One Profile Fits All: Profile-Guided Linux Kernel Optimizations for Data Center Applications

Muhammed Ugur, Cheng Jiang, Alex Erf, Tanvir Ahmed Khan, Baris Kasikci
University of Michigan

Abstract

Modern data center applications have multi-megabyte instruction footprints that easily exhaust on-chip cache structures, which typically have a size of only a couple hundred kilobytes. Consequently, today's data center applications suffer from frequent Instruction cache (I-cache) and Instruction Translation Lookaside Buffer (I-TLB) misses, causing performance losses worth millions of dollars. To make matters worse, the multi-megabyte instruction footprint of the Linux kernel precipitates an undue burden on the performance of data center applications.

In this paper, we perform a comprehensive characterization of the Linux kernel's control-flow behavior for 8 data center applications from different domains and observe that these applications show close resemblance in their usage of Linux kernel features. Based on this insight, we combine Linux kernel execution profiles from different data center applications to generate a "universal" profile which we use to optimize the code layout of the Linux kernel. Our evaluation shows that profile-guided optimizations of the Linux kernel using this universal profile achieve an average end-to-end application speedup of 8.02% for 8 data center applications.

1 Introduction

Modern data center applications have large instruction footprints due to complex and deep software stacks [8,9,12,15,23, 26, 30]. Recent studies from Google and Facebook estimate that the typical instruction footprints for these applications range from tens to hundreds of megabytes [9,26]. On the other hand, on-chip cache structures like Instruction cache (I-cache) and Instruction Translation Lookaside Buffer (I-TLB) range only up to hundreds of kilobytes [16]. As a result, today's processors face frequent I-cache and I-TLB misses while running data center applications [8,12,15,47] and incur millions of dollars in energy and management costs [9]. Consequently, even a single-digit speedup for these widely-deployed applications substantially reduces data centers' Total Cost of Ownership (TCO) and planet-scale carbon footprint [30].

The significance of reducing I-cache and I-TLB misses for data center applications has inspired researchers from both academia and industry to propose code layout optimizations that improve instruction locality [11, 13, 17, 19–26, 28, 31]. These techniques monitor the execution of data center applications in production and leverage production profiles to reorder basic-blocks and functions in a profile-guided manner. Prior works [11, 25] have shown the effectiveness of these profile-guided optimizations (PGO) in achieving 7-11% end-to-end speedup for real-world data center applications. Consequently, these techniques are widely deployed in today's data centers [9, 11].

Despite the widespread adoption of PGO techniques for data center applications, optimizing the Linux kernel in a profile-guided manner has received little attention from both academic and industry researchers [5, 7, 29, 43–45]. To close this gap, we investigate the implications of profile-guided kernel optimizations for data center applications in this paper. Specifically, we examine the kernel usage for 8 real-world data center applications and observe that these applications spend 43-74% (61% on average) of their overall execution time executing instructions within the kernel. Our characterization also reveals that 43-79% of all I-cache misses (61% on average) and 10-86% (46% on average) of all I-TLB misses for these data center applications originate from the kernel. Thus, our investigation shows the significance of optimizing the Linux kernel's multi-megabyte code footprint for data center applications.

Encouraged by these results, we examine the kernel profiles for different data center applications to measure the similarity and variation across different profiles. In particular, we study how the control-flow behavior of the Linux kernel varies across different data center applications since this behavior dominates how the code layout is reordered in a profile-guided manner. To this end, we leverage state-of-the-art measures of similarity metrics, including cosine similarity and L_p-norms, and show that these 8 applications exhibit close resemblance in their usage of the kernel's control-flow behavior.

Table 1: Data center applications, their versions, and their benchmarks we study.

Applications	Versions	Benchmarks
`Apache` [35]	2.4.41	ApacheBench [32]
`Nginx` [39]	1.18	ApacheBench [32]
`Redis` [41]	5.0.7	Redis benchmark [6]
`Memcached` [37]	1.5.22	Redis benchmark [3,6]
`LevelDB` [33]	1.22	db_bench [4]
`RocksDB` [42]	6.15.2	db_bench [4]
`MySQL` [38]	8.0.23	sysbench [2]
`PostgreSQL` [40]	12.5	sysbench [2]

Driven by our characterization's insight, we combine multiple kernel profiles from different data center applications and generate a "universal" kernel profile. Furthermore, we optimize the Linux kernel using this universal profile and evaluate the effectiveness of the optimization in the context of 8 data center applications. In our evaluation, universal profile-guided kernel optimizations achieve an average end-to-end speedup of 8.02% for our selected data center applications. Our evaluation also shows that optimizations using a universal kernel profile achieve an almost identical speedup to optimizations performed using application-specific Linux kernel profiles.

In this paper, we make the following contributions:

- We perform a comprehensive characterization of the Linux kernel's control-flow behavior across 8 real-world data center applications and show that profile-guided optimizations of the Linux kernel have significant potential to improve data center performance in a generalized way.
- We combine kernel profiles from multiple applications to create a universal profile and evaluate the effectiveness of using this universal profile to optimize the Linux kernel – providing a substantial speedup for data center applications.

2 Understanding the kernel's usage for modern data center applications

2.1 Experimental methodology

Data center applications. We investigate the Linux kernel's usage for modern data center applications using 8 open-source applications that are widely-deployed in today's data centers. Table 1 lists these data center applications and their benchmarks we study. We analyze two web server applications (`Apache` and `Nginx`), two in-memory caching systems (`Redis` and `Memcached`), two on-disk key-value databases (`LevelDB` and `RocksDB`), and two relational database management systems (`MySQL` and `PostgreSQL`).

Data collection methodology. We conduct our experiments in Ubuntu 20.04 LTS using Linux kernel 5.11. We build and optimize the Linux kernel in a profile-guided manner using LLVM (13.0.0). To collect the kernel's execution profile, we leverage the instrumentation-driven `gcov` [36] tool, `clang` [1], and sampling-based Linux `perf` [34] tool. We use a 12-core 1.10GHz Intel(R) Core(TM) i7-10710U machine, with 64KB of L1-cache (32KB instruction and 32KB data), 256KB of L2-cache, 20MB of L3-cache (shared across the same NUMA node), and 16 GB of RAM.

Profile comparison metrics. We analyze the usage of the Linux kernel across different applications by comparing the kernel execution profiles for these applications. To do this comparison, we use several state-of-the-art similarity and difference metrics from data mining and machine learning literature. Such metrics include confusion matrices, cosine similarity, and L_p norms. We provide a brief description of these metrics in the next sections.

The Linux kernel's execution profile consists of several control-flow information, including execution frequencies for functions/branch instructions and taken/not-taken (*i.e.*, fall-through) frequencies for branch instructions. To understand how the kernel's profile varies across different applications, we compare this control-flow information using confusion matrices. To create a confusion matrix for control-flow information, we first create a feature vector for each application using a common (ordered) set of features. For example, the feature vector for function execution frequency is the number of executions for each kernel function invoked by the application. Similarly, the feature vector for branch execution frequency is the number of executions for each branch instruction invoked by the application in the kernel. Next, we compare these feature vectors using cosine similarity and L_p norms and create confusion matrices.

Cosine similarity. The first measure of similarity we use is cosine similarity. Let $\boldsymbol{u}$ and $\boldsymbol{v}$ be the feature vectors for the applications i and j respectively. The cosine similarity is then defined as,

$$s(\boldsymbol{u},\boldsymbol{v}) = \frac{\boldsymbol{u}\cdot\boldsymbol{v}}{\|\boldsymbol{u}\|\|\boldsymbol{v}\|},$$

where $\boldsymbol{u}\cdot\boldsymbol{v}$ is the dot product between the feature vectors, $\boldsymbol{u}$ and $\boldsymbol{v}$, and $\|\cdot\|$ is the norm for a given feature vector. Cosine similarity has the range between zero and one, where one represents two perfectly aligned feature vectors and zero represents two feature vectors that are orthogonal to each other. Geometrically, cosine similarity can be interpreted as the angle between two *unit* vectors. This makes it natural to use cosine similarity to evaluate any metric that could be normalized to one, such as function execution frequency and branch execution frequency.

L_p norms. The second metric we consider is L_p norms. L_p norms of vector differences are defined as,

$$s(\boldsymbol{u},\boldsymbol{v}) = \|\boldsymbol{u}-\boldsymbol{v}\|_p = \left(\sum_{k=1}^{|\boldsymbol{v}|} |\boldsymbol{u}_k - \boldsymbol{v}_k|^p\right)^{\frac{1}{p}},$$

where $|\cdot|$ is the length of the feature vector. Note that for our analyses, $|\boldsymbol{u}| = |\boldsymbol{v}|$. The norm of vector differences has a

Data center applications	% of cycles spent in kernel	% of instructions executed in kernel	% of I-cache misses in kernel	% of I-TLB misses in kernel
Apache	62.88%	64.55%	60.37%	63.11%
Nginx	74.02%	74.86%	68.3%	80.45%
Redis	69.84%	56.96%	78.88%	85.55%
Memcached	69.07%	55.04%	74.31%	51.62%
LevelDB	43.05%	30.27%	56.67%	31.7%
RocksDB	46.34%	41.79%	42.97%	18.19%
MySQL	63.26%	59.21%	64.86%	10.38%
PostgreSQL	60.00%	73.22%	43.85%	24.40%

Table 2: Hardware performance event (*e.g.*, CPU cycles, instructions, I-cache misses, and I-TLB misses) statistics for data center applications: Linux kernel is the root cause for a large fraction (46-61%) of hardware issues including I-cache and I-TLB misses for data center applications.

lower bound of zero, which occurs when the two input feature vectors are identical. The input to vector difference norm does not need to be normalized to unit vectors. Therefore, L_p norms are best suited for cases where the feature elements are bounded. For example, we use L_p norms to compute the similarities between different applications' branch taken and not-taken execution frequencies. We evaluated L_p norms for $p = \{0, 1, 2\}$. L_0-norm corresponds to the total number of nonzero elements in a vector. In our analysis, it represents the number of different elements in the feature vector. L_1-norm is also known as Manhattan distance, which sums the absolute difference in each component of the feature. L_2-norm is also known as Euclidean distance. Compared to L_1-norm, L_2-norm shows tolerance toward small discrepancies. Also, L_2-norm penalizes relatively large differences between features.

2.2 Is the kernel a bottleneck for modern data center applications?

We start our characterization of the Linux kernel by measuring what percentage of the overall execution time for data center applications is spent in the kernel. Specifically, we estimate the percentages of overall CPU cycles, executed instructions, I-cache misses, and I-TLB misses that originate from the Linux kernel for each application. Table 2 shows the results.

As shown in Table 2, these data center applications spend, on average, 61% of their total CPU cycles and 57% of all executed instructions in the kernel. Additionally, on average, 61% of all I-cache misses and 46% of all I-TLB misses originate from the kernel for these 8 applications. Therefore, we conclude that the Linux kernel can be a significant hardware performance bottleneck for data center applications.

2.3 How does the kernel's control-flow behavior vary across different applications?

Since the kernel is responsible for a large fraction of the overall execution time for data center applications, optimizing the end-to-end performance for these applications must aim to improve the performance of the Linux kernel. To improve the Linux kernel's performance, we mainly consider compiler-based profile-guided optimizations. These profile-guided optimizations change the code layout of a program via basic-block reordering, function reordering, and function splitting based on the program's execution profile. Such profile-guided optimizations provide significant performance benefits for a program if the profile used for optimization matches closely with the execution profile for the common case. Therefore, we examine how the Linux kernel's profile varies across different data center applications in this subsection.

Function reordering and function splitting primarily leverage the execution frequencies of different functions whereas basic-block reordering uses the execution frequencies of different branch instructions along with the taken and not-taken frequencies of different conditional branch instructions. Consequently, we study the similarity and variation in kernel profiles for different applications based on execution frequencies for all functions and branch instructions along with taken and not-taken frequencies for conditional branch instructions.

Function execution frequencies. We first study the variation in kernel profiles based on all function execution frequencies in the Linux kernel. For each application, we create a feature vector of function frequencies as described in §2.1. Then, we normalize these feature vectors so that each of these vectors has a unit norm. Next, we calculate the cosine similarity among different feature vectors from different data center applications and show the confusion matrix in Fig. 1.

As shown in Fig. 1, almost all these data center applications show close resemblance with each other based on their kernel function usage. Only LevelDB and RocksDB noticeably differ from other applications. however, the minimum similarity score is only 0.37. This suggests that even for applications with the widest function usage diversity, there is still a 37% match among their profiles. Consequently, profile-guided function reordering and splitting for the kernel should provide performance benefits across different applications based on these execution frequencies.

Branch instruction execution frequencies. Next, we examine kernel profiles for these data center applications based on the execution frequencies of different branch instructions. Similar to the execution frequencies of functions, we generate a feature vector for each application based on the execution frequencies of all branch instructions from the Linux kernel. After normalizing these feature vectors to have a unit norm, we measure the cosine similarity among different feature vectors of different applications and show the confusion matrix in Fig. 2.

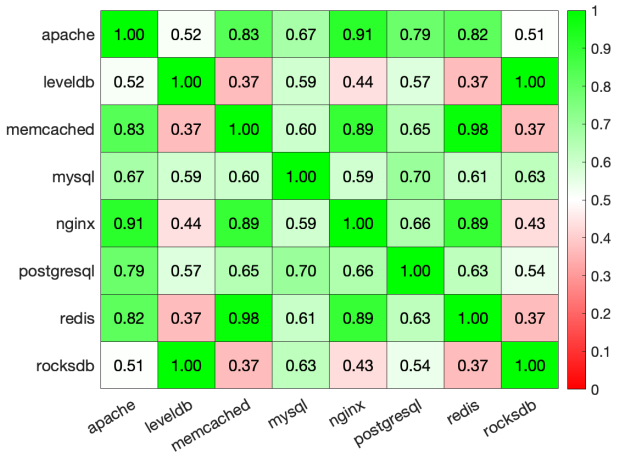

Figure 1: Confusion matrix using cosine similarities for different data center applications based on the Linux kernel's function execution frequencies: apart from `LevelDB` and `RocksDB`, all other applications exhibit close similarity in their kernel function usage.

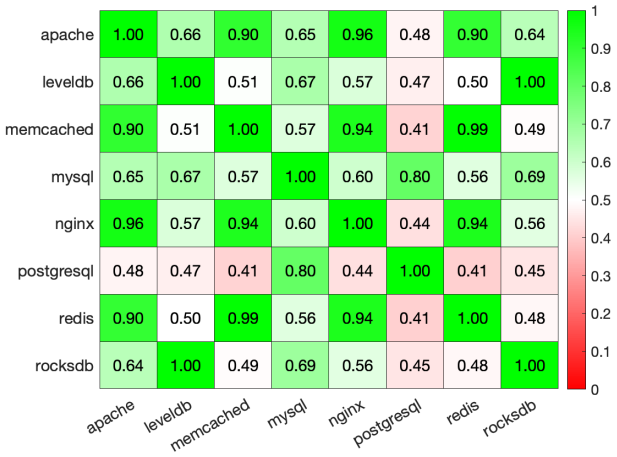

Figure 2: Confusion matrix using cosine similarities for different data center applications based on the Linux kernel's branch execution frequencies: apart from `PostgreSQL`, all other applications exhibit close similarity in their usage of branch instructions from the kernel.

As shown in Fig. 2, we observe that kernel profiles for these applications, with the exception of `PostgreSQL`, are similar based on branch instruction execution frequency. However, even for `PostgreSQL`, the minimum score for the measure of similarity is 0.41 suggesting that there is still a 41% match in branch instruction frequency for these applications. Apart from execution frequencies of branch instructions, profile-guided basic-block reordering also depends on the taken and non-taken frequencies of conditional branch instructions. Hence, we next investigate the taken and not-taken frequencies of conditional branch instructions from the Linux kernel for these applications.

Taken and not-taken frequencies of conditional branch instructions from the kernel. Conditional branch instructions have two possible directions: (1) the taken direction or (2) the not-taken or fall-through direction. Since the sum of the taken and not-taken frequency for a given branch is equal to the total execution frequency, we represent taken and not-taken frequencies simultaneously with a single parameter, the *taken probability*. We measure the taken probability for a given conditional branch as the ratio of the branch's taken frequency divided by the branch's total execution frequency.

To compare the Linux kernel's profile across different data center applications, we create a feature vector for each application based on taken probabilities for all conditional branch instructions from the kernel. As each entry to these feature vectors ranges from zero to one, we measure the variation among these feature vectors using L_p-norms, as described in §2.1. We show the corresponding confusion matrix using L_p-norms for different data center applications in Fig. 3.

As shown in Fig. 3, the L_0-norm and L_1-norm values are small across different data center applications, which highlights that these applications do not exhibit wide diversity among their usage of conditional branch instructions within the kernel. Specifically, the maximum L_0-norm value is 0.33 while the maximum L_1-norm value is only 0.16. Additionally, L_2-norm values across all applications are always smaller than 0.005, and hence, we do not include the corresponding confusion matrix in this paper.

Since taken/not-taken frequencies of all conditional branch instructions along with execution frequencies of all branch instructions exhibit similar behavior across different applications, we conclude that profile-guided basic-block reordering for the kernel using only one application's profile will still provide a substantial performance benefit for a different application. Next, we describe how we combine kernel profiles from all applications to generate a universal profile that can achieve significant performance benefits for all applications.

3 Implementation

We leverage LLVM's profile tools to generate a universal profile and `clang` [1] to optimize the Linux kernel using the universal profile. To generate the universal profile, we combine kernel profiles from different data center applications using the `merge` option of the tool, `llvm-profdata`. While merging the kernel profiles for different data center applications, we assign equal weights to each application's profile.

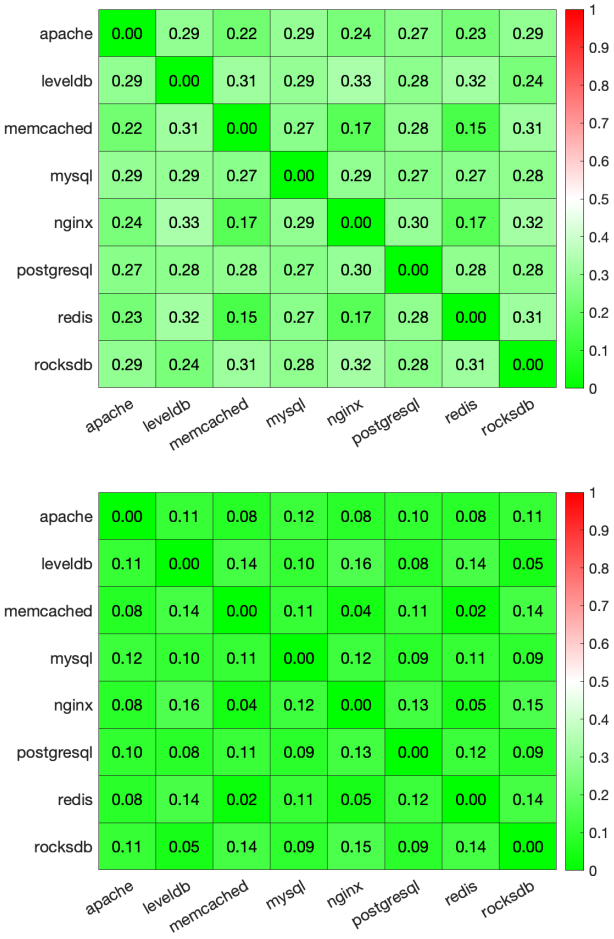

Figure 3: Confusion matrix using L_0-norm (top) and L_1-norm (bottom) for different data center applications based on taken and not-taken frequencies for all branch instructions from the Linux kernel: data center applications show little variation in their usage of kernel conditional branch instructions.

4 Evaluation

In this section, we evaluate the effectiveness of optimizing the Linux kernel using the universal profile that we generate by combining kernel profiles from different data center applications. Specifically, we use `clang` [1]'s profile-guided optimizations on the Linux kernel and measure the end-to-end speedup (improvement in throughput) for our data center applications. For comparison, we also create application-specific kernels for each application and measure each application's end-to-end speedup using this application-specific Linux kernel. We show the speedup for the universal (merged) profile and application-specific profile compared to a non-optimized (default) kernel for each application in Fig. 4.

As shown in Fig. 4, the universal profile-guided optimizations achieve significant performance speedup for almost all

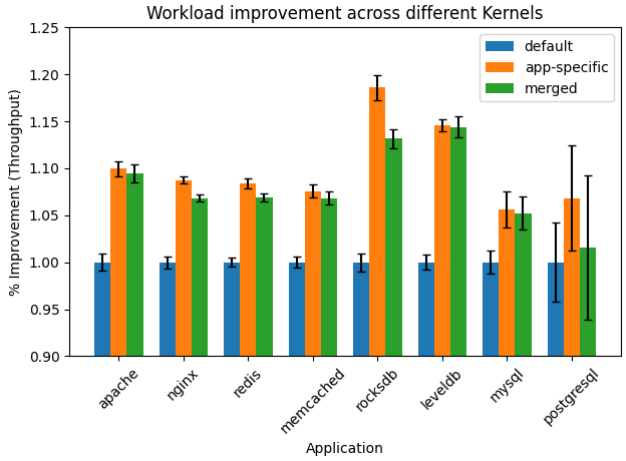

Figure 4: End-to-end speedup provided by profile-guided Linux kernel optimizations for different data center applications: Linux kernel optimizations using the universal (merged) profile provide comparable performance to kernel optimizations using application-specific profiles.

of these data center applications apart from `PostgreSQL`. For `PostgreSQL`, optimizations using the universal profile only provide a 1.59% average speedup since the branch instruction execution frequencies from the kernel for `PostgreSQL` differ from their execution frequencies for the remaining applications. On average, the universal profile-guided optimizations achieve an average end-to-end speedup of 8.02% across all applications, which is close to the average speedup of 10.03% that the application-specific profiles provide.

5 Related Work

Profile-guided optimizations of data center applications. Large instruction footprints of modern data center applications make a compelling case for optimizing these applications in a profile-guided manner [8, 9, 12, 15, 30]. Consequently, a plethora of recent techniques [11, 13, 17, 19–26, 28, 31] optimize the performance of data center applications by applying basic-block reordering, function reordering, and function splitting in a profile-guided manner. Unfortunately, all of these widely-deployed techniques only operate on data center applications themselves and have largely neglected the performance implications of the kernel. Hence, in this work, we investigate the performance potential of profile-guided kernel optimizations and show that such optimizations provide notable speedup for widely-used data center applications.

Profile-guided optimizations of Linux kernel. Previous work for Linux kernel PGO focuses on creating application-specific kernels and re-writing the kernel binary [27, 29, 43–45]. The main limitation of these works is their focus on the implementation of kernel PGO and the feasibility of

application-specific kernels. Our work instead focuses on analyzing kernel usage across different applications and generating a single universal profile to optimize all of these applications. Previous work has also investigated binary-level techniques to reduce a kernel's memory footprint for specific applications [10, 18]. Our work focuses solely on improving performance (*i.e.*, application latency and throughput) through traditional compiler-based optimizations.

Measuring profile similarity. Prior works [11, 14, 31, 46] on measuring profile similarity and diversity primarily investigate how execution profiles vary across different inputs or versions of an application and focus on each application on its own. Instead, our work leverages state-of-the-art measures of similarity metrics to compare kernel profiles across different applications and focuses on the operating system kernel, which is a unique program that supports many diverse workloads.

6 Conclusion

Modern data center applications lose significant performance potential due to frequent I-cache and I-TLB misses. In this paper, we showed that a large fraction of these misses emerge from the Linux kernel. Consequently, we investigated the implications of profile-guided kernel optimizations for these applications and observed that kernel profiles exhibit close similarities across different data center applications. Based on this insight, we combined the kernel profiles from different applications to generate a universal profile. We then optimized the Linux kernel using compiler-based profile-guided optimizations with this profile to improve application performance. In our evaluation, Linux kernel optimizations using the universal profile achieved an average end-to-end speedup of 8.02% for 8 widely-used data center applications.

References

[1] Clang c language family frontend for llvm. [Online; accessed 19-Nov-2021].

[2] Github - akopytov/sysbench: Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench.

[3] Github - antirez/mc-benchmark: Memcache port of redis benchmark. https://github.com/antirez/mc-benchmark.

[4] Github - benchmarking tools: facebook/rocksdb wiki. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[5] Profile-guided optimization for the kernel. https://lwn.net/Articles/830300/.

[6] Redis benchmark - redis. https://redis.io/docs/reference/optimization/benchmarks/.

[7] Optimizing linux kernel with bolt. https://lpc.events/event/11/contributions/974/, 2021.

[8] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 643–656. IEEE, 2018.

[9] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the 46th ISCA*, 2019.

[10] Dominique Chanet, Bjorn De Sutter, Bruno De Bus, Ludo Van Put, and Koen De Bosschere. System-wide compaction and specialization of the linux kernel. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 95–104, 2005.

[11] Dehao Chen, Tipp Moseley, and David Xinliang Li. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *CGO*, 2016.

[12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *Acm sigplan notices*, 47(4):37–48, 2012.

[13] Google. Propeller: Profile guided optimizing large scale llvm-based relinker. https://github.com/google/llvm-propeller, 2020.

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

[15] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd ISCA*, 2015.

[16] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. Ripple: Profile-guided instruction cache replacement for data center applications. In *Proceedings (to appear) of the 48th International Symposium on Computer Architecture (ISCA)*, ISCA 2021, June 2021.

[17] Rahman Lavaee, John Criswell, and Chen Ding. Codestitcher: inter-procedural basic block layout optimization. In *Proceedings of the 28th International Conference on Compiler Construction*, pages 65–75, 2019.

[18] Chi-Tai Lee, Jim-Min Lin, Zeng-Wei Hong, and Wei-Tsong Lee. An application-oriented linux kernel customization for embedded systems. *J. Inf. Sci. Eng.*, 20(6):1093–1107, 2004.

[19] David Xinliang Li, Raksit Ashok, and Robert Hundt. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 53–61, 2010.

[20] C-K Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. Ispike: a post-link optimizer for the intel/spl reg/itanium/spl reg/architecture. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 15–26. IEEE, 2004.

[21] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. Vespa: static profiling for binary optimization. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–28, 2021.

[22] Guilherme Ottoni. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165, 2018.

[23] Guilherme Ottoni and Bin Liu. Hhvm jump-start: Boosting both warmup and steady-state performance at scale. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 340–350. IEEE.

[24] Guilherme Ottoni and Bertrand Maher. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 233–244. IEEE, 2017.

[25] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2019.

[26] Maksim Panchenko, Rafael Auler, Laith Sakka, and Guilherme Ottoni. Lightning bolt: powerful, fast, and scalable binary optimization. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pages 119–130, 2021.

[27] Somu Perianayagam, HaiFeng He, Mohan Rajagopalan, Gregory Andrews, and Saumya Debray. Profile-guided specialization of an operating system kernel. In *Proc. Workshop on Binary Instrumentation and Applications*, 2006.

[28] Karl Pettis and Robert C Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 16–27, 1990.

[29] Mohan Rajagopalan, Somu Perinayagam, HaiFeng He, Gregory Andrews, and Saumya Debray. Binary rewriting of an operating system kernel. In *Proc. Workshop on Binary Instrumentation and Applications*, 2006.

[30] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 513–526, 2019.

[31] David W Wall. Predicting program behavior using real or estimated profiles. *ACM SIGPLAN Notices*, 26(6):59–70, 1991.

[32] Wikipedia contributors. Apachebench — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=ApacheBench&oldid=1061230570, 2021. [Online; accessed 3-April-2022].

[33] Wikipedia contributors. Leveldb — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=LevelDB&oldid=1060026512, 2021. [Online; accessed 3-April-2022].

[34] Wikipedia contributors. Perf (linux) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Perf_(Linux)&oldid=1035926020, 2021. [Online; accessed 3-April-2022].

[35] Wikipedia contributors. Apache http server — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Apache_HTTP_Server&oldid=1079941743, 2022. [Online; accessed 3-April-2022].

[36] Wikipedia contributors. Gcov — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Gcov&oldid=1066269648, 2022. [Online; accessed 3-April-2022].

[37] Wikipedia contributors. Memcached — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Memcached&oldid=1064747973, 2022. [Online; accessed 3-April-2022].

[38] Wikipedia contributors. Mysql — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=MySQL&oldid=1080373897, 2022. [Online; accessed 3-April-2022].

[39] Wikipedia contributors. Nginx — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Nginx&oldid=1077266637, 2022. [Online; accessed 3-April-2022].

[40] Wikipedia contributors. Postgresql — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=PostgreSQL&oldid=1076044937, 2022. [Online; accessed 3-April-2022].

[41] Wikipedia contributors. Redis — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Redis&oldid=1075238126, 2022. [Online; accessed 3-April-2022].

[42] Wikipedia contributors. Rocksdb — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=RocksDB&oldid=1077997787, 2022. [Online; accessed 3-April-2022].

[43] Pengfei Yuan, Yao Guo, and Xiangqun Chen. Experiences in profile-guided operating system kernel optimization. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, pages 4:1–4:6, New York, NY, USA, 2014. ACM.

[44] Pengfei Yuan, Yao Guo, and Xiangqun Chen. Rethinking compiler optimizations for the linux kernel: An explorative study. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, pages 2:1–2:7, New York, NY, USA, 2015. ACM.

[45] Pengfei Yuan, Yao Guo, Lu Zhang, Xiangqun Chen, and Hong Mei. Building application-specific operating systems: a profile-guided approach. *Science China Information Sciences*, 61(9):092102, Aug 2018.

[46] Mingzhou Zhou, Bo Wu, Yufei Ding, and Xipeng Shen. Profmig: A framework for flexible migration of program profiles across software versions. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12. IEEE, 2013.

[47] Yufeng Zhou, Xiaowan Dong, Alan L Cox, and Sandhya Dwarkadas. On the impact of instruction address translation overhead. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 106–116. IEEE, 2019.