Experiences in Profile-Guided Operating System Kernel Optimization

Pengfei Yuan, Yao Guo, and 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

The operating system kernel plays an important role in the whole computer system. In this paper, we try to improve application performance by optimizing the underlying operating system kernel. The technique we take advantage of is profile-guided optimization, which is a compiler optimization technique commonly used in user applications. We implement the technique in the Linux kernel and present some preliminary results evaluated on six popular server applications: Apache, nginx, MySQL, PostgreSQL, Redis and memcached. We also discuss some opinions about kernel performance improvement and some problems with applying this technique to the kernel.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Performance

General Terms

Experimentation, Performance

Keywords

Operating system kernel, Profile-guided optimization, Server application

1. INTRODUCTION

Traditionally, the operating system kernel is optimized to meet the performance requirements of different types of applications. Tradeoffs are made everywhere to guarantee that there is no performance bottleneck. However, such a general-purpose kernel is often suboptimal in a specific application scenario.

In this paper, we propose a general method to optimize the operating system kernel for each specific application. In our method, we take advantage of a compiler optimization technique called profile-guided optimization (PGO). PGO is commonly used in user applications for performance improvement. Well-known projects such as Firefox [16] and PHP [14] have already adopted this technique for a few years. But as far as we know, the technique has not been applied to the operating system kernel yet. We implement the method in the Linux kernel and show that it is an effective method for improving the performance of various popular server applications.

We decide to conduct experiments on server applications mainly for three reasons. First, many server applications run on dedicated servers. For example, many web servers running Apache do not run other applications. This feature makes server applications a perfect target of application-specific kernel optimization. Second, server applications are key components of the Internet infrastructure. Improving the performance of server applications has significant potential benefit to billions of Internet users. Third, many server applications are system-intensive, thus making kernel optimizations more accelerative.

Performance is a very hot topic in the operating system kernel community. We have seen numerous discussions on performance regressions and related bug fixes in the Linux kernel mailing list. Recent research on kernel performance mainly focuses upon problems faced in the multicore era, such as kernel locking and synchronization mechanisms [15], cache partitioning and management [22], cache-friendly process coscheduling [18], DDR-friendly memory allocation [20]. Compared with our method, previous work mainly focuses on specific performance problems, instead of specific applications.

The main contribution of our method is making application-specific kernel optimizations possible. Our method enables applications to run on matching operating system kernels that can maximize performance. The optimization process does not involve any source code modification either to the target applications or to the kernel.

The rest of this paper is organized as follows. Sec-

APSys '14 June 25–26 2014, Beijing, China

http://dx.doi.org/10.1145/2637166.2637227

tion 2 provides some background knowledge about our method. Readers that are familiar with the PGO technique may skip this section. Section 3 describes how we implement PGO in the Linux kernel. In Section 4, we evaluate the method with six real-world server applications. In Section 5, we discuss advantages of using PGO in the kernel and some problems with applying the PGO technique to the kernel. Section 6 concludes this paper.

2. BACKGROUND

Profile-guided optimization has been well studied in the compiler design field [17]. The technique is supported by modern compilers including GCC [9], Microsoft Visual C++ Compiler [11], Intel C++ Compiler [10], etc.

PGO consists of the following three phases.

The first phase is instrumentation. In this phase, the compiler instruments the target application during compilation in order to collect profile information that will be used for later optimizations. The profile information consists of control flow traces, value and address profiles, etc. For example, the `-fprofile-generate` option which is used in this phase in GCC enables the following options: `-fprofile-arcs`, `-fprofile-values`, `-fvpt`. The three options are responsible for profiling control flow graphs, profiling values, and doing value profile transformations respectively.

The second phase is collection. In this phase, the instrumented target application is executed for several times to collect profile information. The execution process should be as close to real-world scenarios as possible. After execution, the collected profile information is stored in some data files. In GCC, the data file format is the same as the format used in gcov, the test coverage program of GCC.

The third phase is optimization. In this phase, the compiler uses the profile information collected in the second phase to optimize the target application. The profile information helps the compiler make better decisions at code layout, function inlining, loop unrolling, etc. In GCC, option `-fprofile-use` enables optimizations based on profile information.

3. IMPLEMENTATION

We implement the PGO technique in the Linux kernel according to the three previously explained phases. Figure 1 demonstrates our implementation.

Gcov is a test coverage program in GCC and it shares the same instrumentation infrastructure with PGO. Since gcov has already been supported by the Linux kernel [19], we only need to make a few modifications to further support PGO in the kernel. The gcov option `--coverage` in GCC only enables the `-fprofile-arcs` instrumentation, so what we do is adding support for

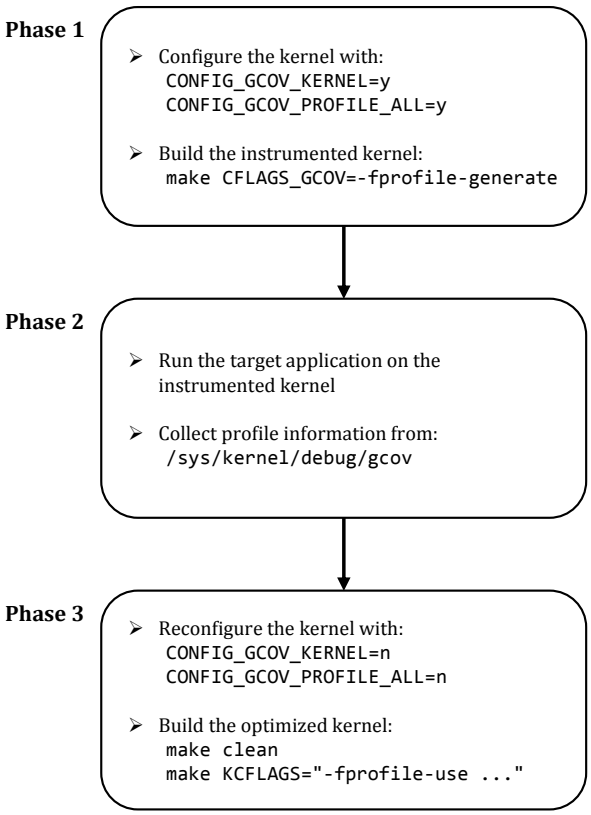

Figure 1: Kernel PGO Process

options `-fprofile-values` and `-fvpt` into the kernel. We add the following profilers and their corresponding merging functions that are used in the instrumentation phase to the kernel gcov subsystem: indirect call profiler, ior profiler, average profiler, one value profiler, interval profiler, and pow2 profiler. Moreover, instrumenting certain kernel source files will break the kernel, resulting in boot failure. Therefore, we disable instrumentation options for these source files in the kernel makefile.

In the first phase, we configure the kernel with options `CONFIG_GCOV_KERNEL` and `CONFIG_GCOV_PROFILE_ALL` enabled and set kernel makefile variable `CFLAGS_GCOV` to "-fprofile-generate". Then we build the instrumented kernel as normal.

In the second phase of our method, we run applications with the instrumented kernel to collect profile information. Afterwards, a shell script is used to collect profile data files from DebugFS.

Another problem with kernel PGO is that the collected profile information lacks counter summary and histogram, which are required by GCC during optimization. The summary and histogram serve as statistics at the whole program level. For user applications, they are calculated by libgcov at the program exit time. But for the kernel, such a mechanism is not feasible. Instead, we

Table 1: Kernel PGO Evaluation Environment

Processor	Intel Core-i7 4770
Memory	32GB DDR3 1600MHz
Network	1Gbps LAN
Kernel	Linux 3.13.5
Kernel compiler	GCC 4.8.3 prerelease
Operating system	Debian sid amd64
File system	tmpfs

write a tool to help calculate the counter summary and histogram for the collected kernel profile information.

In the third phase of our method, we disable gcov-related kernel options previously set on, and rebuild the kernel with kernel makefile variable `KCFLAGS` set as "-fprofile-use -fprofile-correction -Wno-error=coverage-mismatch -fprofile-dir=/path/to/profile". The kernel image we get in this phase is profile-guided optimized.

4. EVALUATION

We evaluate our method with six server applications, namely Apache, nginx, MySQL, PostgreSQL, Redis and memcached. We do not choose popular PHP or Java-based server applications like WordPress or Confluence because their performance mostly depends on the PHP intepreter or the Java virtual machine, as well as the supporting web servers and database servers.

Our evaluation environment is listed in Table 1. The six server applications run on the test machine. We have another client machine running testing tools. The test machine and the client machine are connected via gigabit Ethernet. We choose the Debian sid distribution for better support of the hardware and the version of kernel we use. We use tmpfs in our evaluation to avoid the uncertainty of disk I/O performance.

In our evaluation, we first run the server applications on the vanilla kernel and get their performance values via benchmarking tools. Then we do the PGO process described in previous sections and get six optimized kernels, for the six server applications respectively. In the profile collection phase, the specific server application runs on the instrumented kernel and the benchmarking tool is executed to produce profile information. Afterwards, we run the server applications on their corresponding optimized kernels and get their performance values again via benchmarking tools.

Figure 2 shows the evaluation result, which is the performance improvements of the six server applications running on the optimized kernels over the vanilla kernel. For Apache, MySQL, PostgreSQL, Redis and memcached, the performance improves by 1.71–10.27%. But for nginx, the performance decreases by 0.59%. Details about the server applications and performance results will be explained subsequently.

4.1 Apache

Apache is the most popular web server with a market share up to 38% of all websites [8]. It was investigated in previous work [15] and proved to be system-intensive. In our evaluation, we use the apache2 package distributed with Debian sid, which is Apache Version 2.4.7. We configure Apache to use the MPM event module [2] and increase the "MaxRequestWorkers" option, which determines the maximum number of connections that will be processed simultaneously, for later testing.

On the client, we use ab [1], the Apache HTTP server benchmarking tool, to get the performance of Apache running on the test machine. The size of the requested file is 4 bytes to avoid saturating the network bandwidth. We run 4 ab instances simultaneously, each one at concurrency level 100, performing 100 HTTP requests at a time. Each ab instance sends one million requests and outputs the average number of requests processed in one second. We add the four numbers to get the throughput of Apache.

On the vanilla kernel, Apache can handle 63889 requests per second. On the profile-guided optimized kernel, Apache can handle 70449 requests per second. The over 10% performance improvement shows that the Apache 2.4 release is still very system-intensive. Even though the MPM event module is introduced for asynchronous processing, the fundamental model of Apache is still based on threads. By optimizing kernel hot paths with profile information, we achieve impressive performance improvement on Apache.

4.2 Nginx

Nginx is another popular web server, which is remarkable for its high performance and low resource consumption. It is the third most widely used web server with a market share of over 15% among all websites [8]. In our evaluation, we use the nginx package distributed with Debian sid, which is Version 1.4.7. We increase the worker_processes option to 8 and the worker_connections option to 1000.

On the client, we also use ab for benchmarking, with the same settings as we do on Apache.

On the vanilla kernel, nginx can handle 97065 requests per second. On the profile-guided optimized kernel, the throughput of nginx decreases by 0.59% to 96494 requests per second. Due to the asynchronous event-driven approach adopted by nginx, it is less system-intensive than Apache. One possible reason that the PGO technique causes nginx performance degradation is that the critical kernel path after PGO becomes less code cache friendly to nginx. As we know, improper decisions at code layout, function inlining or loop unrolling may result in disturbance to cache locality. The performance degradation in this single case might be related to compiler misoptimization. But

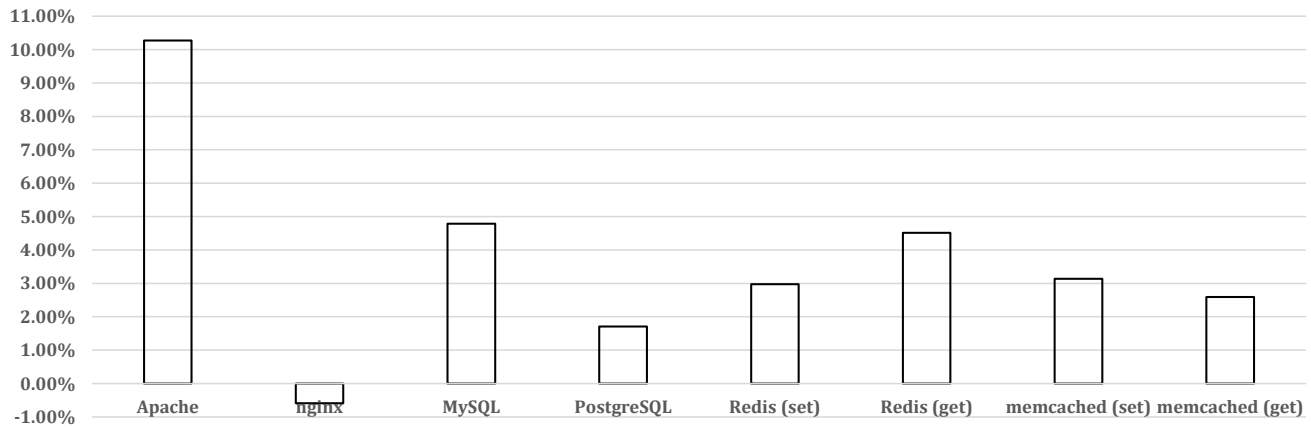

Figure 2: Server Application Performance Improvements with Kernel PGO

we still need to investigate the root cause to make our method more reliable.

4.3 MySQL

MySQL is the most popular open-source relational database management system [5]. It is widely used in small websites for data management. In our evaluation, we use MySQL Community Server 5.6.16, which is the latest generally available release. We configure MySQL to use the default InnoDB storage engine [7].

The benchmarking tool we use to test the performance of MySQL is dbt2 [4], which is an open-source implementation of the TPC-C benchmark specification. It is an online transaction processing performance test. The dbt2 benchmark suite consists of a dataset generator, a database client, and a benchmarking driver. We first generate a dataset of 8 warehouses and import it to MySQL. Then we run the database client program and MySQL on the test machine. On the client machine, we run the dbt2 driver with 100 threads per warehouse. Each thread simulates a terminal connected to the server. The dbt2 performance metric is NOTPM, the number of new order transactions processed in one minute.

On the vanilla kernel, the throughput of MySQL is 69163 NOTPM. On the profile-guided optimized kernel, the throughput increases to 72474 NOTPM. The 4.79% improvement shows that MySQL is also system-intensive, although data query processing is CPU-intensive.

4.4 PostgreSQL

PostgreSQL is the second most popular open-source relational database management system [5]. Compared with MySQL, it provides better fundamental database features and more database capabilities [3]. In our evaluation, we use PostgreSQL 9.3.3. We increase the checkpoint_segments option to 64 because of checkpoint warnings in the PostgreSQL log file [13].

The benchmarking tool we use is also dbt2. We use the same settings as we do on MySQL.

On the vanilla kernel, the throughput of PostgreSQL is 71852 NOTPM. On the profile-guided optimized kernel, the throughput increases to 73080 NOTPM. Previous research also investigated PostgreSQL [15]. As the Linux kernel and PostgreSQL evolves, the performance of PostgreSQL is less restricted by the kernel. So our method achieves less performance improvement on PostgreSQL than on MySQL.

4.5 Redis

Redis is the most popular key-value store [5], widely available on many cloud platforms. Redis is a mostly single-threaded program and makes use of event-driven techniques to achieve concurrency [12]. In our evaluation, we use Redis Version 2.8.7.

On the client, we use the redis-benchmark tool, which is released together with Redis, to test the performance. We test the throughput of set and get operations. In the test, we run 4 redis-benchmark instances simultaneously, with 100 parallel connections per instance.

On the vanilla kernel, the throughput of Redis is 326363 set operations per second and 344239 get operations per second. On the profile-guided optimized kernel, the throughput increases to 336074 set operations per second and 359769 get operations per second. Compared with nginx, which also uses event-driven techniques, the throughput of Redis improves after kernel PGO. We believe the reason is that Redis is more system-intensive due to much more network connections processed per second than nginx.

4.6 Memcached

Memcached is the second most popular key-value store [5], which is also widely available on various cloud platforms. Compared with Redis, memcached is multi-threaded as well as event-driven. But memcached does not support data persistence. In our evaluation, we use memcached 1.4.17.

Although memcached supports multi-threading very well, we configure memcached to use only one thread because we fail to saturate the server if memcached serves with multiple threads. The same reason applies to our Redis test: we do not run multiple Redis instances on the test machine. Both Redis and memcached are highly efficient [6].

When testing memcached, we use the mc-benchmark tool, which is ported from the redis-benchmark tool to adapt to the memcached protocol. Benchmarking settings are the same as we do on Redis.

On the vanilla kernel, the throughput of memcached is 366089 set operations per second and 514800 get operations per second. On the profile-guided optimized kernel, the throughput increases to 377569 set operations per second and 528159 get operations per second. Since memcached does not support data persistence, it is faster than Redis. According to previous research, memcached itself is good enough, hardware is the bottleneck [15].

5. DISCUSSION

The method we propose in this paper to improve application performance by optimizing the underlying operating system kernel via the PGO technique is a general approach. We can use this method to adapt the kernel to any specific application or scenario. Although the method is not specialized, the profile-guided optimized kernel we get from this method is specific to the application scenario it is optimized for. We should not expect that the kernel performs well on other scenarios according to the principle of the PGO technique.

In this paper, we mainly focus on optimizing the kernel for server applications because many server applications are known to be system-intensive. Since servers play a key role in the Internet, a little improvement on server performance may contribute a lot to the productivity of the whole Internet.

Unlike server applications, desktop applications and mobile applications are mostly interactive. The performance problems in these applications often reside in their models of human-computer interaction. For mobile applications, energy efficiency is much more important. As far as we can see, there is little we can do in the kernel to help deal with the energy bugs or other issues discovered in mobile applications.

Recent research concerning operating system kernel performance is concentrated on specific features of the kernel, such as locking [15], memory management [20, 22], process scheduling [18]. These optimizations are general to applications. Looking into the Linux kernel mailing list, we find that posts about improving kernel performance also have the same characteristic. Our method pursues the opposite: it targets the whole kernel but is specific to each application scenario.

Linus Torvalds was in favor of checking compiler hints in the kernel source code instead of shipping profile information with kernel releases [21]. Although runtime profile feedback can help developers optimize the code, the optimization requires a lot of tradeoffs because source code changes must be general. In our method, there is no need to ship profile information with releases since different users may have different customizations on their own. We only provide a way to make customized optimizations possible. As long as the compiler is smart enough, our method should be reliable.

Currently, the compiler only takes into account the kernel profile information. Although the application behavior is to some extent reflected in the kernel profile information, the compiler still knows nothing about the interaction between the kernel and the application.

In our evaluation, we make use of benchmarking tools to test the performance of the six server applications because it is difficult to figure out and measure real-world workloads. We think that the results from ab, redis-benchmark and mc-benchmark are meaningful because the functionalities of Apache, nginx, Redis and memcached are onefold. The dbt2 benchmark suite implements the TPC-C benchmark specification, which should also be representative.

6. CONCLUSION

In this paper, we propose a new method for improving application performance by optimizing the underlying operating system kernel. Unlike previous work, our method targets the whole kernel but is specific to each application scenario. We implement the method in the Linux kernel with the support of profile-guided optimization from GCC.

In evaluation, we run benchmarks on six popular server applications. For Apache, MySQL, PostgreSQL, Redis and memcached, performance improvements range from 1.71% up to 10.27%. But for nginx, performance decreases by 0.59% using our method. The culprit leading to performance degradation will be investigated to make our method more robust and reliable.

As future work, we will look into kernel PGO under real-world workloads such as small servers running Linux, Apache, MySQL and PHP together, servers hosting virtual private servers, clusters used for big data analysis.

ACKNOWLEDGMENT

This work is supported by the National Basic Research Program of China (973) under Grant No. 2011CB302604, the High-Tech Research and Development Program of China under Grant No. 2013AA01A605, the National Natural Science Foundation of China under Grant No. 61103026, 61121063, U1201252.

References

[1] ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/current/programs/ab.html.

[2] Apache MPM event. http://httpd.apache.org/docs/current/mod/event.html.

[3] Comparison of relational database management systems. http://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems. Accessed: 2014-03-20.

[4] Database test 2. http://osdldbt.sourceforge.net.

[5] DB-engines ranking. http://db-engines.com/en/ranking. Accessed: 2014-03-20.

[6] How fast is Redis? http://redis.io/topics/benchmarks.

[7] The InnoDB storage engine. http://dev.mysql.com/doc/refman/5.6/en/innodb-storage-engine.html.

[8] March 2014 web server survey. http://news.netcraft.com/archives/2014/03/03/march-2014-web-server-survey.html.

[9] Options that control optimization. http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[10] Profile guided optimization (PGO) options. http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/cpp-win/GUID-87339BCB-7547-4633-93EB-E11EF55906AA.htm.

[11] Profile-guided optimizations. http://msdn.microsoft.com/en-us/library/e7k32f4k.aspx.

[12] Redis latency problems troubleshooting. http://redis.io/topics/latency.

[13] Write ahead log. http://www.postgresql.org/docs/9.3/static/runtime-config-wal.html.

[14] Asthana, A. Speed up Windows PHP performance using profile guided optimization (PGO). http://blogs.msdn.com/b/vcblog/archive/2013/05/06/speeding-up-php-performance-for-your-application-using-profile-guided-optimization-pgo.aspx.

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

[16] Glek, T. Squeezing every last bit of performance out of the Linux toolchain. https://blog.mozilla.org/tglek/2010/04/12/squeezing-every-last-bit-of-performance-out-of-the-linux-toolchain/.

[17] Gupta, R., Mehofer, E., and Zhang, Y. Profile guided code optimizations. In *The Compiler Design Handbook*, Y. N. Srikant and P. Shankar, Eds. CRC Press, 2002.

[18] Jaleel, A., Najaf-abadi, H. H., Subramaniam, S., Steely, S. C., and Emer, J. CRUISE: Cache replacement and utility-aware scheduling. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 249–260.

[19] Oberparleiter, P., Nemeth, M., and Hrbata, F. Using gcov with the Linux kernel. https://www.kernel.org/doc/Documentation/gcov.txt.

[20] Park, H., Baek, S., Choi, J., Lee, D., and Noh, S. H. Regularities considered harmful: Forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 181–192.

[21] Torvalds, L. https://lkml.org/lkml/2006/1/5/350.

[22] Zhang, X., Dwarkadas, S., and Shen, K. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), EuroSys '09, ACM, pp. 89–102.