each function equals the sum of the weights of its incoming arcs in the call graph. Therefore, C^3 starts by processing function B, which is then appended to the



(a) after 1st cluster merge



(b) after 2nd cluster merge



(c) after 3rd and final cluster merge

Figure 5. Example of C^3 heuristic processing the call graph from Figure 2.

cluster containing A. The result of this merge is illustrated in Figure 5(a). Next, function D is processed, and it is merged with the cluster containing function C. The result of this step is shown in Figure 5(b). At this point, there are two clusters left: $A; B$ and $C; D$. Next, function C is processed, and its cluster ($C; D$) is appended to cluster $A; B$, resulting in the final cluster $A; B; C; D$, which is illustrated in Figure 5(c).

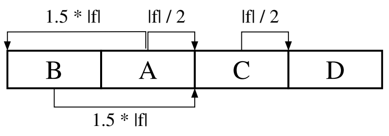
We now quantitatively compare the final layouts obtained by PH ($B; A; C; D$) and C^3 ($A; B; C; D$) for the example in Figure 2. For simplicity, we assume that all 4 functions have the same size $|f|$ and that all calls appear exactly in the middle of the caller's body (i.e. at a $|f|/2$ distance from the caller's start). Figures 6(a) and (b) illustrate the code layouts obtained using the PH and C^3 heuristics, respectively. These figures also illustrate the distances between the call instructions and their targets with both code layouts. Plugging in the arc weights from the call graph from Figure 2, we obtain the total distance jumped through the calls in each case:

cost(PH) = $100{*}1.5{*}|f| + 40{*}0.5{*}|f| + 30{*}1.5{*}|f| + 90{*}0.5{*}|f|$
$\therefore$ cost(PH) = $(150 + 20 + 45 + 45) * |f| = 260 * |f|$
cost(C^3) = $100{*}0.5{*}|f| + 40{*}1.5{*}|f| + 30{*}0.5{*}|f| + 90{*}0.5{*}|f|$
$\therefore$ cost(C^3) = $(50 + 60 + 15 + 45) * |f| = 170 * |f|$

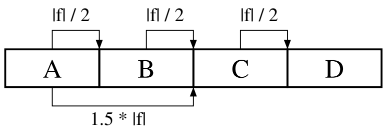
Therefore, relative to PH, C^3 results in a 35% reduction in the total call-distance in this case. In practice, as the experiments in Section 7 demonstrate, such reduction in call-distance results in a reduction in I-cache and I-TLB misses, and therefore an increase in IPC and performance.

6. Huge Pages for the Text Section

Function layout heuristics like C^3 and PH increase performance by improving the efficiency of the processor's caches, particularly the I-TLB. Once the hot functions have been clustered into a small subsection of the binary using these



(a) PH layout



(b) C^3 layout

Figure 6. Call-distances for the layouts obtained for the example from Figure 2 using (a) the PH heuristic, and (b) the C^3 heuristic.

techniques, it is possible to exploit TLB features to further reduce misses and improve performance.

Modern microprocessors support multiple page sizes for program code. For example, Intel's Ivy Bridge microarchitecture supports both 4 KB and 2 MB pages for instructions. Using these huge pages allows the processor to map larger address ranges simultaneously, which greatly reduces pressure on the I-TLB. There are usually a limited number of huge I-TLB entries available, e.g. Ivy Bridge provides only 8 entries, so it is important for programs to use these entries judiciously.

On systems that support multiple page sizes, the use of huge pages for either data or code is always an opportunity to be evaluated for large-scale applications. On Linux, the `libhugetlbfs` library [16] provides a simple mechanism to map a program's entire text section onto huge pages. However, as our experiments in Section 7.5 demonstrate, mapping the entire text section of a binary onto huge pages can put too much pressure on the limited huge I-TLB entries and thus result in a performance degradation.

In principle, one could add support for partially mapping a binary's text section onto huge pages by modifying the Linux loader. However, shipping kernel patches has its own drawbacks in practice, as it increases deployment risk and slows down the experimental process compared to an application-level solution.

To avoid this complexity, we implement huge page mapping in user code via a new library. At startup, the application copies the hot function section to scratch space, and unmaps that address range. That range is then re-mapped using anonymous huge pages, and the text is copied back in place.[2] This technique allows the application to map the hottest functions using a small number of huge pages. We measure the performance impact of this technique in Section 7.

[2] Note that unmapping `.text` will make symbols unavailable to `perf`; however this can be easily overcome by writing the static symbols to `/tmp/perf-PID.map` at startup.

7. Evaluation

This section evaluates the PH and C^3 heuristics, as well as the selective use of huge pages, on the four large-scale applications introduced in Section 2: AdIndexer, HHVM [4], Multifeed, and TAO [6].

The experimental results presented here were obtained on Linux-based servers powered by dual 2.8 GHz Intel Xeon E5-2680 v2 (Ivy Bridge) microprocessors, with 10 cores and 25 MB LLC per processor. The amount of RAM per server was 32 GB for HHVM and Multifeed, and 144 GB for AdIndexer and TAO. The applications were compiled using GCC 4.9 with -O3 optimization level.

We measured the performance of HHVM running Facebook using a custom-built performance-measurement tool. This tool first warms up HHVM's JIT and the data layer (e.g. MemCache), and then runs a measurement phase. In this phase, the server is heavily loaded with requests from a selected set of production HTTP requests, and the CPU time consumed by each request is measured. The overall result is a weighted average of the individual endpoint results representing the contribution of each endpoint to the overall production workload [17].

For AdIndexer, Multifeed, and TAO the performance results were obtained by running the same production traffic on identical machines. The performance was measured by monitoring the CPU utilization of the servers over a few hours during steady state (i.e. after they were warmed up). Detailed performance data was obtained from the hardware performance counters read through the Linux `perf` tool.

7.1 Performance Results

We gather performance results using five different configurations. The first three of these configurations correspond to the three different approaches to order the functions: the default order picked by the linker (the baseline), PH, or C^3. The other two configurations correspond to enabling the use of huge pages on top of PH and C^3. For both PH and C^3, only the hot part of the text section, which includes the functions seen during profiling, is mapped to huge pages.

Figure 7 compares the performance of our applications with the PH and C^3 function-sorting algorithms, with and without the use of huge pages, over the baseline binaries. For this comparison, we looked at both the CPU utilization and the instructions-per-cycle (IPC). For all these workloads, we found that both of these metrics perfectly correlate.

Figure 7 shows that C^3 invariably performs better than PH. Without using huge pages, C^3 achieves an average IPC improvement of 5.46%, compared to 2.64% for PH. Mapping the hot functions onto huge pages boosts the average performance improvement with PH to 6.35%, while the performance with C^3 goes up to 7.51% on average. The largest performance improvement was measured for TAO, where C^3 with huge pages improved performance by 11.0%. Overall, by using C^3 and huge pages, the average performance improvement over PH is 4.87%.

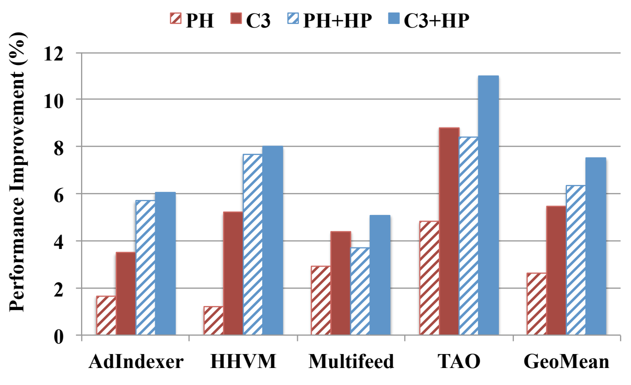


Figure 7. Performance comparison on the Facebook workloads (measured in IPC).

In all scenarios, C^3 outperforms PH. However, we notice that the benefit of C^3 over PH is smaller when huge pages are enabled. As shown in Section 7.2, this is because using huge pages greatly reduces the I-TLB footprint of the application, and so the particular ordering algorithm is somewhat less important in this scenario. We also note that, in all these scenarios, the use of huge pages was beneficial. However, as studied in Section 7.5, the use of huge pages has to be judicious otherwise it may degrade performance.

7.2 I-TLB Performance Comparison

To understand the performance improvements achieved by the different configurations, we look at detailed micro-architectural counters obtained through the Linux `perf` tool to compute misses per thousand instructions (MPKI) during steady-state execution. This section compares the effect of the different configurations on the I-TLB misses for our workloads, and Section 7.3 compares I-cache misses.

Figure 8 compares the I-TLB miss rates for the various configurations. Without huge pages, PH reduces the I-TLB misses by 32.4% on average, while C^3 reduces this metric by 44.2% on average over the baseline. This is the main effect of function sorting, and it directly correlates with the performance wins reported in Section 7.1. With huge pages, the gap between C^3 and PH on I-TLB misses is smaller (67.4% versus 65.6%, respectively). In all configurations, C^3 reduces I-TLB misses more than PH does. Similarly, the use of huge pages for the hot functions is beneficial in all cases. Overall, by combining C^3 with huge pages, we observed a 51.8% reduction in I-TLB misses over PH without huge pages.

7.3 I-Cache Performance Comparison

Figure 9 compares the I-cache performance for the various configurations. Without huge pages, C^3 always improves I-cache misses, providing an average reduction of 5.9% in this metric. PH, however, sometimes increases the I-cache miss rate, and results on a 1.7% average reduction in this metric.

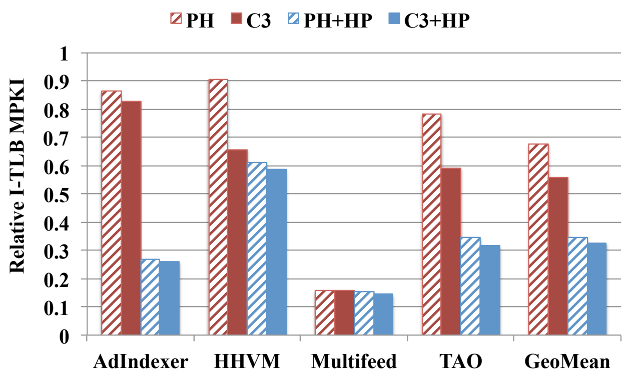


Figure 8. I-TLB performance comparison on the Facebook workloads.

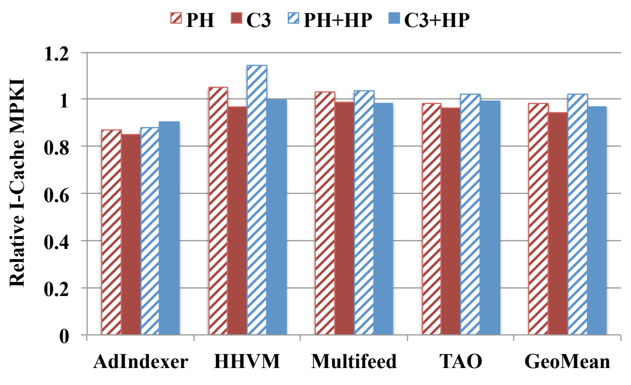


Figure 9. I-cache performance comparison on the Facebook workloads.

Another interesting observation is that the use of huge pages invariably increases the I-cache misses, on average by 3.8% with PH and 3.0% with C^3. Although we could not confirm the root-cause for this effect, we suspect it may be due to an overlap in the memory hierarchy when I-TLB misses and I-cache misses occur simultaneously and how this situation is reported through the hardware performance counters. Since huge pages significantly reduce the I-TLB misses, we suspect that, when enabling huge pages, fewer I-cache misses are masked by I-TLB misses, thus increasing the number of I-cache misses reported.

7.4 Call-Distance Comparison

In Section 5.2, we illustrated and provided some intuition about how C^3 can theoretically reduce the average call-distance compared to PH. In this section, we provide some experimental data confirming this effect in practice.

For this study, we focus on AdIndexer, HHVM, and Multifeed. We instrumented the `hfsort` tool to report various statistics after the functions have been sorted with the two different methods. The collected data is shown in Figures 10

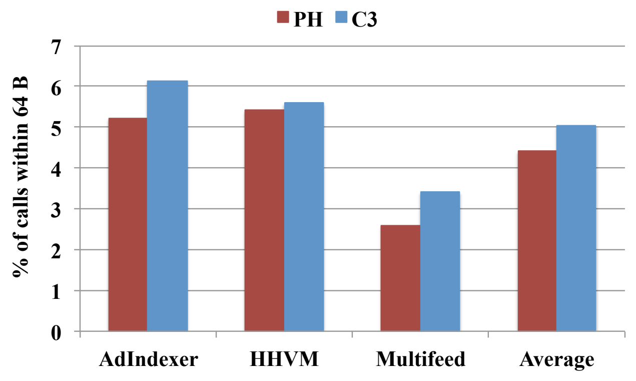


Figure 10. Percentage of calls within 64 B (cache-line size) with PH and C^3.

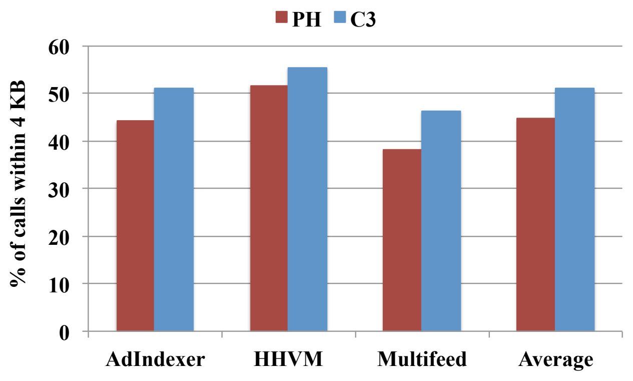


Figure 11. Percentage of calls within 4 KB (regular page size) with PH and C^3.

and 11, which show that C^3 significantly increases the number of calls with a distance of 64 B and 4 KB. On average, C^3 increases the number of calls within 64 B and 4 KB over PH by 16.8% and 14.7%, respectively. These are the calls that may turn out to be within a I-cache line or a regular memory page. The increase on these metrics aligns with C^3's benefit in reducing I-cache misses and I-TLB misses without huge pages compared to PH.

When we look at the number of calls within the 2 MB huge-page size, there is very little difference between PH and C^3. In fact, both PH and C^3 are able to keep the vast majority of the calls ($\sim$94%) within a 2 MB distance. This explains why there is a smaller gap in I-TLB misses and performance between the two sorting algorithms when huge pages are used.

7.5 Effect of Huge Pages without Function Sorting

In all the experiments above, the use of huge pages was restricted to the hot functions in the `.text` section of the binary, i.e. those that where sampled at least once during the profiling collection. However, it is possible to apply huge pages without profiling information. In this section, we evaluate the approach of mapping the entire `.text` section onto huge pages without clustering the hot functions.

We performed this study for the HHVM binary, mapping all its 133 MB of static binary onto huge pages, and preserving the functions in the default order picked by the linker. Our evaluation of this version of HHVM revealed a 1.15% performance regression running Facebook web traffic when compared to the baseline with the same function order and without mapping any function to huge pages. Analyzing the I-TLB behavior, we observed that mapping all the text section onto huge pages doubled the number of I-TLB misses compared to the baseline. Aggressively mapping too many pages onto huge pages puts too much pressure on the limited huge-page I-TLB entries — Intel IvyBridge has only 8 huge-page I-TLB entries, organized in a single level. Therefore, even though our experiments in Section 7.1 showed that combining huge pages with hot-function clustering is beneficial, the simple approach of mapping all the text section onto huge pages can be detrimental to the performance of large binaries like HHVM. Besides this performance issue, our experience mapping too many pages as huge in production systems has revealed that the Linux kernel starts to misbehave after some time, being overwhelmed by the need to relocate physical pages to satisfy requests for huge pages. Overall, our experience mapping functions onto huge pages proved that they need to be used judiciously, such as in combination with hot-function clustering as applied in this work.

7.6 Comparison with `gprof`

The `gprof` profiling tool [13] implements a variant of the PH algorithm, available through the `---function-ordering` option. This implementation is similar to PH in the that it clusters functions without paying attention to their roles (caller vs. callee) like C^3 does. However, `gprof`'s technique is different from PH because it also partitions the functions based on how hot they are and their number of callers. Basically, the hottest functions with the most callers are placed first, followed by other hot functions, followed by colder functions, finally followed by unused functions at the end [13].

In order to evaluate `gprof`'s technique on the same data collected for `hfsort`, we wrote a small tool to convert the `perf` data that feeds `hfsort` into `gprof`'s `gmon` format [13]. These data was passed to `gprof --function-ordering`, which outputs its selected list of functions. This list of functions was then passed to the linker via a linker script, in the same way that the list of functions generated by `hfsort` is used.

For this evaluation, we focused on AdIndexer, HHVM, and Multifeed, with and without the use of huge pages. Figure 12 presents the results, along with the data presented for PH and C^3 in Section 7.1. Without huge pages, `gprof` performs slightly better than PH, with a 0.44% advantage on average, but still significantly worse than C^3. However,

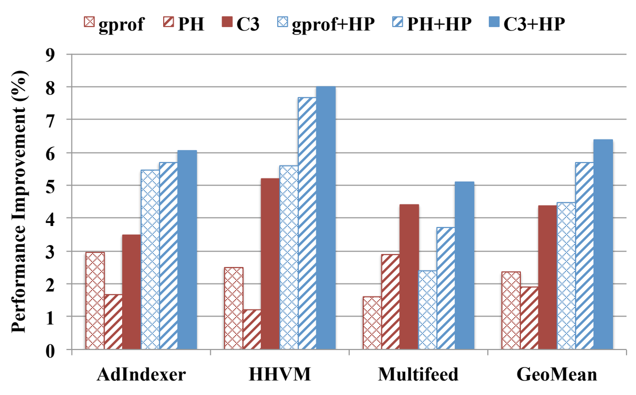


Figure 12. Performance comparison of gprof, PH, and C^3 function-sorting algorithms.

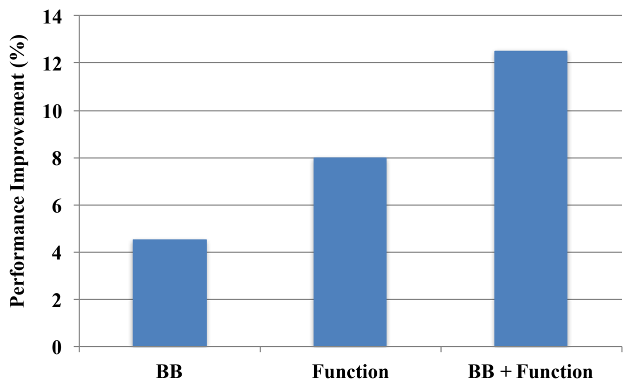


Figure 13. Performance comparison of basic-block and function reordering (measured in IPC).

with huge pages, `gprof` performs significantly worse than the other techniques. We found that this is due to the fact that `gprof` lists even unused functions. As shown in Section 7.5, mapping too many pages as huge can have a negative effect. Also, we note that, in order to enable the linker to successfully place the functions in a reasonable amount of time ($\sim$ 1 hour), we had to limit the number of functions output by `gprof` to 20,000. Based on the observation made in Section 7.5, we suspect that, had we been able to successfully link the binaries with all functions output by `gprof`, the binaries would perform even worse, since this would potentially increase the pressure on the processor's 2 MB I-TLB entries even further.

7.7 Interaction with Basic-Block Reordering

Many compilers and binary-optimization tools use profiling data to reorder basic blocks within a function, with the goals of improving I-cache performance and branch prediction. Such techniques are implemented in GCC and production compilers from Intel and Microsoft, for example. Most of these techniques are based on code instrumentation to obtain profiling data, which makes them less attractive from a deployment point of view. One interesting alternative is AutoFDO [1], which is based on non-intrusive profiling via hardware performance counters, similar to what we use in this paper. In this section, we evaluate our function-sorting technique in combination with basic-block reordering and hot-cold splitting.

Unfortunately, AutoFDO support in the latest GCC version currently available in our environment (GCC 4.9) is not very stable, and we have encountered correctness issues related to C++ exceptions when applying AutoFDO to our binaries. So, instead, in this evaluation we use an internal binary-optimization tool [18] based on LLVM [19], which is in advanced development stage at this point. This tool uses last-branch record (LBR) profiling data collected with the Linux `perf` tool to reorder the basic blocks in a linked binary, including moving cold blocks to a separate section.

Figure 13 shows the results of our comparison on the HHVM binary. These data shows that, although both techniques aim at improving code locality, they play complementary roles and their benefits are additive. When applied alone, basic-block reordering provides a 4.5% performance improvement for HHVM, while function reordering with huge pages provides an 8.0% improvement. When applied in combination, the two techniques together improve HHVM's performance by 12.5%.

8. Related Work

This section describes some of the previous related work on changing the binary layout in order to improve performance. Performance can be improved in different ways, by improving the behavior of different architectural or microarchitectural features. So we break down the related work roughly based on the hardware feature that they try to improve the behavior.

Our work increases performance by improving code locality, which primarily reduces I-TLB misses, and also I-cache misses. The seminal work by Pettis and Hansen [3] tackles the same problems, but with different approaches. In fact, they proposed three different techniques, one of which is the reordering of functions. That technique is the PH heuristic described in Section 5.1 in this paper, and which we extensively compared against C^3 in Section 5.2 and Section 7. This algorithm is commonly used in practice, having been implemented in compilers, binary optimizers, and performance tools [3, 12–14]. Besides function reordering, PH also proposed two other intra-procedural transformations to improve code locality: a technique to improve the layout of a function by reordering its basic blocks, and a hot-cold splitting technique that separates the cold blocks of a function separate from its hot blocks. These techniques are implemented in the binary optimizer used in the evaluation in Section 7.7 [18] and, as our evaluation demonstrated, are orthogonal to function reordering.

The GCC compiler [8] supports function reordering via its link-time optimizer (LTO), which simply partitions functions into two sections (`.text.hot` and `.text.unlikely`) based on profiling data to segregate the hot and cold functions (via `--freorder-functions`). Google's branch of GCC contains a linker plug-in to reorder functions using the PH algorithm [15]. We suspect that the Intel and Microsoft production compilers might implement the PH heuristic to reorder functions, although we could not verify this information because those are not open-source projects and we have not found any public information about their specific techniques.

Ramirez et al. [12] specifically studied the impact of code layout optimizations on transaction-processing workloads. Their workloads were also very large and significantly suffered from code locality issues. To improve code locality, Ramirez et al. used the Spike binary optimizer [14], which uses a straightforward implementation of the PH heuristic [3].

LIPO [2] describes a lightweight feedback-driven framework for cross-module optimizations. They mention that function reordering was ones of the attempted optimizations, but it provided no benefit on SPEC-CPU2000 benchmarks. It is unclear what reordering technique was implemented and whether it was evaluated on larger applications.

Besides [3], various previous work have investigated the use of compiler optimizations to improve the code layout within a function with the goal of improving performance, by trying to either increase locality or reduce cache conflicts [20–22]. Boehm et al. [23] explored aggressive inlining and hot-cold code splitting in a post-link optimizer [24] that uses binary instrumentation to obtain a dynamic call graph.

Li et al. [25] use code layout to improve cache defensiveness and politeness among multiple applications executing in a shared cache environment. Therefore, their optimization goal is different from ours, which is to improve performance of a single application. Furthermore, they show that techniques that improve one of these goals do not necessarily improve the other. Also, they do not use a dynamic call graph, but instead they use a temporal-relation graph, which requires more expensive, instrumentation-based profiling.

Another opportunity for improving performance via code layout is to try to improve the behavior of the processor's branch predictors. Several works have explored this opportunity, including [26–29].

In addition to code layout optimizations to improve micro-architectural behavior, much research has focused on improving performance by data layout and transformations [30–33]. Notably, Raman et al. [30] builds a field layout graph representing the access affinity of struct fields, and optimizes the layout by clustering nodes to maximize locality while minimizing false sharing. While the problem domain is different (and more constrained), the use of graph clustering to improve locality is similar.

The drawbacks of instrumentation-based profiling, with its required two compilation steps and high profiling overheads, have long been recognized as the main reason why feedback-driven optimizations are not adopted more widely. This has motivated the recent work on SampleFDO [34] and AutoFDO [1], which instead rely on hardware-event sampling data to guide feedback-driven optimizations. These approaches required careful engineering in order to obtain adequate sampling accuracy at the instruction and basic-block levels, since the profile data are used to guide intra-procedural compiler optimizations in those work. In our work, this is less of an issue because our tool operates at a coarser granularity, by reordering whole functions.

9. Conclusion

This paper studied the impact of function placement for large-scale data-center applications. This study focused on four highly optimized applications used to run some of the largest websites in the world, including Facebook, Baidu, and Wikipedia. This investigation was conducted using sampling-based profiling and `hfsort`, which is an open-source tool we built to reorder functions based on profile data. Our experiments showed that the traditional function-ordering algorithm by Pettis and Hansen [3] improves the performance of these applications by 2.6% on average. In this paper, we also identified a weakness in this widely used algorithm, and then describe a new algorithm, C^3, that we designed based on this insight. Our experimental evaluation demonstrated that C^3 further improves the performance of the studied applications by an average of 2.9% on top of Pettis and Hansen's algorithm. Finally, this paper described a simple technique to selectively map portions of a binaries text section onto huge pages on Linux. Our experimental evaluation demonstrated that, when combined with C^3, this technique further improves the performance of the studied applications by an average of 2.0%.

Although the presented techniques were studied in the context of `hfsort` and a set of four applications, we believe that the benefits of both C^3 and selective use of huge pages are applicable in general and these techniques can benefit other tools and applications, including smaller applications running on systems with more constrained caches and TLBs. Finally, future work can also investigate the potential of applying these techniques to dynamically generated code.

Acknowledgments

We would like to thank Mark Williams for contributing the technique to enable huge pages for the text section described in Section 6, Taewook Oh for implementing the Pettis-Hansen algorithm in `hfsort`, and Denis Sheahan, Pallab Bhattacharya, and Mark Santaniello for helping with performance experiments. We also thank the anonymous reviewers for their valuable feedback.

References

[1] D. Chen, D. X. Li, and T. Moseley, "AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 12–23, 2016.

[2] X. D. Li, R. Ashok, and R. Hundt, "Lightweight feedback-directed cross-module optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 53–61, 2010.

[3] K. Pettis and R. C. Hansen, "Profile guided code positioning," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 16–27, 1990.

[4] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi, "The Hiphop Virtual Machine," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pp. 777–790, 2014.

[5] Alexa, "The top 500 sites on the web." Web site: http://www.alexa.com/topsites, May 2015.

[6] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "Tao: Facebook's distributed data store for the social graph," in *Proceedings of the USENIX Conference on Annual Technical Conference*, pp. 49–60, 2013.

[7] G. Ottoni, "hfsort: a tool to sort hot functions." Web site: https://github.com/facebook/hhvm/tree/master/hphp/tools/hfsort.

[8] GCC Team, "Gnu Compiler Collection." Web site: http://gcc.gnu.org.

[9] I. L. Taylor, "A new ELF linker," in *Proceedings of the GCC Developers' Summit*, 2008.

[10] Intel Corporation, *Intel$^{®}$ 64 and IA-32 Architectures Software Developer's Manual*. No. 325384-039US, May 2011.

[11] E. Petrank and D. Rawitz, "The hardness of cache conscious data placement," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 101–112, 2002.

[12] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J. Larriba-Pey, P. G. Lowney, and M. Valero, "Code layout optimizations for transaction processing workloads," in *Proceedings of the Annual International Symposium on Computer Architecture*, pp. 155–164, 2001.

[13] J. Fenlason and R. Stallman, "GNU gprof: the GNU profiler," *Manual, Free Software Foundation Inc*, 1997.

[14] R. Cohn, D. Goodwin, and P. G. Lowney, "Optimizing Alpha executables on Windows NT with Spike," *Digital Technical Journal*, vol. 9, no. 4, pp. 3–20, 1997.

[15] S. Tallam, "Linker plugin to do function reordering using callgraph edge profiles." https://gcc.gnu.org/ml/gcc-patches/2011-09/msg01440.html, September 2011.

[16] libhugetlbfs Team, "libhugetlbfs." Web site: https://github.com/libhugetlbfs.

[17] E. Bakshy and E. Frachtenberg, "Statistics and optimal design for benchmarking experiments involving user traffic," in *Proceedings of the International World Wide Web Conference*, pp. 108–118, 2015.

[18] M. Panchenko, "Building a binary optimizer with LLVM," *European LLVM Developer's Meeting*, March 2016.

[19] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 75–86, 2004.

[20] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proceedings of the Annual International Symposium on Computer Architecture*, pp. 242–251, 1989.

[21] S. McFarling, "Program optimization for instruction caches," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 183–191, 1989.

[22] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder, "Procedure placement using temporal ordering information," in *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 303–313, 1997.

[23] O. Boehm, D. Citron, G. Haber, M. Klausner, and R. Levin, "Aggressive function inlining with global code reordering," Tech. Rep. H-0247, IBM Research, November 2006.

[24] I. Nahshon and D. Bernstein, "FDPR - a post-pass object code optimization tool," in *Proceedings of the International Conference on Compiler Construction*, pp. 97–104, April 1996.

[25] P. Li, H. Luo, C. Ding, Z. Hu, and H. Ye, "Code layout optimization for defensiveness and politeness in shared cache," in *Proceedings of the International Conference on Parallel Processing*, 2014.

[26] D. A. Jiménez, "Code placement for improving dynamic branch prediction accuracy," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 107–116, 2005.

[27] B. Calder and D. Grunwald, "Reducing branch costs via branch alignment," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 242–251, 1994.

[28] F. Mueller and D. B. Whalley, "Avoiding conditional branches by code replication," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 56–66, 1995.

[29] M. Yang, G.-R. Uh, and D. B. Whalley, "Efficient and effective branch reordering using profile data," *ACM Trans. Program. Lang. Syst.*, vol. 24, pp. 667–697, Nov. 2002.

[30] E. Raman, R. Hundt, and S. Mannarswamy, "Structure layout optimization for multithreaded programs," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 271–282, 2007.

[31] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 139–149, 1998.

[32] T. M. Chilimbi and R. Shaham, "Cache-conscious coallocation of hot data streams," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 252–262, 2006.

[33] K. S. McKinley, S. Carr, and C.-W. Tseng, “Improving data locality with loop transformations,” *ACM Trans. Program. Lang. Syst.*, vol. 18, pp. 424–453, July 1996.

[34] D. Chen, N. Vachharajani, R. Hundt, X. Li, S. Eranian, W. Chen, and W. Zheng, “Taming hardware event samples for precise and versatile feedback directed optimizations,” *IEEE Transactions on Computers*, vol. 62, no. 2, pp. 376–389, 2013.