SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2018, Vol. 61 092102:1–092102:17
https://doi.org/10.1007/s11432-017-9418-9

Building application-specific operating systems: a profile-guided approach

Pengfei YUAN[1,2], Yao GUO[1,2*], Lu ZHANG[1,2], Xiangqun CHEN[1,2] & Hong MEI[1,2]

[1]*Key Laboratory of High-Confidence Software Technologies* (*Ministry of Education*), *Peking University, Beijing* 100871, *China;*
[2]*School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871, *China*

Received 21 November 2017/Revised 31 January 2018/Accepted 13 March 2018/Published online 13 August 2018

Abstract Although operating system optimization has been studied extensively, previous work mainly focuses on solving performance problems. In the cloud era, many servers only run a single application, making it desirable to provide an application-specific operating system (ASOS) that is most suitable for the application. In contrast to existing approaches that build ASOS by manual redesign and reimplementation, this paper presents Tarax, a compiler-based approach to constructing an ASOS for each application. With profile collected from executing the target application on an instrumented Linux kernel, Tarax recompiles the kernel while applying profile-guided optimizations (PGOs). Although GCC has already implemented the optimization process that can be applied to user applications, it does not work on the Linux kernel directly. We modify the Linux kernel and GCC to support kernel instrumentation and profile collection. We also modify GCC to reduce the size of optimized kernel images. We conduct experiments on six popular server applications: Apache, Nginx, MySQL, PostgreSQL, Redis and Memcached. Experimental results show that application performance improves by 8.8% on average (up to 16%) on the ASOS. We also perform detailed analysis to reveal how the resulting ASOS improves performance, and discuss future directions in ASOS construction.

Keywords operating system, Linux kernel, performance, GCC, profile-guided optimization

Citation Yuan P F, Guo Y, Zhang L, et al. Building application-specific operating systems: a profile-guided approach. Sci China Inf Sci, 2018, 61(9): 092102, https://doi.org/10.1007/s11432-017-9418-9

1 Introduction

As the foundation of a computer system, the operating system (OS) is critical to the performance of all applications running on it, especially system-intensive applications that invoke kernel features extensively [1–3]. As a result, OS optimization has been studied extensively, which includes a vast of research work trying to optimize every aspect of an OS. Most of these efforts tend to solve specific performance problems in a general way that is suitable for various types of applications. In order to provide a "one for all" OS, tradeoffs are made to guarantee that the performance is consistently good for all applications. However, such a general-purpose OS is often suboptimal for a specific application.

Many computers run only a very small set of applications or even a single application. For example, the computer behind an ATM machine typically runs only a single application. Many web servers run nothing but the Apache server. In the cloud era, as there are more and more servers, it becomes more

* Corresponding author (email: yaoguo@pku.edu.cn)

prevalent to run a single application on each dedicated server or virtual machine, instead of running many applications on one server as in the past.

As a result, we argue that application-specific operating system (ASOS) should be built to provide an optimal running environment for each application. ASOS was first proposed by Anderson [4] more than 20 years ago. ASOS differs from ordinary OS in that it focuses on the performance of a specific application, instead of the overall performance for all possible applications. Figure 1 shows the general idea of ASOS, where each application runs on a dedicated OS kernel. Recent examples of ASOS are mostly based on the principle of library OS introduced by exokernel [5]. For example, unikernel [6] focuses on optimizing for the cloud and is adopted by LightVM [7] for its low memory footprint. Arrakis [8] and IX [9] are proposed for datacenter workloads.

Although application-specific library OS can achieve significant size or speed improvement, it typically requires an entire reimplementation of the OS kernel, and even the applications running on it. It typically requires first manually identifying the performance bottleneck, and then redesigning and reimplementing the whole system. Although it is realistic to build an ASOS for some particular application or a set of applications, it will be almost impossible to build an ASOS for every application running on it.

In this paper, we propose Tarax, a compiler-based approach that takes advantage of profile-guided optimizations (PGOs) to construct an ASOS for each application. Compared to previous approaches, Tarax does not need to modify OS source code when building an ASOS. Therefore we can build an ASOS in significantly fewer efforts. While most existing work could only develop an ASOS for one type of applications, Tarax can build a true ASOS that is specific to each application.

Specifically, Tarax extends PGO in GCC to perform application-specific optimizations on the Linux kernel. PGO makes use of feedback collected from runtime profiling to guide the compiler optimization of a program. By employing runtime feedback, the compiler can provide more accurate optimizations than without the feedback. PGO is commonly used for user applications to improve performance. Well-known projects such as Firefox and PHP have already adopted this technique for a few years. GCC itself can also be built with PGO and shows about 7% speedup.

We have demonstrated the feasibility of applying PGO to the Linux kernel to achieve speedups in a workshop paper [10]. This paper extends previous work, and proposes a more general solution in Tarax to build ASOS with the help of PGO. We also perform comprehensive analysis on the experimental results to provide insights on how profile information helps improve OS performance.

Since PGO in GCC cannot be directly applied on the Linux kernel, we investigate the reasons why PGO does not work and make corresponding modifications to the Linux kernel and GCC to support kernel instrumentation and feedback collection. At the same time, to make GCC more suitable for building ASOS, we also modify GCC to produce smaller Linux kernel binaries when feedback is available.

Overall, we have modified 1017 lines of code in the Linux kernel and 148 lines of code in GCC and submitted some of our modifications to the open source community, some of which has already been accepted. We also automate the Tarax procedures so that an ASOS can be constructed with little user intervention, and no need to modify the Linux kernel code as well.

We make the following main contributions in this paper.

• We propose Tarax, a compiler-based approach to constructing ASOS, which achieves the "one kernel for one application" goal. Tarax is highly automated with a dedicated toolchain, such that users do not need to make any manual modifications to the kernel or to the target applications.

• We conduct experiments on six popular server applications to demonstrate the effectiveness of Tarax. The results show that the performance of these applications improves by up to 16% when running on the Linux kernel optimized for each application. We also perform extensive analysis to reveal insights into the performance optimization opportunities arising from Tarax.

• To the best of our knowledge, Tarax is the first comprehensive work that successfully enables PGOs on modern OSs such as the Linux kernel. Besides the optimizations presented in this paper, we believe more opportunities could arise for performing more advanced optimizations on the Linux kernel to achieve extra application-specific benefits.

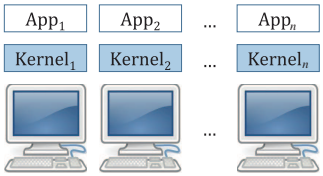

Figure 1 (Color online) The general idea of ASOS.

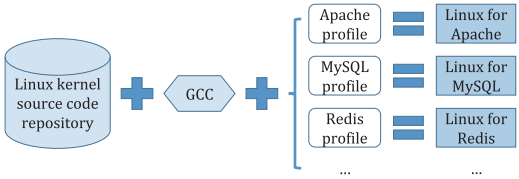

Figure 2 (Color online) Tarax overview.

2 Tarax overview and challenges

Our ultimate goal is to build ASOS automatically. To achieve this goal, Tarax is designed as a compiler-based approach that takes advantage of PGOs. Figure 2 presents an overview of Tarax. We target to build application-specific Linux kernels for popular server applications such as Apache, MySQL and Redis. With the profile feedback from running individual applications, we rely on the compiler (i.e., GCC) to perform better optimizations on Linux kernel source code, and create kernel images optimized for the corresponding application.

2.1 PGO in GCC

PGO has been well studied in the compiler community [11]. The compiler attempts to mitigate the cost of a program's generality by using feedback information such as control flow graph (CFG) and expression value profiles, which are collected in one or more previous program runs. The compiler then focuses its optimization efforts on the frequently executed portions of the program by understanding the run-time tendencies within these portions. PGO has been applied to large open source projects such as Firefox and Chrome.

A typical PGO process consists of the following phases.

- **Instrumentation.** 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.
- **Profile collection.** The instrumented target application is executed to collect profile information. The execution process should reflect real-world runtime scenarios.
- **Optimization.** The compiler uses the profile information collected in the previous phase to optimize the target application. The profile information helps the compiler make better decisions on branch prediction, basic block reordering, function inlining, and loop unrolling.

In GCC, PGO instrumentation can be enabled by turning on an option (-fprofile-generate). After instrumentation, one needs to run the application and collect profile data. Finally, the application is recompiled with an option (-fprofile-use) to turn on the compiler optimizations using the collected profile data: branch optimizations, basic block reordering, function inlining, register allocation, and code partitioning. Recent GCC versions also support sampling-based AutoFDO (automatic feedback directed optimizer), which does not require instrumentation. We will discuss it in Subsection 6.3.

2.2 Challenges

Applying PGO to user applications such as Firefox can be performed by simply enabling the related options in GCC. However, applying it directly to the Linux kernel faces several technical challenges.

(1) How to enable kernel instrumentation. To collect profile feedback from individual applications, the kernel should be instrumented. However, unlike user applications, some features in the Linux kernel conflict with compiler instrumentation, which may result in Linux failing to boot.

(2) How to collect profile information. In order to enable profile collection, the compiler has some auxiliary libraries that the instrumented program should link against. But the Linux kernel is self-contained and does not allow linking against external libraries.

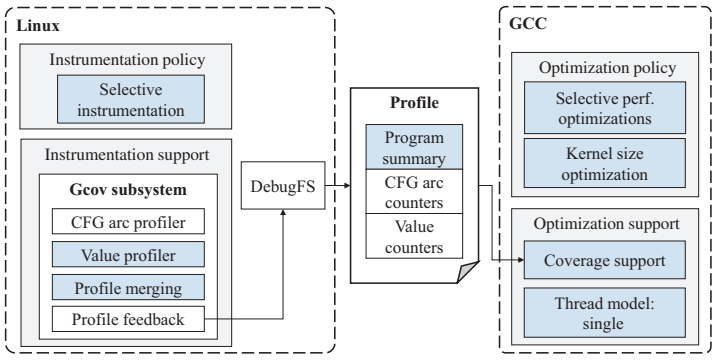

Figure 3 (Color online) System architecture of Tarax.

(3) When to collect profile information. For an instrumented program, the profile feedback is collected on exit. However, the runtime behavior of the kernel is different since it never really exits. We need to collect profile feedback on-the-fly and carry on post processing.

(4) How to choose correct optimizations. If profile feedback is available, the compiler can perform more aggressive optimizations that are otherwise disabled by default. Some optimizations will cause wrong code generation on certain kernel functions, resulting in build failure.

3 Tarax design and implementation

3.1 Tarax design

Figure 3 presents the architecture of Tarax, where the shaded components involve our modifications and implementation. To solve the challenges listed in Subsection 2.2, we make modifications to Linux, GCC, as well as the profile data files collected from Linux.

We first need to design an approach to enable PGO instrumentation on the Linux kernel. Fortunately, the gcov subsystem of Linux shares the same instrumentation infrastructure and the same data format with the PGO implementation in GCC. The main difference is that gcov includes instrumentation capabilities on only CFG profiling. It does not support value profiling, which is required by PGO.

In order to enable full kernel instrumentation, Tarax extends the gcov subsystem of Linux. We also modify GCC to adapt to kernel instrumentation. To collect profile information from the kernel, we make use of the existing debug filesystem (DebugFS) interface. To choose better optimizations for the kernel, we modify the optimization option handling logic in GCC to support better size-speed tradeoff.

3.2 Kernel instrumentation

Linux kernel instrumentation in Tarax is based on the gcov subsystem. It already supports the -fprofile-arcs instrumentation, which is used in coverage testing. In order to support full PGO instrumentation, we make modifications to the gcov subsystem to support value profiling, and also make modifications to handle various other issues.

• **Value profiling.** To support profiling on values via instrumentation, we add the following profilers that are used in the instrumentation phase to the kernel gcov subsystem: indirect call profiler, ior profiler, average profiler, one value profiler, interval profiler, pow2 profiler, time profiler, and indirect call topn profiler. These profilers work together with the CFG arcs profiler, which is already supported in the Linux kernel. Besides these profilers, we also need to add profile merging functions, which are included in the auxiliary libraries of GCC. An instrumented program should link against these libraries, but the Linux kernel building process does not allow linking against external libraries. So we port these functions to Linux to keep the kernel code self-contained.

• **Disabling TLS.** The PGO implementation in GCC is designed for user applications and makes use of thread-local storage (TLS) in value profiling. The TLS mechanism, which uses an extra segment

register, requires kernel support. However, it is not available in the kernel itself. The kernel's per-CPU allocation, which is similar to TLS, uses a different segment register and is not available before kernel initialization. So we disable this feature in kernel instrumentation. Specifically, we add the --disable-threads and --disable-tls options when configuring and building GCC.

• **Selective instrumentation.** After the above modifications, the instrumented kernel may still not be able to boot because some functions, if instrumented, interfere with the self-patching mechanism in the Linux kernel. To solve this problem, we further modify Linux source file "arch/x86/kernel/paravirt.c", using the function-specific option pragma optimize provided by GCC to disable value profiling instrumentation on incompatible functions including _paravirt_nop, _paravirt_ident_32 and _paravirt_ident_64. Moreover, the profiler functions themselves cannot be instrumented. So we disable instrumentation on the whole gcov subsystem in the makefile.

3.3 Profile collection

With the implementations described in Subsection 3.2, we have incorporated instrumentation capabilities to the Linux kernel. However, certain statistics such as counter summary and histogram, which are required by GCC during optimization, are calculated by an auxiliary library when the instrumented program exits. Although this is normal for user applications, such statistics for the kernel will be missing since the kernel does not actually exit after it boots up.

To solve this problem, our implementations include:

• We write a utility program to help calculate the counter summary and histogram after collecting profile data from the DebugFS interface;

• Instead of collecting profile data at program exit, we collect the profile data of the kernel on-the-fly;

• As the kernel never exits, we specify the start and the end of the profile collection process based on the start and the end of the target application running on it.

3.4 Application-specific optimizations

With profile feedback available, the compiler can now perform application-specific optimizations on the kernel. We make the following efforts to fix optimization errors and improve its performance.

• **Fixing optimization problems.** Some optimizations are incompatible with kernel code, resulting in assembler errors at kernel build time. For example, code reordering is incompatible with some kernel functions that have complex inline assembly, such as the function _static_cpu_has_safe in "arch/x86/include/asm/cpufeature.h". To solve this issue, we disable optimization options on a per-source-file basis in the kernel makefile. The advanced compiler optimizations enabled by PGO may also cause kernel boot failure due to misoptimization. For example, the __schedule function in "kernel/sched/core.c" may cause kernel panic if compiled with aggressive optimizations, We use the function specific option pragma optimize to disable optimizations on a per function basis.

• **Selective size optimization.** Ideally, reducing kernel code size helps reduce instruction cache misses and improve kernel performance. However, if we turn on aggressive size optimization in GCC (-Os), it severely degrades kernel performance (Subsection 5.3). In order to make better size-speed tradeoffs, we modify GCC to enable aggressive size optimization only when the profile data shows that the whole translation unit is never executed[1)]. Specifically, we change the cost model of GCC x86 backend from ix86_tune_cost to ix86_size_cost to optimize for code size, set the code alignment constraint to one byte to remove function-level code padding, and turn off optimizations that may increase code size on the never-executed code. In this way, we are able to reduce the instruction cache footprint of the kernel as much as possible without sacrificing performance.

1) Translation unit is the input to a compiler from which an object file is generated.

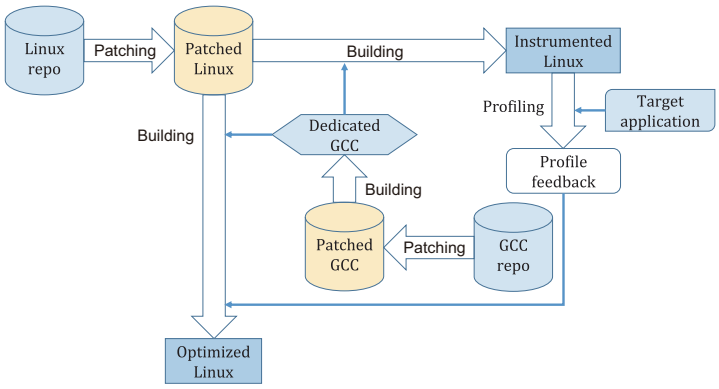

Figure 4 (Color online) The building and optimization workflow in Tarax.

3.5 Workflow and automation

We automate Tarax with a dedicated toolchain, as shown in Figure 4. In the figure, the procedures in italic are automated using shell scripts. Only booting up the instrumented kernel and running the target application require user intervention. The workflow includes the following steps.

(1) Preparation. We patch the Linux kernel and GCC with the above modifications. Then we build a dedicated GCC binary for kernel optimization.

(2) Instrumentation. We configure the kernel with CONFIG_GCOV_KERNEL and CONFIG_GCOV_PROFILE_ALL options enabled and set kernel makefile variable CFLAGS_GCOV to -fprofile-generate. Then we build the instrumented kernel with the CC variable set as our dedicated GCC.

(3) Profiling. We boot the instrumented kernel and run the target application to collect kernel profile information from DebugFS. This step requires user involvement to run different applications.

(4) Optimization. We disable gcov-related kernel options previously set on, and rebuild the kernel with makefile variable KCFLAGS set as "-fprofile-correction -Wno-error=coverage-mismatch -fprofile-use -fprofile-dir=/path/to/profile".

3.6 Implementation

Our implementation of Tarax is summarized as follows.

- **Linux.** We have modified eight source files (1017 lines of code), including five in the gcov subsystem, two in the x86-specific code, and "kernel/sched/core.c". The modified gcov subsystem contains auxiliary libraries ported to support instrumentation and profiling (420 lines of code).
- **GCC.** We have modified three source files (148 lines of code), including two in the coverage support code and one in the compiler driver.
- **Utilities.** We have implemented 2 utilities (395 lines of C++ code) for profile data file processing.
- **Scripts.** We have implemented six shell scripts to automate the building process.

We have submitted two patches for the gcov subsystem to Linux and one of them has been accepted to the mainline kernel[2)]. We have also submitted a patch for GCC that improves optimization option handling, which is in the revision process to meet the GCC acceptance criteria. We also plan to release the automated toolchain to the public to encourage further research and improvement in this direction.

4 Experimental setup

4.1 Environment

Our experimental environment includes a test machine running the target applications and a client machine running benchmarking tools. Table 1 lists the experimental environment. The test machine and

2) Git commit ID is a992bf83.

Table 1 Experimental environment

Type	Parameters
Processor	Intel Core i7-4770
Memory	32 GB DDR3 1600 MHz
Network	10 Gbps LAN
Kernel	Linux 4.1.2
Kernel compiler	GCC 5.1.1
Operating system	Debian sid amd64
File system	tmpfs

Table 2 Application versions

Application name	Version
Apache	2.4.23
Nginx	1.10.2
MySQL	5.6.25
PostgreSQL	9.3.9
Redis	3.0.2
Memcached	1.4.21

the client machine are connected via 10 Gigabit Ethernet. We choose Debian sid as the target Linux distribution for better hardware and software support. We also use tmpfs to avoid the uncertainty of disk I/O performance.

4.2 Benchmarking methodology

We conduct experiments on six server applications that are known to be system-intensive, namely Apache, Nginx, MySQL, PostgreSQL, Redis and Memcached. Table 2 lists the application versions used in our experiments. We first run the six server applications on the vanilla kernel and measure their performance via benchmarking tools. Then we carry out the optimization process described in Subsection 3.5 and get six optimized kernels for the six server applications, respectively. Finally, we run the target applications on their corresponding optimized kernels and measure their performance again. The characteristics of the six applications and their benchmarking configurations are as follows.

• **Apache**, the most popular web server, has been investigated in previous work [12] and proved to be system-intensive. We configure the web server to serve both static and dynamic requests. The response size ranges from 256 to 2048 bytes. We do not choose even larger response sizes to avoid network bandwidth saturation. On the client, we generate randomized requests, with the ratio of static:dynamic requests evenly distributed at 1:1. The tool we use is ab, the Apache HTTP server benchmarking tool.

• **Nginx** is another popular web server. We use the same benchmarking settings as Apache.

• **MySQL** is the most popular open-source relational database system, widely used in small websites for data management. The benchmarking tool we use is dbt2, an open-source implementation of the TPC-C benchmark specification. It is an online transaction processing performance test. The dbt2 performance metric is NOTPM, the number of new order transactions processed in one minute.

• **PostgreSQL** is another popular database system. The benchmarking tool we use is also dbt2.

• **Redis** is the most popular key-value store, widely available on many cloud platforms. It is a mostly single-threaded program and makes use of event-driven techniques to achieve concurrency. The benchmarking tool we use is memtier. We configure it to generate randomized workloads with the ratio of get:set operations evenly distributed at 1:1.

• **Memcached** is another popular key-value store. Compared with Redis, Memcached is multi-threaded and event-driven, but does not support data persistence. The benchmarking tool we use is also memtier.

Table 3 Application performance on the vanilla kernel and the kernels optimized by Tarax

Application (m)	Performance				Improvement (%)
	Vanilla		Tarax		
	Mean	Stdev (%)	Mean	Stdev (%)	
Apache (requests/s)	61843	0.16	69186	0.71	11.9
Nginx (requests/s)	255397	0.25	298443	0.30	16.9
MySQL (trans/min)	70499	0.25	74489	0.43	5.7
PostgreSQL (trans/min)	80943	0.59	83194	0.50	2.8
Redis (operations/s)	367807	0.45	396407	0.23	7.8
Memcached (operations/s)	427715	0.80	464129	0.23	8.5
Average (geomean)					8.8

5 Evaluation

In our evaluation, we first perform experiments to compare the performance and code sizes of the Tarax-optimized kernels and the vanilla kernel. We then perform dynamic profiling on the kernels to collect detailed statistics on instruction cache misses and branches. Finally, we switch on specific GCC optimizations with and without profile feedback, respectively, to collect performance numbers. We use these experiments to answer the following questions:

- What are the performance benefits of kernels optimized by Tarax in comparison to the vanilla kernel? Are the optimized kernels application-specific? (Subsection 5.1)
- Is Tarax general enough to adapt to different workloads, different hardware architectures and different Linux versions? (Subsection 5.2)
- How does Tarax affect kernel code sizes? (Subsection 5.3)
- Where do the performance benefits come from? (Subsection 5.4)
- Does the profile feedback really help the compiler to perform better optimizations? (Subsection 5.5)

5.1 Performance comparison

5.1.1 *Overall performance*

We first compare the overall performance of the optimized kernels and the vanilla kernel. We run each benchmark five times and calculate the arithmetic means, which are shown in Table 3. The results show that Tarax achieves positive performance improvement consistently for all six applications, with improvement of more than 16% for Nginx. On average, application performance is improved by 8.8% when running on the corresponding optimized kernels[3)].

We also present the standard deviations of performance numbers from different test runs in Table 3. Although the standard deviations for specific applications may increase or decrease, they are all relatively low, which indicates that the performance improvement are stable throughout the experiments.

The performance improvement numbers are in the same range as PGO on user applications. According to our experience, the JavaScript performance of Firefox improves by about 5% using PGO. A recent result on SPEC CPU2006 shows 4.5% improvement after applying PGO.

5.1.2 *Cross evaluation*

To investigate whether the optimized Linux kernels are really application-specific, we also run each application on kernels optimized for other applications. Figure 5 shows the result matrix, where all numbers shown are normalized to the application performance on its own optimized kernel.

If the optimized kernel is best-suited for the target application, the numbers on the diagonal should be the highest; all other numbers should be below 1 since they are running on kernels optimized for other applications. We can see that most of the results follow this pattern, with all applications except Memcached achieving the best performance on their own optimized kernel.

3) All averages are calculated as geometric means throughout this paper, unless otherwise noted.

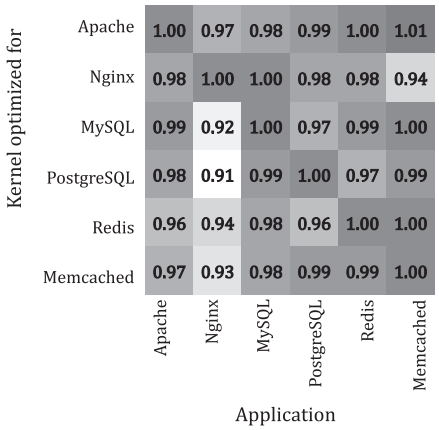

Figure 5 Performance speedup on different optimized kernels. For each application, the numbers are normalized to the performance when it runs on the kernel optimized for itself.

For example, when we run Nginx on all six kernels, the performance of it running on other kernels ranges from 91% to 97% of the performance on its own kernel. On the other hand, although the performance of Memcached is generally good while running on other kernels, the performance of other applications running on the kernel optimized for Memcached could be as low as 93% of their best performance. One possible reason is that the kernel hot path of Memcached is a subset of other applications.

The results show that we have created truly application-specific Linux kernels for each application. Running an application on an arbitrary (albeit) optimized kernel could degrade its performance by 9%.

Note that Apache and Nginx exhibit different behaviors on these kernels even though they are both web server applications, which indicates that it is sometimes difficult to build a uniformly good kernel even for a set of applications with similar functionalities.

5.2 Sensitivity analysis

5.2.1 *Sensitivity on workloads*

In our experiments, the workloads are generated randomly such that different workloads are used during profiling and testing. However, they still follow the same distribution. If the workload distribution of an application changes, will it affect the performance improvement achieved by Tarax?

Figure 6 shows how workload changes may influence application performance of Nginx and Memcached. The kernel for Nginx is optimized with the ratio of static:dynamic requests set as 1:1. When the ratio changes, performance improvement of Nginx ranges from 15% to 21%. The kernel for Memcached is also optimized with the ratio of get:set operations set at 1:1. When the ratio changes, performance improvement of Memcached ranges from 8% to almost 10%. Workload changes influence the performance of Apache and Redis similarly. Since dbt2 generates randomized database workloads, we do not perform workload sensitivity analysis on MySQL and PostgreSQL.

The results show that the optimized kernels are robust against application workload changes.

5.2.2 *Sensitivity on hardware platforms*

How does Tarax perform on different hardware platforms? Figure 7 presents application performance improvement of the optimized kernels over the vanilla kernel on both Intel and AMD microprocessors. The AMD microprocessor we use in the comparison is FX-8350.

We can see that performance improvements of PostgreSQL and Redis are higher on AMD, but the average performance improvement is higher on Intel. On average, Tarax achieves 7.6% performance improvement on AMD, which shows that it is still effective on a different hardware platform.

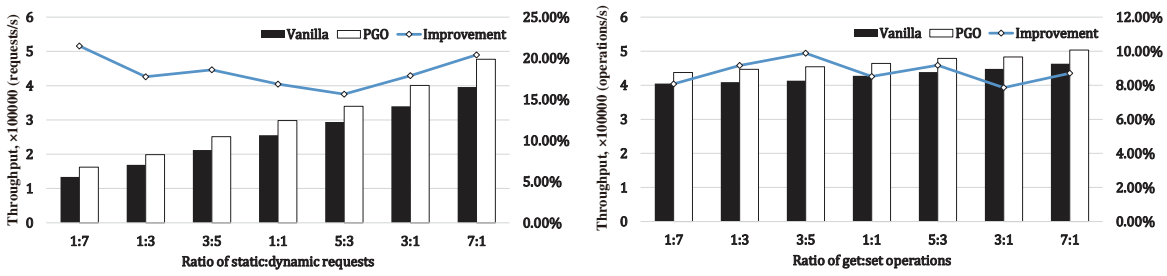

Figure 6 (Color online) Performance comparison with different workload. (a) Nginx; (b) memcached.

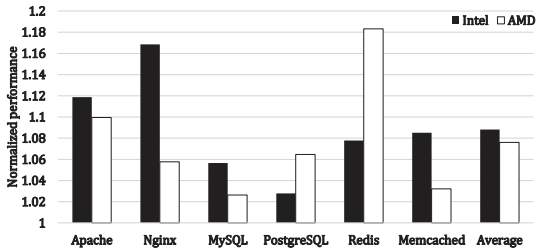

Figure 7 Performance speedup on different hardware platforms (normalized to the corresponding vanilla kernel).

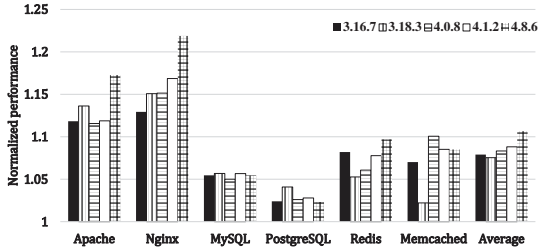

Figure 8 Performance on different Linux versions (normalized to the corresponding vanilla kernel).

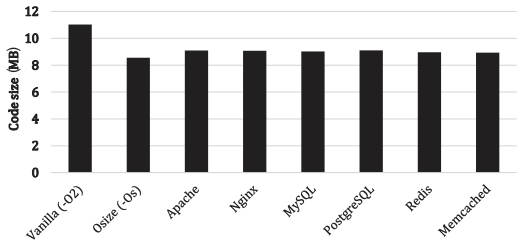

Figure 9 Comparison of kernel code sizes.

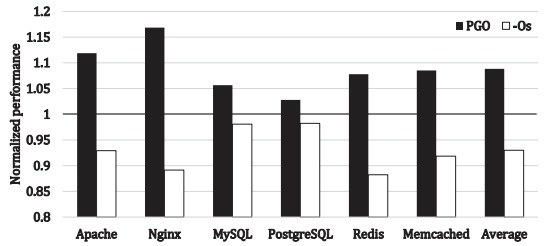

Figure 10 Performance comparison: Tarax vs. kernel compiled with -Os (normalized to the vanilla kernel).

5.2.3 *Sensitivity on Linux versions*

How do different OS versions affect the optimization effectiveness? Figure 8 shows application performance on five different Linux versions: 3.16.7, 3.18.3, 4.0.8, 4.1.2, and 4.8.6.

Because there are many changes between these Linux versions, we can see that the performance numbers vary significantly on some applications. For example, the performance improvement on Memcached ranges from 2% to 10%. However, the average performance improvement is steady and consistent, ranging from 7.5% to 10.7%, which indicates that Tarax is effective on different kernel versions.

On Linux 4.8.6, Tarax achieves 10.7% average performance improvement, which is the highest among the five versions. It shows that Tarax is still effective along the Linux version evolution.

5.3 Kernel code size comparison

A smaller kernel is beneficial as it could reduce instruction cache misses (which will be shown later). Figure 9 compares the code sizes of the optimized kernels, with the vanilla kernel and the kernel compiled with aggressive size optimization -Os. We measure the .text section size of the kernel image.

We can see that the optimized kernels are significantly smaller than the vanilla kernel compiled with the default -O2 option; but they are a little larger than the kernel compiled with -Os.

We also compare the performance of the Tarax-optimized kernels and the kernel compiled with -Os, which is shown in Figure 10. We can see that the kernel compiled with -Os is much slower than the

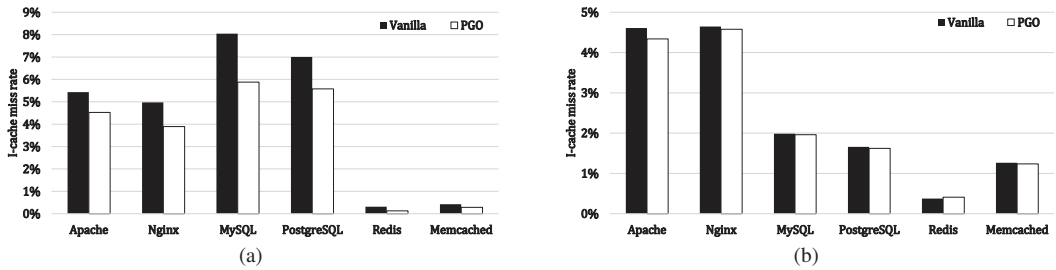

Figure 11 I-cache miss rates from dynamic profiling. (a) Kernel; (b) application.

Tarax-optimized kernels; they are even slower than the vanilla kernel (-O2). This shows that aggressive size optimization could actually degrade the kernel performance; however, the compiler can make better decisions on size-speed tradeoffs with profile feedback.

5.4 Dynamic profiling analysis

In order to explain how the application-specific kernels are optimized, we perform dynamic profiling with perf [13] to collect performance related statistics. The sampling range is 10 s during application execution. As perf supports profiling the kernel and user mode separately, we can calculate instruction cache (I-cache) miss rates for the kernel and the application, respectively. We use the number of executed instructions to approximate the number of I-cache accesses. We then calculate misprediction and taken rates of branch instructions in kernel mode.

5.4.1 *Instruction cache statistics*

Figure 11 presents the statistics on I-cache for both kernels and applications. We can see that the I-cache miss rates for applications are reduced slightly in five of the six benchmarks (I-cache miss rate of Redis increases slightly from 0.37% to 0.41%). However, the I-cache miss rates for the kernels are significantly reduced; the biggest reduction is 2.17 percentage points for MySQL. For Memcached, the I-cache miss rate for the kernel is reduced by more than 58% (from 0.31% to 0.13%). The result shows that Tarax improves the I-cache metrics, which is a major contribution to kernel performance speedup.

5.4.2 *Branch optimizations*

Figure 12 shows the profiling results of branch instructions in the kernel mode. We expect that the compiler should make better decisions on branch prediction and code layout with profile feedback.

Figure 12(a) shows that compiler branch prediction does not help reduce branch misprediction rate in the kernel, which is expected (explained in Subsection 2.1).

Instead of reducing branch mispredictions, the compiler exploits branch probabilities to reduce the number of taken conditional branches. Figure 12(b) shows that the number of taken branches have been reduced by over 50% for some kernels. Figure 12(c) shows over 1/3 reduction in branch-taken rates for all optimized kernels. Please note that branch-taken rates on the vanilla kernel are all higher than 50% because the compiler cannot reverse the condition and invert conditional branches used in loops without profile feedback. Reducing taken branches favors I-cache locality as well. It is another contribution to kernel performance speedup.

5.4.3 *Function inlining*

We also make use of clock cycle based sampling to see which kernel functions are live at runtime. Taking Apache as an example, Figure 13 shows the top 10 live kernel functions when it runs on the vanilla kernel and the optimized kernel, respectively.

We can see that many of the top 10 functions in the two kernels are different. Taking the most frequently executed function thread_group_cputime as an example, it is invoked by function thread_group_cputime

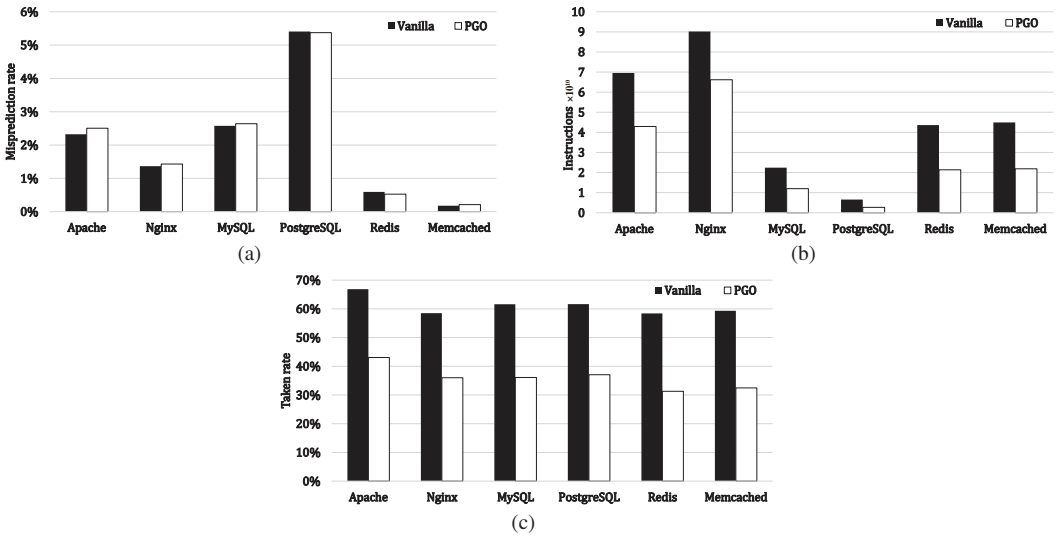

Figure 12 Branch statistics from dynamic profiling. (a) Branch misprediction rate; (b) taken branch instructions; (c) branch taken rate.

%	Function name
10.01	thread_group_cputime
3.82	rwsem_down_write_failed
1.58	__switch_to
1.56	__schedule
1.56	_raw_spin_lock
1.30	_raw_spin_lock_irqsave
1.25	task_sched_runtime
1.18	copy_user_enhanced_fast_string
0.96	__fget
0.85	tcp_ack
0.85	page_fault

(a)

%	Function name
11.43	thread_group_cputime_adjusted
4.77	rwsem_down_write_failed
2.18	enqueue_task_fair
1.83	dequeue_task_fair
1.80	__switch_to
1.74	task_sched_runtime
1.71	_raw_spin_lock
1.59	_raw_spin_lock_irqsave
1.47	copy_user_enhanced_fast_string
1.35	context_switch
1.29	select_task_rq_fair

(b)

Figure 13 Top 10 live kernel functions at runtime of Apache. (a) Vanilla kernel; (b) optimized kernel.

```
void thread_group_cputime_adjusted(
        struct task_struct *p,
        cputime_t *ut,
        cputime_t *st)
{
    struct task_cputime cputime;
    thread_group_cputime (p, &cputime);
    cputime_adjust(&cputime,
        &p->signal->prev_cputime,
        ut, st);
}
```

Figure 14 Code of kernel function thread_group_cputime_adjusted.

_adjusted, whose source code is shown in Figure 14. In the optimized kernel, thread_group_cputime does not appear in the sampling results, as it is inlined during optimization.

Kernel developers do not manually inline the function because it is invoked in multiple places. In the vanilla kernel, the compiler does not inline the function because it acts conservatively without runtime profile. Inlining the function in all places it is invoked may bloat the kernel and hurt performance (which will be shown later). However, with profile information available, GCC is able to perform smarter inlining on places where the callee has been invoked most frequently.

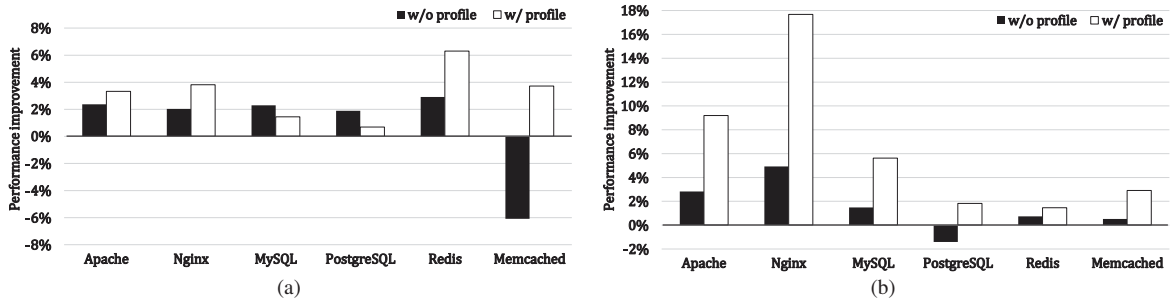

Figure 15 Effects of profile feedback on different GCC optimizations, results shown are performance improvements of enabling the respective option over disabling it, with or without profile feedback. (a) Function inlining; (b) basic block and function reordering.

5.5 Profile feedback analysis

We manually control some GCC optimization options and compare their performance to see how runtime profile influences these optimizations. Specifically, we control option -finline-functions for function inlining and option -freorder-blocks-and-partition for code reordering.

Figure 15 shows the results of profile feedback analysis, which are performance improvements of enabling the options over disabling them, with or without profile feedback.

From Figure 15(a), we can see that the performance improvements of five applications are higher when performing function inlining with runtime profile. The profile feedback of PostgreSQL does not help the compiler perform better function inlining on the kernel. For Memcached, aggressive function inlining without profile feedback severely degrades performance by over 6%.

Figure 15(b) shows that the performance improvements of all six applications are higher when performing code reordering with runtime profile. For PostgreSQL, code reordering without profile feedback degrades performance by 1.4%.

The results show that runtime profile is beneficial for these two optimizations in most cases. Without profile feedback, aggressive optimizations could degrade performance.

6 Discussions

6.1 Application scenario

Tarax can be used as a general approach to improving application performance by optimizing the underlying kernel. We can use Tarax to adapt the kernel to any specific application or scenario. For example, when an application evolves to a new version, we can easily rebuild a new kernel to offer the best possible performance for the new version. This would be almost impossible with manual redevelopment.

6.2 Kernel stability guarantees

In Subsection 3.4, we have disabled aggressive optimizations in PGO on particular files that caused trouble during optimization. We have not found any other cases that may affect kernel stability during runtime after applying Tarax. However, to further ensure that PGO does not introduce any instability to the kernel, we can disable all aggressive optimizations enabled by PGO. In this way, Tarax can apply the same optimizations as in -O2, with only the static branch predictions of GCC replaced by profile feedback. Thus we can guarantee that the Tarax-optimized kernels are as stable as the kernel optimized with -O2.

Disabling aggressive optimizations may impact the speedup. However, based on our preliminary experiments on Linux 4.8.6, the speedup is still about 6% on average, which shows that profile feedback helps improve kernel performance even without these extra optimizations.

6.3 Sampling-based profiling

Traditionally, PGO requires instrumentation to collect profile feedback. GCC have recently introduced AutoFDO, which can collect feedback using sampling-based profiling. We do not adopt AutoFDO in Tarax because it is limited to CFG arcs profiling and requires last branch record support from Intel processors. It cannot be applied to other processors or virtual machines. Moreover, the performance improvement of AutoFDO is 15%–22% lower than PGO [14, 15].

6.4 Further optimizations

The current Tarax implementation makes few modifications to the existing optimizations in GCC. We have mainly tried to take advantage of existing optimizations in GCC to create application-specific Linux kernels. Although the current results are already promising, we expect that more aggressive optimizations could be applied along this direction. For example, we have shown that with profile information available, the compiler makes better size-speed tradeoffs in comparison to aggressive size optimization (-Os) (Subsection 5.3). In some cases, profile information actually degrades performance compared to no profile information (Subsection 5.5). These indicate that more fine-grained control on GCC optimizations can potentially achieve greater improvement.

6.5 Limitations

During the implementation and evaluation of Tarax, we have made some choices and tradeoffs to stay focused on the main objective: to improve the application-specific performance of the Linux kernel.

Application selection. This paper focuses on optimizing the kernel for server applications because many server applications are known to be system-intensive. Unlike server applications, desktop and mobile applications are mostly interactive. The performance problems in these applications often reside in their models of human-computer interaction, instead of in the kernel. As a result, these applications may not benefit much from Tarax [16].

Experimental setup. In the experiments, we try to reduce factors that may influence application performance other than the kernel, which widely exist in real-world cases. For example, we use tmpfs to avoid the uncertainty of disk I/O performance. We also use high-speed network (10 Gbps) to increase throughput and stress the kernel, as such network is already used in many cloud environments.

Evaluation methodology. Since PGO is a machine learning like approach, strict evaluation may need separate training and testing inputs. For example, SPEC CPU2006 requires that only the training input can be used in PGO. However, we do not strictly follow the rules because we want to emphasize the potential benefits of Tarax. Although we use the same benchmarking tool for both training and testing, random workloads are generated for each execution.

6.6 Future directions

Building virtual appliances for cloud. As most cloud servers run virtual machines instead of OS on bare metal, we can extend Tarax to perform optimizations on the combination of the Linux kernel, the middleware and the application to create a specially optimized software stack which can be distributed and deployed as virtual appliances. We have applied Tarax and PGO to the famous LEMP stack, which consists of Linux, Nginx, MySQL and PHP, and achieved 5.4% performance improvement for WordPress.

Link-time optimization (LTO). Previous work has explored link-time compaction and specialization techniques to reduce memory footprint of the Linux kernel [17]. Tarax can also be combined with LTO. Performing optimization at link-time provides the compiler with more chances to carry out inter-procedural analysis, which will become more accurate when profile feedback is available.

Kernel refactoring. Another future direction is profile-guided kernel source code restructuring. The profile feedback can help us figure out the relations between kernel functions at runtime. We can use this information to eliminate unnecessary functions in the kernel with proper refactoring. We can also use the profile information to rearrange functions in translation units, which increases optimization opportunities.

7 Related work

We will discuss related work in three areas: ASOSs, general kernel performance optimization, and feedback-directed optimization (FDO).

7.1 Application-specific operating systems

Anderson first proposed the idea of ASOS [4], which is an application-specific design where as much of the OS as possible is pushed into runtime library routines linked with each application. Earlier ASOSs are based on kernel specialization [18], which can improve the performance of a specific system call. However, kernel specialization has not been applied to the whole kernel as Tarax does.

Since exokernel [5] was proposed, much research has been conducted on pursuing application-specific OS kernels based on the principle of library OS. Modern library OSs use virtual machine monitors as the exokernel [19]. The Mirage unikernels [6] are single-purpose appliances that are compile-time specialized with significant reduction in image sizes, and improvement in efficiency and security. Arrakis [8] and IX [9] are optimized for high I/O performance in datacenter workloads. They both adopt the library OS principle and use virtualization technologies to accelerate I/O.

A library OS is specific to applications. Although it can achieve significant size and speed improvement, library OS typically requires entire reimplementation of the kernel, and even the applications running on it. Instead of reimplementing an application-specific kernel, Tarax leverages profile-guided recompilation, such that the kernel can be optimized for each application without source code modification to either the kernel or the application. Furthermore, we can apply PGO on customized OS kernels such as library OS, to bring extra application-specific performance benefits on top of the library OS design.

7.2 Kernel optimization

Improving kernel performance [20] is an everlasting topic in the OS research community. With every generation of computer innovations, there is extensive research work on how to improve the OS performance accordingly [21]. Besides research publications on kernel performance [12, 22], many more optimizations have been applied to the Linux kernel to fix performance bugs and improve its performance; but most of these implementation efforts have never been published. For Linux file system and memory management, 8% and 27.4% of patches are for performance optimization respectively [23, 24].

All these performance optimizations to the kernel focus on specific performance problems, but are general to applications. Tarax pursues the opposite: it targets the whole kernel but is specific to each application scenario.

7.3 Feedback-directed optimization

FDO is a more general concept than PGO. FDO is used to describe all techniques that alter the execution of a program based on tendencies observed in its present or past runs. PGO alters the execution of the target program via compilation, based on tendencies observed in the past runs of the program.

Previous studies have explored kernel performance improvement opportunities using profile-based compiler optimizations [25, 26]. Although they share similar goals to Tarax, they perform kernel PGO on systems like HP9000/720 [25] and AS/400 [26], which have been outdated for decades. In contrast, Tarax applies PGO to the Linux kernel, which is much more complex, widely adopted and well supported.

Profile feedback can also play an important role in specific optimizations for the kernel such as I-cache packaging [27] and on-demand code loading of infrequently executed code [28].

Although FDO/PGO techniques have been extensively used in user applications, they have not been widely adopted in the kernel. To the best of our knowledge, Tarax is the first comprehensive approach that enables PGO on the Linux kernel to achieve significant performance improvement.

8 Concluding remarks

We have presented Tarax, a compiler-based approach that takes advantage of PGO to construct ASOSs. Specifically, Tarax extends the current PGO implementation in GCC to enable Linux kernel instrumentation, profiling and application-specific optimization.

Experimental results on six popular server applications show that Tarax could improve the Linux kernel performance by up to 16%. Detailed analysis has provided insights on how profile feedback helps GCC perform better optimizations on the Linux kernel in an application-specific manner. With Tarax, we believe there will be abundant opportunities to improve the Linux kernel performance further for each application running on it.

Acknowledgements This work was partly supported by National Key Research and Development Program (Grant No. 2017YFB1001904), and National Natural Science Foundation of China (Grant No. 61772042).

References

1 Mei H, Guo Y. Network-oriented operating systems: status and challenges (in Chinese). Sci Sin Inform, 2013, 43: 303–321
2 Mei H, Guo Y. Toward ubiquitous operating systems: a software-defined perspective. Computer, 2018, 51: 50–56
3 Mei H. Understanding "software-defined" from an OS perspective: technical challenges and research issues. Sci China Inf Sci, 2017, 60: 126101
4 Anderson T E. The Case for Application-Specific Operating Systems. Berkeley: University of California, 1992
5 Engler D R, Kaashoek M F, O'Toole J J. Exokernel: an operating system architecture for application-level resource management. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, 1995. 251–266
6 Madhavapeddy A, Mortier R, Rotsos C, et al. Unikernels: library operating systems for the cloud. In: Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, Houston, 2013. 461–472
7 Manco F, Lupu C, Schmidt F, et al. My VM is lighter (and safer) than your container. In: Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, 2017. 218–233
8 Peter S, Li J L, Zhang I, et al. Arrakis: the operating system is the control plane. ACM Trans Comput Syst, 2015, 33: 11
9 Belay A, Prekas G, Klimovic A, et al. IX: a protected dataplane operating system for high throughput and low latency. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), Broomfield, 2014
10 Yuan P F, Guo Y, Chen X Q. Experiences in profile-guided operating system kernel optimization. In: Proceedings of the 5th Asia-Pacific Workshop on Systems, Beijing, 2014
11 Gupta R, Mehofer E, Zhang Y. Profile Guided Code Optimizations. Boca Raton: CRC Press, 2002
12 Boyd-Wickizer S, Clements A T, Mao Y, et al. An analysis of Linux scalability to many cores. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, Vancouver, 2010
13 Melo A. The new Linux 'perf' tools. In: Proceedings of the 17th International Linux System Technology Conference (Linux Kongress), Nuremberg, 2010. 21–24
14 Chen D H, Vachharajani N, Hundt R, et al. Taming hardware event samples for FDO compilation. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, Toronto, 2010. 42–52
15 Chen D H, Li D X, Moseley T. AutoFDO: automatic feedback-directed optimization for warehouse-scale applications. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization, Barcelona, 2016. 12–23
16 Yuan P F, Guo Y, Chen X Q, et al. Device-specific Linux kernel optimization for android smartphones. In: Proceedings of the 6th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, Bamberg, 2018. 65–72
17 Chanet D, Sutter B D, Bus B D, et al. Automated reduction of the memory footprint of the Linux kernel. Trans Embed Comput Syst, 2007, 6: 23
18 Pu C, Autrey T, Black A, et al. Optimistic incremental specialization: streamlining a commercial operating system. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, 1995. 314–321
19 Wang X L, Luo T W, Hu J Y, et al. Evaluating the impacts of hugepage on virtual machines. Sci China Inf Sci, 2017, 60: 012103
20 Lynch W C. Operating system performance. Commun ACM, 1972, 15: 579–585
21 Chen J B, Bershad B N. The impact of operating system structure on memory system performance. SIGOPS Oper Syst Rev, 1993, 27: 120–133
22 Lozi J P, Lepers B, Funston J, et al. The linux scheduler: a decade of wasted cores. In: Proceedings of the 11th European Conference on Computer Systems, London, 2016

23 Lu L, Arpaci-Dusseau A C, Arpaci-Dusseau R H, et al. A study of Linux file system evolution. In: Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13), San Jose, 2013. 31–44
24 Huang J, Qureshi M K, Schwan K. An evolutionary study of Linux memory management for fun and profit. In: Proceedings of 2016 USENIX Annual Technical Conference (USENIX ATC 16), Denver, 2016. 465–478
25 Speer S E, Kumar R, Partridge C. Improving UNIX kernel performance using profile based optimization. In: Proceedings of USENIX Winter 1994 Technical Conference, San Francisco, 1994
26 Schmidt W J, Roediger R R, Mestad C S, et al. Profile-directed restructuring of operating system code. IBM Syst J, 1998, 37: 270–297
27 Flower R, Luk C K, Muth R, et al. Kernel optimizations and prefetch with the Spike executable optimizer. In: Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4), Austin, 2001
28 Chanet D, Cabezas J, Morancho E, et al. Linux kernel compaction through cold code swapping. In: Transactions on High-Performance Embedded Architectures and Compilers II. Berlin: Springer, 2009. 173–200