Rethinking Compiler Optimizations for the Linux Kernel: An Explorative Study

Pengfei Yuan Yao Guo Xiangqun Chen

Key Laboratory of High-Confidence Software Technologies (Ministry of Education)
School of Electronics Engineering and Computer Science, Peking University
{yuanpf12, yaoguo, cherry}@sei.pku.edu.cn

Abstract

Performance of the operating system kernel is critical to many applications running on it. Although many efforts have been spent on improving Linux kernel performance, there is not enough attention on GCC, the compiler used to build Linux. As a result, the vanilla Linux kernel is typically compiled using the same `-O2` option as most user programs. This paper investigates how different configurations of GCC may affect the performance of the Linux kernel. We have compared a number of compiler variations from different aspects on the Linux kernel, including switching simple options, using different GCC versions, controlling specific optimizations, as well as performing profile-guided optimization. We present detailed analysis on the experimental results and discuss potential compiler optimizations to further improve kernel performance. As the current GCC is far from optimal for compiling the Linux kernel, a future compiler for the kernel should include specialized optimizations, while more advanced compiler optimizations should also be incorporated to improve kernel performance.

Keywords Operating system kernel, Linux, Compiler, GCC, Optimization

1. Introduction

Compiler has been the primary focus for optimizing the performance of an application. The performance of highly complex applications could be improved by as much as an order of magnitude in comparison to compiling with no optimizations. As a result, compiler becomes one of the most sophisticated software itself. For example, GCC contains over 14 million lines of code; LLVM/Clang contains over 4 million lines of code.

Modern compilers like GCC and Clang provide different levels of optimization such as `-O1`, `-O2`, `-O3`, with each subsequent level optimizing further on performance but probably incurring other unwanted side-effects. When embedded systems are considered, GCC and Clang also provide options like `-Os` to optimize more aggressively for code size in favor of small cache and storage.

On the other hand, the Linux kernel, which is also one of the most complicated software systems, contains more than 19 million lines of code. Although GCC has advanced optimization options (such as `-O3`) that can be applied to many applications to achieve performance speedup, the best option officially recommended to build the Linux kernel is `-O2`, targeting balanced performance and code size optimization, or `-Os`, used for embedded systems that prefer smaller code size. The compiler options used for building the Linux kernel are the same as used by regular applications.

It is perplexing to even think about that, with so many people contributing efforts on optimizing the Linux kernel from all angles, it still shares the same compiler (GCC) with so many other small and big applications. Although the Linux kernel achieves reasonable performance from the general-purpose compiler optimizations, should we build a dedicated compiler for the Linux kernel to further improve its performance? Should we at least implement some specific optimizations for the Linux kernel? In what extent can a dedicated compiler impact the kernel performance?

We attempt to answer the following questions in this paper by comparing the performance of different GCC variations:

- How do different levels of compiler optimizations affect Linux performance? (§4.1)
- How does size optimization affect performance, compared to speed optimizations? (§4.2)
- How do different GCC versions affect kernel performance? (§4.4)

APSys 2015, July 27–28, 2015, Tokyo, Japan.
Copyright © 2015 ACM 978-1-4503-3554-6/15/07…$15.00.
http://dx.doi.org/10.1145/2797022.2797030

- How does profile-guided optimization affect Linux kernel performance? (§4.3)
- How does switching specific optimization options affect Linux performance? (§4.5)
- What could we do in the compiler to improve the performance of the Linux kernel? (§5)

With detailed experimental results on a set of server applications, we show that different GCC configurations may affect the Linux kernel performance significantly. The key results are summarized as follows:

- Compared to the recommended option (`-O2`), more aggressive optimizations (`-O3`) may improve the application performance by as much as 15%, while aggressive size optimization (`-Os`) may degrade the application performance by as much as 14%.
- With feedback information collected from kernel instrumentation, the application performance can be improved by almost 8% on average (up to 13%).
- Different GCC versions have little to no influence on kernel performance, which results in less than 1% performance variation for all applications on average.
- Enabling or disabling specific optimizations (for example, function inlining and reordering) may speed up some applications, while slowing down others. This suggests that it is beneficial to perform application-specific compiler optimization for the kernel.

The rest of this paper is organized as follows. Section 2 introduces background information and related work. Section 3 presents the experimental setup and benchmarks used in our empirical study. We present detailed results and analysis in Section 4, then discuss implications, observations and potential future directions in Section 5. Finally, Section 6 concludes our study.

2. Background and Related Work

2.1 GCC Optimizations

Currently, the Linux kernel can only be compiled with GCC in production. Building the Linux kernel with other compilers such as LLVM/Clang is still experimental [15]. Therefore, we will focus on compiler optimizations in GCC [6].

The following GCC optimization levels are typically adopted in production:

- `-O2` enables nearly all mature compiler optimizations that do not involve significant space-speed tradeoffs.
- `-Os` enables all `-O2` optimizations that do not typically increase code size as well as further optimizations designed to reduce code size.
- `-O3` enables all `-O2` optimizations as well as optimizations that involve space-speed tradeoffs in favor of speed, such as aggressive function inlining and loop unrolling.

Besides these optimization levels, there are more advanced GCC optimizations including profile-guided optimization (PGO) and link-time optimization (LTO).

2.1.1 PGO

Profile-guided optimization (PGO) optimizes the target program via recompilation, based on tendency profiles observed in the past runs of the program. GCC introduced the PGO technique in 2001 [7].

PGO in GCC makes use of profile feedback in optimizations such as register allocation, code partitioning, loop unrolling, function inlining and basic block reordering [10].

Traditional PGO in GCC utilizes instrumentation to collect profile feedback. The new GCC 5.1 release[1] introduces Auto-FDO [3, 4], which is a sampling based approach.

2.1.2 LTO

Link-time optimization (LTO) is a type of compiler optimization performed to a program at link time. LTO in GCC was proposed in 2005 and merged in 2009 [5]. It is one of the most actively developed features of GCC [9].

Traditionally, GCC compiles C/C++ program on a file-by-file basis, which limits its ability to perform interprocedural analysis. LTO enables GCC to extend interprocedural optimizations such as constant prorogation, function inlining and reordering at the whole-program level [8].

2.2 Compiler Optimizations in Linux

The Linux kernel makes use of compiler optimizations conservatively. The kernel build system supports `-O2` and `-Os` directly. Previous attempt of adding `-O3` support was rejected by kernel developers [13].

Optimization level `-Os` used to be the default in Linux. Later `-O2` became the default due to performance penalties incurred by `-Os` size optimizations [16]. However, for some mobile and embedded devices with limited storage, the option `-Os` is still used in default kernel configurations.

PGO is widely adopted in user applications for performance improvement. In our previous work [17], we have shown that incorporating PGO to the Linux kernel can achieve significant performance improvement. We will perform similar experiments with the latest Linux and GCC versions in this paper.

Kernel developers have also tried to build Linux with LTO. However, their work has not been merged into the mainline kernel yet.

2.3 General OS Optimizations

The OS research community has been continuously attempting to improve kernel performance. Recent work has found scalability bottlenecks of the Linux kernel in a many-core environment [2]. The problems have been fixed in later

[1] Released on April 22, 2015

Table 1: Experimental environment.

Processor	Intel Core-i7 4770
Memory	32GB DDR3 1600MHz
Network	10Gbps LAN
Kernel	Linux 3.19.3
Kernel compiler	GCC 4.9.3 prerelease
Operating system	Debian sid amd64
File system	tmpfs

Table 2: Benchmarking applications.

Application	Version	Workload
Apache	2.4.10	ApacheBench
Nginx	1.6.2	ApacheBench
MySQL	5.6.21	DBT2
PostgreSQL	9.3.5	DBT2
Redis	2.8.17	redis-benchmark
Memcached	1.4.21	mc-benchmark

Linux versions. However, the page coloring mechanism, which is used to control last level cache partitioning in a multi-core environment [11, 18], is not adopted by the Linux kernel.

The principle of library OS has been adopted by systems like Arrakis [12] and IX [1] for performance improvement. Library OS for Linux has also been introduced recently [14].

Compared to the general OS optimizations and library OS implementations, compiler-based optimizations can be automated and further improve kernel performance.

3. Experimental Setup

This section presents the experimental environment and the benchmarks used in the experiments.

3.1 Experimental Environment

Table 1 lists our experimental environment on the test machine, where we run the benchmarking applications. We have another client machine for workload generation. The test machine and the client machine are connected via 10 Gigabit Ethernet.

We choose Debian sid distribution for better support of hardware and the kernel version used. We use *tmpfs* in our evaluation to avoid the uncertainty of disk I/O performance. The kernel and the compiler are the latest stable versions available[2].

3.2 Benchmarks

To measure the performance of the Linux kernel, we choose six server applications that are known to be system-intensive, namely Apache, Nginx, MySQL, PostgreSQL, Redis and Memcached.

Table 2 lists the application versions and workloads. All the applications are configured to run in multi-threaded/multi-process mode except Redis, which is a mostly single-threaded program.

Detailed parameter setups and evaluation methods can be found in our previous work [17]. The differences are listed as follows:

[2] As of March 30, 2015, one month before the submission deadline of this paper.

Table 3: Application performance on different kernels.

Application (metric)	`-Os`	`-O2`	`-O3`
Apache (req/s)	117,632	127,814	130,321
Nginx (req/s)	462,777	537,589	556,723
MySQL (tx/min)	67,377	70,661	71,008
PostgreSQL (tx/min)	75,115	79,763	79,536
Redis (op/s)	325,074	352,547	405,417
Memcached (op/s)	804,521	844,439	845,322

Table 4: Kernel code size.

Kernel	Code size (byte)
`-Os`	7,516,062
`-O2`	9,593,058
`-O3`	12,085,028

- The file size of web server response is increased to 1KB due to network bandwidth upgrade (1Gbps in previous work).
- The number of parallel benchmarking tool instances on the client machine is increased to 8.
- Memcached is configured with 8 servicing threads.

4. Results and Analysis

We first compare the three generally adopted optimization options of GCC on the Linux kernel, namely `-Os`, `-O2` and `-O3`.

Table 3 shows the overall performance results, which are measured by the number of operations each application is able to perform per second (throughput). The results will be analyzed next.

Table 4 shows the code size of the three kernel images, measured by the size of the `.text` section. Compared to the kernel compiled with the default `-O2` option, the kernel compiled with `-Os` is 22% smaller in code size, while the kernel compiled with `-O3` is 26% larger. The kernel image compiled with `-O3` is 60% larger than `-Os` in code size, which indicates significant size difference that should be considered in the context of embedded devices.

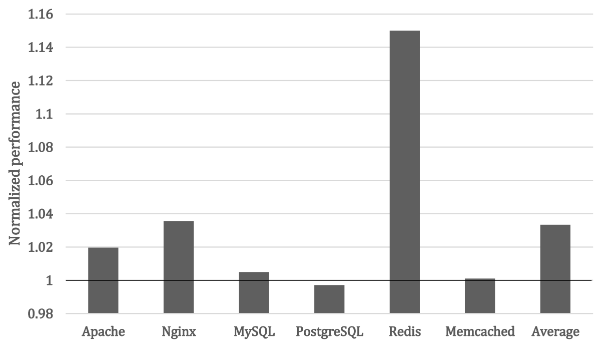

Figure 1: Performance results of aggressive speed optimization (`-O3`).

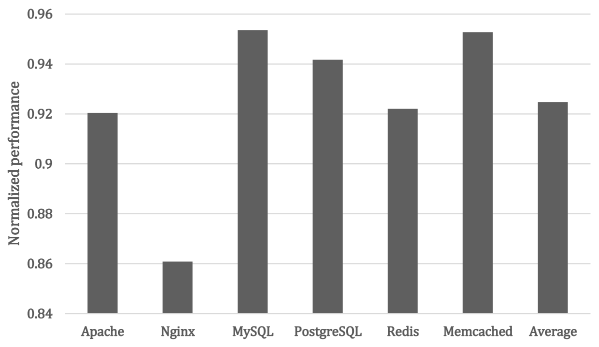

Figure 2: Performance results of size optimization (`-Os`).

4.1 Aggressive Speed Optimization

Figure 1 shows application performance[3] on the kernel compiled with `-O3`, which is normalized to performance of `-O2`. On average[4], the kernel with more aggressive speed optimization is 3.3% faster than the vanilla kernel.

Unlike the traditional perception that `-O3` would hurt performance of programs as large as the Linux kernel, our results show that application performance generally improves (5 out of 6 applications) when running on the kernel compiled with `-O3` and a modern microprocessor with sufficient cache[5].

Another commonly mentioned reason against adopting `-O3` in compiling Linux is that some aggressive optimizations might affect the correctness of the Linux kernel. However, we have not met any errors or abnormal phenomena during our experiments, which suggests that potential errors have already been fixed by Linux and GCC developers. Of course, more comprehensive testing is probably needed before `-O3` is widely used to build all Linux kernels, but it is worthwhile to investigate this issue further.

4.2 Size Optimization

Figure 2 shows application performance on the kernel compiled with `-Os`. Compared to the vanilla kernel compiled with `-O2`, performance is degraded for all six applications. On average, the size optimized kernel is 7.5% slower than the vanilla kernel.

Among the six applications, Nginx is most sensitive to size optimization. One possible reason is that Nginx has the largest network throughput, which is over 5Gbps in our experiments. The large network throughput puts higher pressure on the kernel, which may amplify the performance penalties incurred by `-Os`.

The inferior performance of `-Os`, combined with the increasing storage on mobile devices, may be the reason that `-O2` has been adopted more and more widely on mobile devices. For example, Google Nexus 5 and Samsung Galaxy S4 use `-O2` as the default option to build the Linux kernel for their Android systems.

4.3 Profile-Guided Optimization

We have presented experiences in profile-guided kernel optimization in our previous work [17]. Figure 3 shows the effectiveness of PGO on the newest GCC and Linux versions. On average, application performance improves by 7.6%. PGO generally outperforms GCC optimization levels `-Os`, `-O2` and `-O3` in kernel optimization.

Redis is the only exception, where `-O3` performs better than PGO. We notice that the performance numbers for Redis suffer higher variation in different runs compared to other applications. One possible reason for the large variation is that Redis is the only single-process and single-thread application among the six. When Redis is running, the CPU frequency is more unstable due to the Intel Turbo Boost technology, which is activated when the operating system requests the highest performance state of the processor. Nonetheless, the results of Redis in Figures 1 and 3 do not affect the conclusion that PGO is generally better than `-O3`.

Compared to our previous results [17], the average performance improvement is consistent. The performance of Nginx improves much more because we upgrade to 10Gbps network, which avoids bandwidth saturation.

4.4 Impact of Different GCC versions

We also conduct experiments to find how the evolution of GCC influences Linux kernel performance. We choose GCC versions 4.6 to 4.9 and use the default `-O2` option to compile the Linux kernel.

Figure 4 shows the application performance comparison normalized to the latest version 4.9. For individual applications like Nginx, performance varies as much as 3% on kernels compiled with different GCC versions (probably due to the same reason as explained in §4.2). But on average, there is no obvious pattern with kernel performance as GCC

[3] Performance values in all figures are normalized to the vanilla kernel compiled with GCC 4.9 and `-O2`.

[4] All average values are calculated as geometric means.

[5] Core i7-4770 has 256KB L1 cache, 1MB L2 cache and 8MB L3 cache.

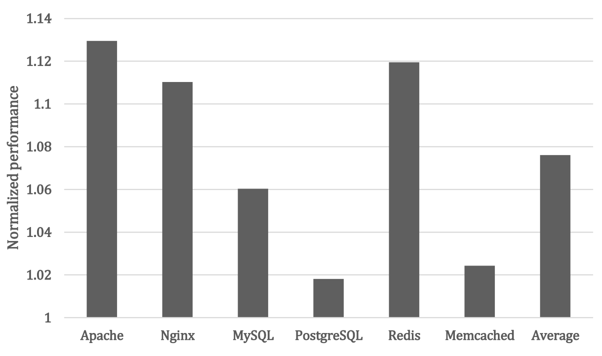

Figure 3: Performance results of profile-guided optimization.

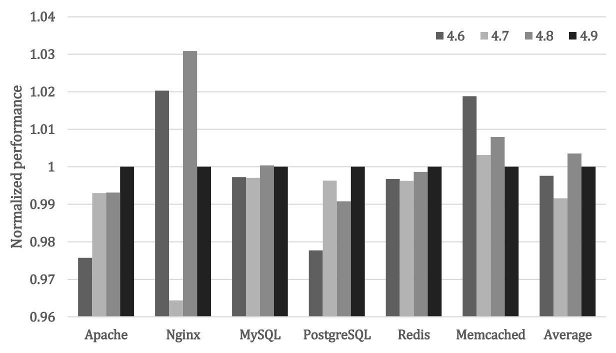

Figure 4: Performance comparison of different GCC versions.

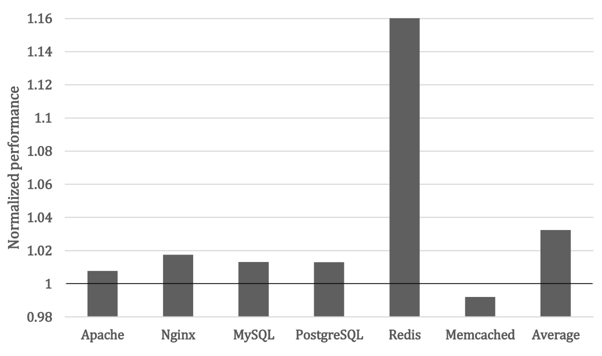

Figure 5: Performance results while enabling aggressive function inlining.

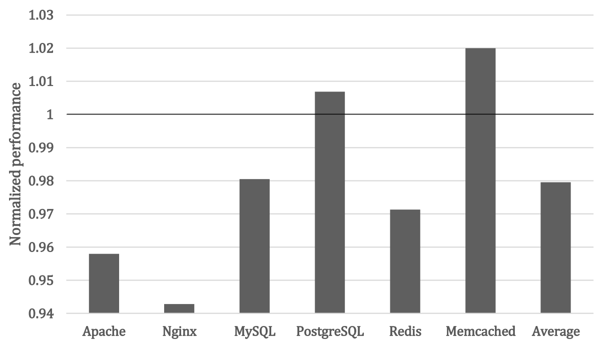

Figure 6: Performance results while disabling code reordering.

version evolves: the performance difference is only 1%, which falls within the error range.

4.5 Controlling Specific Optimizations

To explore the effectiveness of detailed compiler optimizations such as function inlining and code reordering, we try enabling/disabling them and comparing them to the vanilla kernel.

Figure 5 shows application performance on the kernel compiled with `-O2` and aggressive function inlining[6] enabled. On average, the kernel is 3.2% faster than the vanilla kernel. Compared with the results in §4.1, we find that aggressive function inlining contributes most to the performance gain of `-O3` over `-O2`.

Figure 6 shows application performance on the kernel compiled with `-O2` and reordering[7] disabled. On average, the kernel is 2% slower than the vanilla kernel.

When specific applications are considered, we can see that enabling aggressive inlining improves performance for most applications except Memcached, while disabling code reordering degrades the performance of four applications but improves for two (PostgreSQL and Memcached). This indicates that detailed compiler optimizations can be tuned in an application-specific way to maximize kernel performance on specific workloads.

[6] Option `-finline-functions`, enabled by `-O3`.

[7] Including function reordering, basic block reordering, and code partitioning.

5. Discussions

5.1 Observations

We have presented detailed analysis on performance comparisons of various GCC configurations applied to the Linux kernel. Based on the results presented above, we have the following observations:

- As the more aggressive `-O3` optimization may improve the application performance by as much as 15%, we suggest that `-O3` should be seriously considered to replace `-O2` when building customized Linux kernel, especially for devices that do not have strict storage limitations.
- As aggressive size optimization (`-Os`) may degrade the application performance by as much as 14%, it should not be used unless the device is severely limited by storage (for example, embedded devices, but probably not for mobile devices such as smartphones).

- With feedback information collected from kernel instrumentation, profile-guided optimization (PGO) can improve application performance by almost 8% on average. For expert users who are sensitive to kernel performance, for example, on a cloud server running heavy workload, PGO can provide kernel images that maximize application performance.
- Without feedback information, the commonly adopted `-O2` option is far from optimal for compiling the kernel. It suggests that we should perform thorough investigation to find out the set of optimizations that maximize kernel performance.
- Different GCC versions have small to none influence on kernel performance, which shows less than 1% performance difference on average. It indicates that current GCC evolution is not targeting programs like the Linux kernel. More GCC and Linux developers should be involved to help improve compiler optimizations for the Linux kernel.

5.2 Future Directions

Kernel performance is critical to many applications running on it. As we have demonstrated that different optimizations on the Linux kernel may affect application performance significantly, more efforts should be devoted to exploring how different compiler optimizations or different combinations of existing optimizations may affect kernel performance. In particular, we believe that the following future directions are worthwhile to explore:

- *Dedicated compiler optimizations for Linux.*

  The Linux kernel is a complicated and special program that requires dedicated compilation techniques to maximize its performance. Our results in §4.4 show that different GCC versions have small to none influence on kernel performance, which indicates that more efforts should be focused on building a GCC branch that is dedicated to better optimizations for kernel performance.

  For example, the results in §4.5 show that code reordering, a general compiler optimization enabled by `-O2`, sometimes leads to performance degradation. We believe similar issues can be found via comprehensive inspection and solving the issues will create a better compiler for Linux.

  Not only should we consider new compiler optimizations for the kernel, we should also find out the combination of existing optimizations that is best suited for the kernel according to our observations. Dedicated optimizations for Linux may also differ from common GCC in default parameter values. The parameters are used by optimizers to determine various optimization thresholds. Considering the differences between the kernel and user programs, it makes sense to search for an optimal set of compiler parameters for kernel compilation.

- *Application-specific optimizations.*

  While 99% of the users would be satisfied with distribution kernels optimized with a dedicated compiler, expert users may further improve kernel performance with application-specific kernel optimization.

  In our previous work [17], we have already shown that applying PGO to the Linux kernel can achieve significant application-specific benefits.

  On the other hand, as we have shown in §4.5, enabling or disabling specific optimizations (for example, function inlining and reordering) may speed up some applications, while slowing down others. This suggests that we could investigate how to achieve further application-specific benefits with more fine-grained control on different optimizations.

- *Introducing more advanced optimizations to Linux.*

  For instance, LTO is another actively developed features of GCC [9], which has not been accepted into the mainline Linux kernel yet. If LTO support is merged into the mainline Linux kernel, it may provide more opportunities to further optimize kernel performance at link-time. Moreover, LTO can be combined with PGO to achieve extra application-specific benefits in the kernel. Similar approaches have been already adopted by user applications like Firefox.

6. Concluding Remarks

We have explored the potential performance impacts of different compiler optimizations on the Linux kernel. Experimental results show that the current compiler (GCC) is a long way from producing the best-performed Linux kernel. To make compiler optimizations perform better for the kernel, not only should we improve existing general compiler optimizations, but also investigate advanced compiler optimizations specifically targeting the Linux kernel.

As opposed to kernel hacking and library OS construction, compiler-based optimization on the kernel can be automated, such that it may provide consistent benefits to all different applications. While general compiler optimizations for the kernel is potentially beneficial, application-specific compiler optimizations for the kernel may achieve greater performance improvement for specific applications and usage scenarios.

Acknowledgments

We would like to thank the anonymous reviewers and especially our shepherd, Wenguang Chen, for their insightful comments and feedback. This work was supported in part by the High-Tech Research and Development Program of China under Grant No.2015AA01A203, and the National Natural Science Foundation of China under Grant No.61421091, 61103026.

References

[1] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, Oct. 2014. ISBN 978-1-931971-16-4.

[2] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, 2010.

[3] 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. *Computers, IEEE Transactions on*, 62(2):376–389, Feb 2013. ISSN 0018-9340. .

[4] Free Software Foundation, Inc. Automatic feedback directed optimizer. `https://gcc.gnu.org/wiki/AutoFDO`, .

[5] Free Software Foundation, Inc. Link time optimization. `https://gcc.gnu.org/wiki/LinkTimeOptimization`, .

[6] Free Software Foundation, Inc. Options that control optimization. `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`, .

[7] Free Software Foundation, Inc. Infrastructure for profile driven optimizations. `https://gcc.gnu.org/news/profiledriven.html`, .

[8] T. Glek and J. Hubicka. Optimizing real world applications with GCC link time optimization. *arXiv preprint arXiv:1010.2196*, 2010.

[9] J. Hubicka. Linktime optimization in GCC, part 1 - brief history. `http://hubicka.blogspot.ca/2014/04/linktime-optimization-in-gcc-1-brief.html`.

[10] J. Hubicka. Profile driven optimisations in GCC. In *GCC Summit Proceedings*, pages 107–124, 2005.

[11] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4): 338–359, Nov. 1992. ISSN 0734-2071. .

[12] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, Oct. 2014. ISBN 978-1-931971-16-4.

[13] J. Ringle. Add option to build with -O3. `https://lkml.org/lkml/2014/3/4/1020`.

[14] H. Tazaki. An introduction of library operating system for Linux. `https://lkml.org/lkml/2015/3/24/254`.

[15] The Linux Foundation. LLVMLinux project overview. `http://llvm.linuxfoundation.org/index.php`.

[16] L. Torvalds. Give up on pushing `CC_OPTIMIZE_FOR_SIZE`. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=281dc5c5ec0f`.

[17] P. Yuan, Y. Guo, and X. Chen. Experiences in profile-guided operating system kernel optimization. In *Proceedings of the 5th Asia-Pacific Workshop on Systems*, APSys '14, 2014.

[18] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 89–102, 2009. ISBN 978-1-60558-482-9. .