Automated Hot_Text and Huge_Pages: An Easy-to-Adopt Solution Towards High Performing Services

Zhenyun Zhuang(✉), Mark Santaniello, Shumin Zhao, Bikash Sharma, and Rajit Kambo

Facebook, Inc., 1 Hacker Way, Menlo Park, CA 94025, USA
{zhenyun,marksan,szhao,bsharma,rkambo}@fb.com

Abstract. Performance optimizations of large scale services can lead to significant wins on service efficiency and performance. CPU resource is one of the most common performance bottlenecks, hence improving CPU performance has been the focus of many performance optimization efforts. In particular, reducing iTLB (instruction TLB) miss rates can greatly improve CPU performance and speed up service running.

At Facebook, we have achieved CPU reduction by applying a solution that firstly identifies hot-text of the (software) binary and then places the binary on huge pages (i.e., 2 MB+ memory pages). The solution is wrapped into an automated framework, enabling service owners to effortlessly adopt it. Our framework has been applied to many services at Facebook, and this paper shares our experiences and findings.

Keywords: Huge pages · Hot-text · Performance · iTLB miss

1 Introduction

Large Internet companies like Facebook feature large amount of back-end servers which serve billions of users that have various types activities (e.g., messaging, video streaming). These servers run many types of services [1]. Given the large scale of the Facebook computation/storage infrastructure, it is important to ensure our services are running efficiently.

Many types of performance improvement works have been done at various layers (e.g., OS, compiler, application/code level, storage level) targeting different services. At Facebook, we have been treating performance improvement works seriously (e.g., [2,3]) by various types of optimizations. Over the years, we have achieved significant amount of efficiencies and better service performance across the fleet. To gain concrete understanding of the cost-saving scale, consider a service that runs on 100 K servers. Assuming the service is bottlenecked by CPU usage, and a performance improvement effort that saves 1% on server CPU usage will result in about 1 K servers being saved.

Performance improvement of services requires software profiling to identify the top performance bottlenecks, root-causing the fundamental issues, proposing

J. Miller et al. (Eds.): ICWS 2019, LNCS 11512, pp. 147–162, 2019.
https://doi.org/10.1007/978-3-030-23499-7_10

solutions to address the issues, implementing the solutions, and testing to verify the effectiveness of the solutions. We have been continuously profiling thousands of services across our fleet for various types of performance inefficiencies. We found that CPU resource is one of the most common performance bottlenecks, hence improving CPU performance has been the focus in many performance efforts.

One type of service inefficiency is high iTLB (instruction Translation Look-aside Buffer) miss rate, which causes CPU to run ineffectively[1]. The penalty of iTLB misses is significant as the access latency difference between *hit* and *miss* can be 10–100 times of difference. A *hit* only takes 1 clock cycle, while a *miss* takes 10–100 clock cycles. To understand more about such latency penalty, let's assume *hit* and *miss* take 1 and 60 cycles, respectively. Thus a 1% miss rate will result in average access latency being 159 cycles, or 59% higher than not having any misses (i.e., 1 cycle).

For a service that experiencing high iTLB miss rate, reducing iTLB miss rates can greatly improve CPU performance and speed up service running time. Various optimization approaches that impprove the software binaries can be applied to reduce iTLB misses. Overall there are three types of optimizations based on the different stages of compiling the source code (i.e., compile/link/post-link time). Examples include optimizing compiler options to reorder functions so that hot functions are located together, or using FDO (Feedback-Driven Optimization) [4] to reduce the size of code regions. In addition to such compiler optimizations that help reduce iTLB miss rate, we also place the binary code on huge pages to further speed up the running services.

In this work, we combine both types of optimizations (i.e., identifying *hot-text* to co-locate frequently accessed functions and deploying the optimized binary on *huge pages*) to reduce iTLB miss rates. More importantly, we design and implement an automated work flow to make the optimizations transparent and maintenance-free for service owners. As a result, various services can benefit from this solution with minimum efforts, practically rendering this easy-to-adopt solution as a "free lunch optimization" for service owners.

Note that though the *hot-text* optimization mostly applies to services written in statically compiled languages (e.g., C/CPP), *huge page* optimization can apply to all services. Given the fact that many of largest scale backend infrastructures in the world (e.g., Facebook, Google, Microsoft) are written in C/CPP, thanks to C/C++'s high efficiency, our proposed solution can be applied to many services running on these infrastructures. Furthermore, for dynamically compiled languages (e.g., Java), the insights gained in this work can also help improve their compiling performance (e.g., in JVM).

This work shares our design, efficiency gains and some issues found. In particular, we focus on the key questions that could be asked by potential adopters including:

– What is the performance issue this solution addresses? For instance, why is high iTLB miss rate bad?

[1] Please refer to Sect. 2 for detailed explanations of iTLB misses.

– What is the design of the solution (i.e., how does it work internally)?
– How much code change is needed?
– How much maintenance overhead is involved?
– What is the downside of this solution?
– How to verify the optimization is applied to my service?

The following writing is organized as follows. Section 2 provides relevant background knowledge. Section 3 walks through the high level design, followed by detailed work flow of the solution in Sect. 4. Section 5 presents performance results of applying this solution. Some related issues are discussed in Sect. 6, and related works are presented in Sect. 7. Finally Sect. 8 concludes the paper.

2 Background

2.1 ITLB (Instruction TLB) Misses

In x86 computing architectures, memory mappings between virtual and physical memory are facilitated by a memory-based page table. For fast virtual-to-physical translations, recently-used portions of the page table are cached in TLB (translation look-aside buffers). There are two types of TLBs: data and instructions, both are of limited sizes.

Since memory access latency is much higher than TLB access latency, address translations that hit TLBs are much faster than missing TLBs. Invariably, the more translation requests that miss the TLBs and have to fall back to page tables (aka, 'page walks'), the slower the instruction executions. iTLB miss rate is a metric to estimate the performance penalty of page walks induced on iTLB (instruction TLB) misses. When the iTLB miss rate is high, a significant proportion of cycles are spent handling the misses, which results in slower execution of the instructions, hence sub-optimal services.

2.2 Huge Pages

Today's computing architecture typically support larger page sizes (2 MB and 1 GB on x86_64, both referred to as huge pages) in addition to the traditional 4 KB pages size. Huge pages reduce number of TLB entries needed to cover the working set of the binary, leading to smaller page tables and reducing the cost of page table walks.

There are two ways of obtaining huge pages on Linux: (1) using THP (transparent huge pages) [5] and (2) reserving huge pages and mounting them as *hugetlbfs* in the application. THP requires minimum changes to the application, however the availability of huge pages is not guaranteed. To reserve huge pages, applying configurations such as *hugepagesz = 2 MB hugepages = 64* (i.e., reserving 64 huge pages of 2 MB each) when booting kernel works.

3 Design

3.1 Overview

When running a service on servers, the corresponding binary needs to be loaded into memory. The binary consists of a set of functions, and they collectively reside in the *text* segment of the binary and are typically loaded during execution using 4 K pages. Each page attempts to occupy an iTLB entry for the virtual-to-physical page translation. Since commodity servers typically have limited number of iTLB entries (e.g., 128 entries for 4 KB pages in *Intel HasWell* architecture [6]), iTLB misses will occur if the text segment is larger than the iTLB entries can cover (e.g., 128 * 4 KB = 512 KB). iTLB misses are counted towards CPU time and are effectively wasted CPU time.

iTLB misses can be reduced by identifying and aggregating frequently accessed instructions into *hot-text* in order to increase spatial locality. By packing hot functions into *hot text*, instruction fetching and prefetching can be more effective and faster, hence a high-performing server and service. Based on our studies with many services, at Facebook more than 90% of the code is *cold* and the remaining is *hot*. By separating hot from cold instructions, expensive micro-architectural resources (iTLB and caches) can more efficiently deal with the hot segment of a binary, resulting in performance improvement.

This benefit can be further enhanced by putting *hot-text* on huge pages for sufficiently large binary (i.e., larger than the regular page size of 4 KB * iTLB entries). By using a single TLB entry, a single 2 MB huge page covers 512 times as much code as a standard 4 K page. More importantly, CPU architectures typically feature some number of TLB entries for huge pages, and they will sit there idle if no huge pages are used. By employing huge pages, those TLB entries can be fully utilized.

3.2 Design Elements and Rationales

The solution consists of three key elements: (a) hot-text optimization, (b) huge page placement, and (3) automated and decoupled pipeline.

The *hot-text* optimization consists of the following steps. First, identifying hot instructions by profiling the running binary. Linux perf tool is used for this purpose. We initially used stack traces, but later switched to LBRs [7] for better data quality and less data footprint. Second, sorting the profiled functions based on access frequencies. We use a tool called HFSort [8,9] to create an optimized order for the hot functions. Finally, a linker script will optimize the function layout in the binary based on the access orders. The result is an optimized binary.

Once the optimized binary with hot-text is obtained, the hot-text region can be further placed on huge pages. We designed an almost-transparent approach which needs little code change for service owners. Specifically, we pre-define a function that remaps the hot-text to huge pages, and all a service owner has to do is calling a pre-defined function early in the *main()* function

Note that *isolating hot-text* and *placing on huge pages* are complementary optimizations, and they can work independently; but combining them achieves best optimization results.

The traditional approach of applying hot-text optimization and huge page placement requires multiple steps, mixes source code and profiled data during linking phase, and involves manual profiling and refreshing, which prevents the solution from being widely deployed. We built a pipeline to automate the entire process, practically making this solution an easy-to-adopt and maintenance-free solution for all applicable services.

The pipeline also decouples the service's profile data from source code, hence allowing smooth and frequent profiling update to accommodate code/usage changes. Specifically, the profiled data are stored in separate storage that is different from source code repository. The advantages of the decoupling is each profiling update becomes part of the linearized code commit history, just like any other source code change. The profiling updates can be treated as source code, enabling easy check-in, review and reverting. Moreover, the profiled data files are stored and retrieved using file handles, hence we don't actually pay the cost of storing these huge almost-not-human-readable files in the source code repository.

In addition to helping reducing iTLB misses, the solution can also help reducing iCache misses. Computing instructions are fetched from memory and executed by CPUs. To expedite the instruction accesses, smaller and faster caches (iCache) are used to hold the instructions. iCache misses can occur when the binary working set is bigger than the iCache size. Caches have multiple levels, and the lowest level iCache is often times much smaller than the binary working set. iCache misses delay the CPU execution due to longer memory access time.

4 Detailed Work Flow

We now present the detailed steps and components of this solution. Note that neither *isolating hot-text* nor *using huge pages* is an invention, and both of them have been tried in several scenarios. However the naive adoption of the solution used to involve multiple manual and tedious steps (e.g., profiling, linking, regularly refreshing the profiles), hence few services have benefited from the solution. To address this issue, we designed an automated framework and data pipelines to remove the manual involvement by wrapping, aggregating and automating each steps. As a result, the solution suite becomes *maintenance free* and requires little code change.

4.1 Diagram and Components

The steps and components of this framework are shown in Fig. 1. Largely it consists of three components of profiling, linking and loading.

- *Profiling.* The profiling component is shown on the top of the diagram. A data-gathering job runs weekly to profile the running service[2]. The job is

[2] We observed that most services are relatively stable with regard to hot functions, hence weekly profiling suffices.

using our *Dataswarm* framework [10], a data storage and processing solution developed and used by Facebook. The job profiles running information of the service (e.g., hot functions), and the profiling is carefully controlled to have very light overhead. Profiled data is then sent to a permanent storage called *Everstore* [11], a disk based storage service.
- *Linking.* When building the service package, the linker script retrieves the profiled hot functions from Everstore and reorders functions in the binary based on the profiles.
- *Loading.* When loading the service binary, OS makes best efforts to put hot-text on huge pages. If no huge pages available, then put on regular pages.

4.2 Code Changes

For a new service that would like to apply this optimization, only three places of small changes (boilerplate code) are needed: (a) Dataswarm pipeline creation; (b) Source code (cpp files) change; and (c) Build configuration change.

Storage Pipeline Creation. On top of our data storage framework of Dataswarm, a data processing job regularly refreshes the profiled hot functions

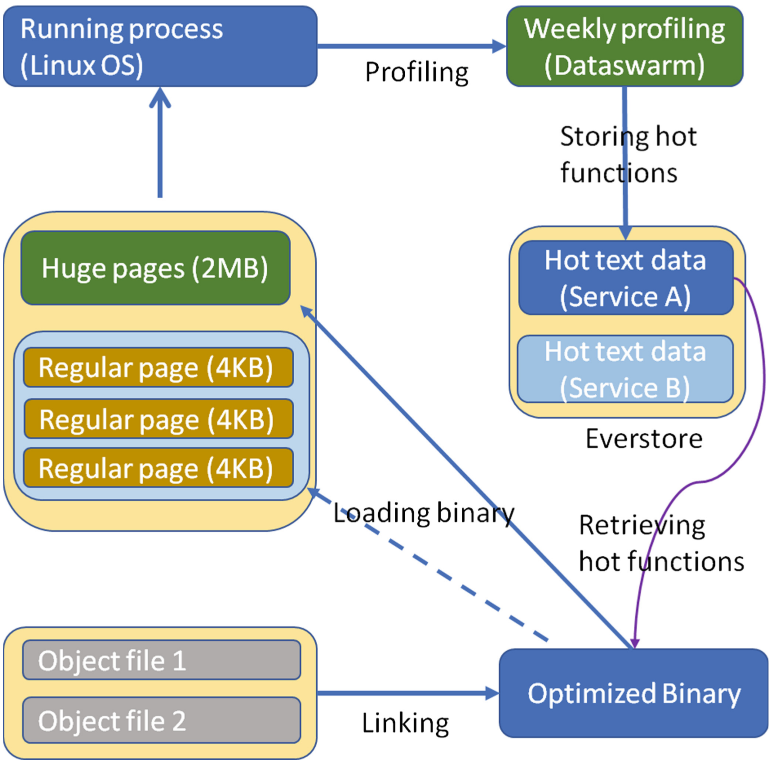
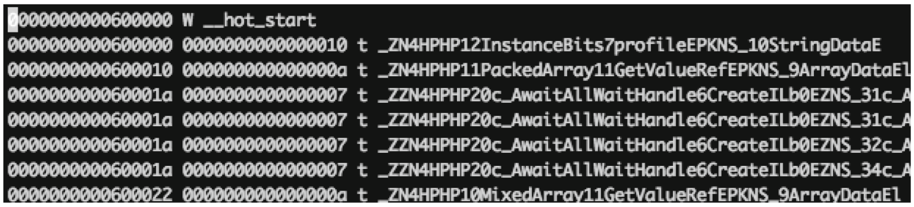


Fig. 1. Diagram of hot-text and huge pages

profiles to reflect updates on the service: code changes and usage pattern changes. The data pipeline kicks off a Dataswarm job weekly for each service. When the job kicks off, it profiles the specified service and generates a file containing the hot functions. The hot function file is stored to Everstore. The file is uniquely identified by a file handle and content checksum.

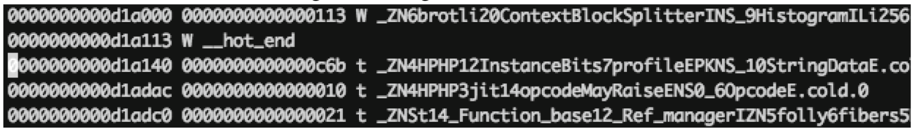
The Dataswarm job also automatically creates a code diff (i.e., code checkin) which updates a meta file containing the newly generated hot-function file handle and checksum, it then automatically lands the diff (i.e., updating the meta file) to the services' source code directory.

Source Code Change. The framework only requires two lines of code change to source code's main cpp file. Specifically, the first line of code change is to include a header file that defines the function that is responsible for putting the



```
0000000000600000 W __hot_start
0000000000600000 0000000000000010 t _ZN4HPHP12InstanceBits7profileEPKNS_10StringDataE
0000000000600010 000000000000000a t _ZN4HPHP11PackedArray11GetValueRefEPKNS_9ArrayDataEl
000000000060001a 0000000000000007 t _ZZN4HPHP20c_AwaitAllWaitHandle6CreateILb0EZNS_31c_A
000000000060001a 0000000000000007 t _ZZN4HPHP20c_AwaitAllWaitHandle6CreateILb0EZNS_31c_A
000000000060001a 0000000000000007 t _ZZN4HPHP20c_AwaitAllWaitHandle6CreateILb0EZNS_32c_A
000000000060001a 0000000000000007 t _ZZN4HPHP20c_AwaitAllWaitHandle6CreateILb0EZNS_34c_A
0000000000600022 000000000000000a t _ZN4HPHP10MixedArray11GetValueRefEPKNS_9ArrayDataEl
```

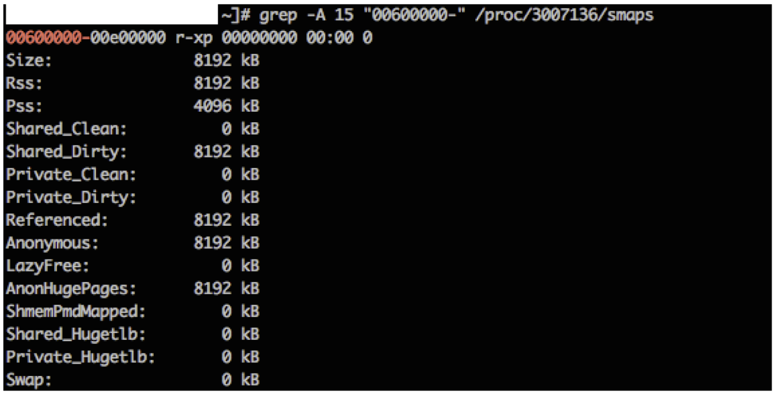
(a) Hot-text region: starting at 0x600000



```
0000000000d1a000 0000000000000113 W _ZN6brotli20ContextBlockSplitterINS_9HistogramILi256
0000000000d1a113 W __hot_end
0000000000d1a140 0000000000000c6b t _ZN4HPHP12InstanceBits7profileEPKNS_10StringDataE.co
0000000000d1adac 0000000000000010 t _ZN4HPHP3jit14opcodeMayRaiseENS0_6OpcodeE.cold.0
0000000000d1adc0 0000000000000021 t _ZNSt14_Function_base12_Ref_managerIZN5folly6fibers5
```

(b) Hot-text region: ending at 0xd1a113 (total size: 7.4 MB)

Fig. 2. Verifying the existence of hot-text



```
[redacted]~]# grep -A 15 "00600000-" /proc/3007136/smaps
00600000-00e00000 r-xp 00000000 00:00 0
Size:               8192 kB
Rss:                8192 kB
Pss:                4096 kB
Shared_Clean:          0 kB
Shared_Dirty:       8192 kB
Private_Clean:         0 kB
Private_Dirty:         0 kB
Referenced:         8192 kB
Anonymous:          8192 kB
LazyFree:              0 kB
AnonHugePages:      8192 kB
ShmemPmdMapped:        0 kB
Shared_Hugetlb:        0 kB
Private_Hugetlb:       0 kB
Swap:                  0 kB
```

Hot-text placed on huge pages (AnonHugePages: 8192KB, or 4 huge pages)

Fig. 3. Verifying the hot-text is deployed on huge pages (host name anonymized)

hot functions to huge pages if possible, and it achieves this by copying the text segment of the binary and using *mmap()* to map the text segment to huge pages.

The second line of code change is to call *hugify_self()* in *main()* function. This function needs to be called in the beginning of the *main()* function for the best result.

Build Configuration Change. The build configuration change allows the profiled data to be retrieved and used during linking. Specifically, it adds a few lines to build TARGETS file. It retrieves the meta file that contains the *hot functions* information of the particular service from *Everstore*. The retrieval is via HTTP, which is supported by *Everstore* and *Buck* [12] using *remote_file* call. To ensure correctness, the meta file is checked by *SHA1* hash.

4.3 Verifying the Optimization Is in Place

To make sure the optimization is in place, two things need to be verified: hot-text is in the binary, and hot-text is placed on huge pages. In the following, we demonstrate the verification steps under Linux.

Hot-text verification. If a binary has the hot-text extracted, the binary should have symbols that indicate the starting/ending address of the hot-text. Specifically, the hot-text region starts with *__hot_start* and ends with *__hot_end*. *nm* utility [13] can list the symbols from the binary, and by examining the output of the symbols (*nm -S –numeric-sort /proc/pid/exe*, where *pid* is the process id of the running binary), we can verify the existence of hot-text.

Let's examine an example. As it shows in Fig. 2 the hot-text region starts from *0x600000* and ends at *0xd1a113*. The total size is 7446803 bytes, or about 7 MB.

Huge pages verification. To verify the hot-text is stored on huge pages, we can examine the running process by checking the *smaps* file, e.g. *grep -A 20 "600000-" /proc/pid/smaps*. As shown in Fig. 3, the *AnonHugePages* allocated is 8192 KB, or about 4 huge pages (2 MB each), indicating the hot-text is loaded to huge pages. In scenarios where hot-text is not put on huge pages, it will show *AnonHugePages: 0 KB*.

5 Performance Results

5.1 Experiences with Many Services

We applied the *hot_text* and *huge_page* solution to many services and gained deep understanding of the improvement degrees on various types of performance metrics. Based on our experiences, typically the most immediate performance improvement is reduced iTLB miss rate, it can also help on other metrics.

– *iTLB miss rate.* This is the most obvious benefit, we consistently see up to 50% iTLB cache miss drop for almost all the services that adopted this solution.

- *CPU usage.* CPU usage typically drops by 5% to 10% across the services we worked on.
- *Application throughput.* Applications typically enjoys higher throughput, thanks to the reduced cpu usage.
- *Application query latency.* The query latency mostly will drop due to reduced iTLB cache miss and faster execution.

Note that depending on services and workload characteristics, not all of these metrics will improve. In addition, different services see improvement on different set of performance metrics, and the degrees of improvement vary.

5.2 An Example Service

To understand more about the performance metrics and the extent of improvement, we choose a service to elaborate on the detailed results. The particular service is an online one which directly serves the users, hence both application throughput and latencies are important. Moreover, the service fleet's footprint is significant with many servers, and it is desired to reduce CPU usage such that a single host can serve more users and the service fleet can be shrinked.

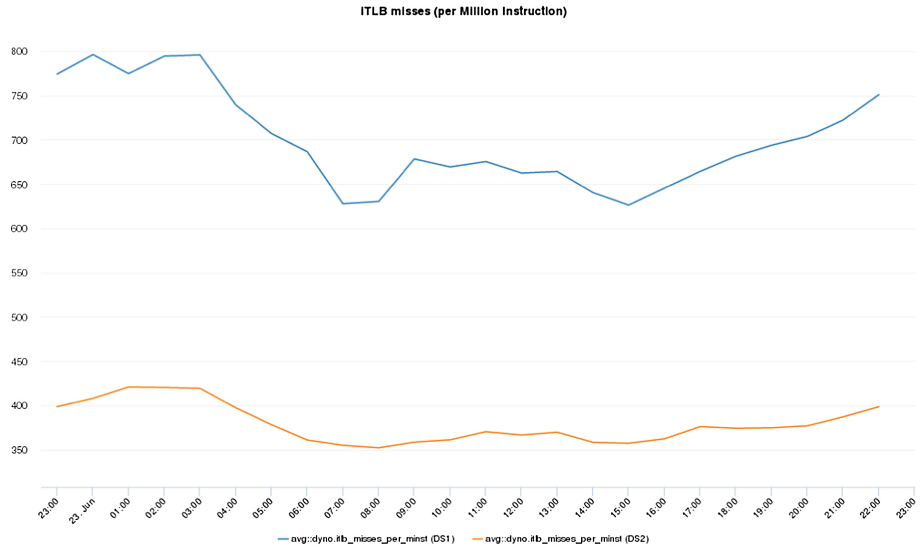
We will examine both application level and server system level metrics. For application level metrics, we consider both query/call latencies and application throughput (i.e., queries per second). We also consider multiple percentiles of latencies. Overall we observe 15% throughput improvement and 20% of latency reduction.

For system level metrics, we consider host cpu usage (total, user and kernel usages) and iTLB miss rates. The iTLB miss rate is almost halved, and cpu usage is 5% lower. Across the 50+ services we have worked on, applying this solution typically reduces cpu usage by 5% to 10%. We also estimated that about half of such cpu reduction gain comes from *hot-text*, while the other half comes from *huge page*.

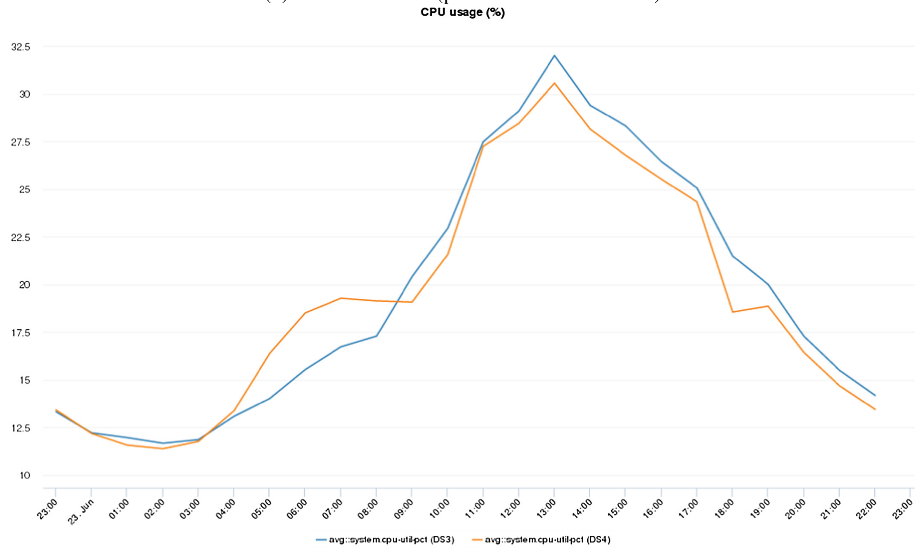
Server System Performance. The iTLB miss rate is shown in Fig. 4(a). Before applying the solution, the iTLB miss rate is up to 800 iTLB misses per million instructions during peaks, which is very severe. After the optimization is in place, iTLB miss rate almost drops by half. Specifically, during peaks, the highest iTLB miss rate is about 425 misses per million instructions, or a 49% drop.

As a result of the dropped iTLB miss rate, the CPU usage drops by 5% (i.e., from 32% to 30.5% at their peaks), as shown in Fig. 4(b). The user level cpu drops by 15%, while kernel level cpu increases by about 7%, as shown in Figs. 5(a) and (b), respectively.

Application Level Performance. Application level metrics are shown in Figs. 5(c) and 6. The blue curve is *before optimization* (i.e., data sets of DS1/DS3/F1), and the yellow curve is *after optimization* (i.e., data sets of



(a) iTLB miss rate (per million instructions)

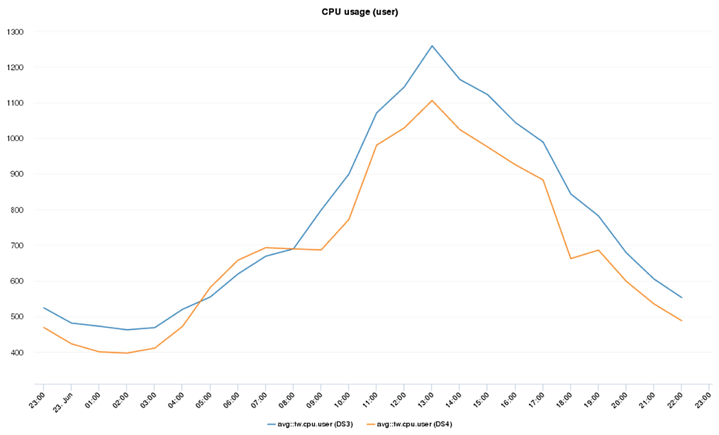


(b) host cpu usage (%)

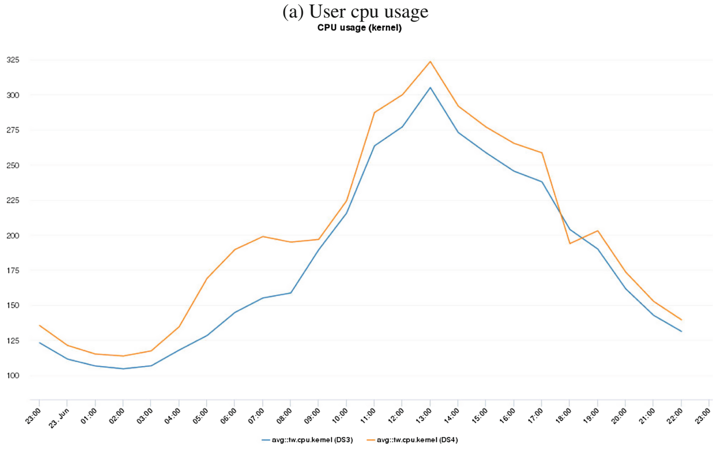
Fig. 4. System level performance (iTLB miss rates and host cpu usage)

DS2/DS4/F2). P50 of application query latencies drops by up to 25%, P90 drops by up to 10%, and P99 drops by up to 60%.

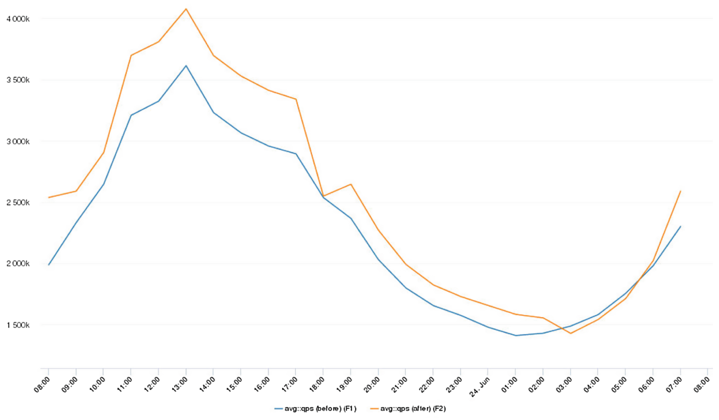
The application throughput (qps)increases by up to 15% (i.e., peak throughput increases from 3.6 M qps to 4.1 M qps). It is very delightful to see both throughput and latency improvements at application level.



(a) User cpu usage

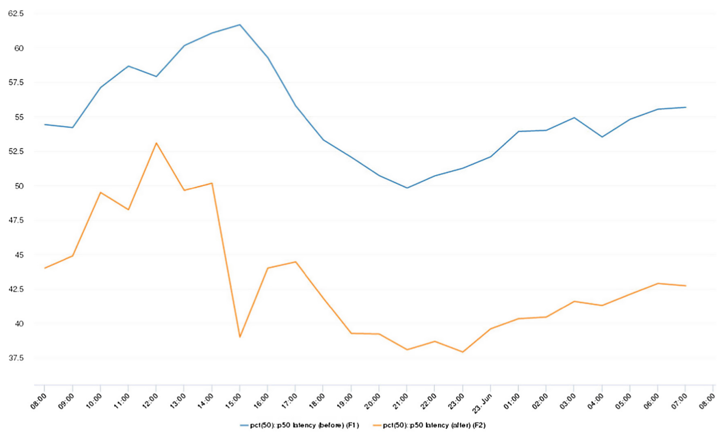


(b) Kernel cpu usage

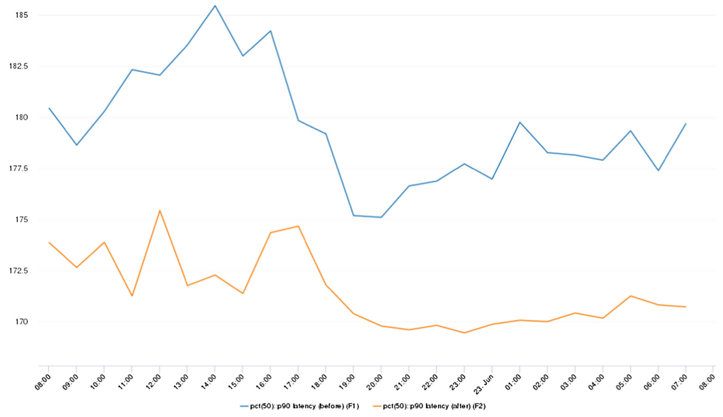


(c) Throughput: Qps (Queries per second)

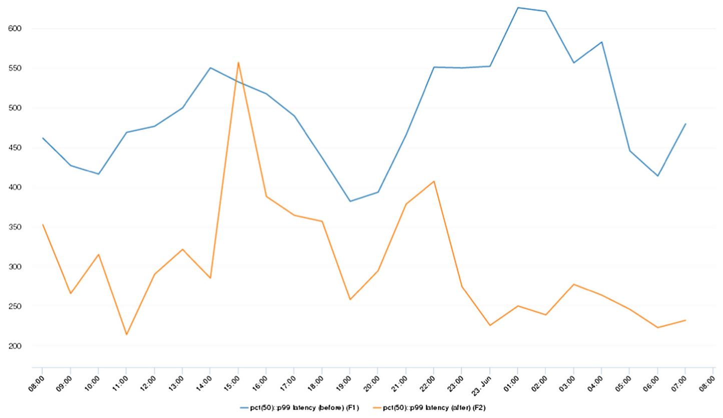
Fig. 5. System (User/kernel CPU) and application level performance (Throughput) (Color figure online)



(a) P50 query latency improvement (ms)



(b) P90 query latency improvement (ms)



(c) P99 query latency improvement (ms)

Fig. 6. Application level performance (query latency) (Color figure online)

6 Discussions

6.1 Hot-Texts Not Being Placed on Huge Pages

During our performance optimizations with some services, we happened to notice that for some services that already incorporated the solution we propose in this paper, some hosts do not place hot-text on huge pages. We digged into that issue and found it is due to the way the huge pages are handled.

Currently the huge pages are not pre-allocated during OS starts, instead, it is a best-effort. When the binary loads, OS will try to find continuous memory big enough for a huge page to place hot-text. If the memory is sufficiently fragmented and no huge pages can be found, then it will fall back to use regular pages.

To what degree does this issue occur depends on many factors that affect memory fragmentation, including system *up-time* and memory pressure level. For instance, we have found *Haswell* hosts are more likely to have such issue than *Broadwell* hosts, thanks to the former's higher load and memory pressure.

In addition to reserving huge pages, another solution is to *defrag* memory before loading the service binary (e.g., */roc/sys/vm/drop_caches* and */proc/sys/vm/compact_memory*). Memory defragmentation can compact fragmented pages, hence resulting in higher chances of being able to find huge pages when loading the binary.

6.2 Hot Function File Retrieval Failure

Building the binary package with this optimization needs to retrieve the hot function file from *Everstore*. *Everstore* is very reliable based on our experience, and only 1 failure is encountered when loading hot functions in a year. But in the worst scenario where if it fails to retrieve the hot function file, the binary build will fail.

6.3 Downside of This Approach

There is very little downside (e.g., very little performance overhead) about using this approach, thanks to the automatic profiling and diff landing. The major downside is longer binary-building time.

Another concern about using huge pages is the memory waste (i.e., up to 1 page), depending on the way they are used. Reserved huge pages are always paged into memory, and the recommendation is to reserve just-enough pages. THP, on the other hand, is free from this concern. The possible memory waste is when a huge page is not entirely used. When the number of huge pages used is small compared to the total available memory, this concern might be negligible. Based on our experiences, most services only use a few 2 MB huge pages, which is trivial compared to the total available memory (e.g., hundreds of GBs).

7 Related Work

Many works optimize the performance of the binary using various types of techniques during different phases of compiling, linking and post-linking of the binaries.

During compiling time, instrumentation-based schemes have been employed by GCC and Microsoft C/C++. In GCC world, such optimization is called FDO (Feedback-Driven Optimization) [4], while Microsoft refers to it as PGO (Profile-Guided Optimization) [4]. These schemes also effectively re-compile hot code for *speed* and cold code for *size*. As a result, the overall code size is typically reduced by FDO/PGO. GCC's AutoFDO (Automatic Feedback Directed Optimizer, [14]) is another feature that uses run-time feedback mechanism to help compiler, enabling wider range of optimizations. Specifically, LLVM supports AutoFDO framework that easily converts linux *perf* output into LLVM consumable profile file.

During linking time, techniques such as the *hot-text* optimization described in this paper [9] use a linker script and operates on a function-by-function basis. Work [9] elaborates on some of the internal mechanisms to make this optimization happen, and we further build a framework to automate the entire process with an end-to-end pipeline. Safe ICF (Identical Code Folding) [15] takes another approach of detecting functions that contain redundancies and folding/merging functions into a single copy.

There are also post-link optimizers. BOLT (Binary Optimization and Layout Tool) [16,17] is a post-link optimizer developed to improve running performance of non-trivial binaries. It operates on a finer basic block granularity and achieves the goal by optimizing application's code layout based on execution profile gathered by sampling profilers (e.g., Linux *perf*). Specifically for Intel, Ispike [18] is another post-link optimizer.

At system level (i.e., Operating system and hardware), countless works have demonstrated the potentials and shared the experiences of speeding up software running on various types of OS and hardware. Work in [19] evaluates the accuracy of multiple event-based sampling techniques and quantifies the impact of the improvements claimed by many other techniques.

Moving up to application level, even more places can be optimized for better performance, thanks to the heterogeneity of different types of applications and services. At Facebook, we have designed and improved many services and products [11,20]. As an example, RocksDB [21] a persistent key-value store developed by Facebook, has been continuously optimized for many different scenarios [2,22].

8 Conclusion

Facebook, having one of the world's biggest computing infrastructures, treats performance optimizations seriously. During the course of various types of performance improvement efforts, we have accumulated techniques, tools and experiences to speed up our services. One of these approaches is an automated framework to incorporate both *hot-text* and *huge pages* and enable service owners to adopt this optimization with minimum effort. The solution identifies hot-text of the binary and places the binary on huge pages. The solution is further wrapped into an automated framework, enabling service owners to effortlessly adopt it. The framework has been applied to dozens of our services, proved effective and has significantly improved our service efficiencies.

Acknowledgements. The solution presented in this paper involves many peoples' efforts, which include new services or feature enhancements of existing services. In particular, we thank Guilherme Ottoni and Bert Maher for working on HFSort, Mark Williams for implementing hugify_self(), Denis Sheahan and Pallab Bhattacharya for substantiating a generic library, and Mirek Klimos for the support that allows automated refreshing of profiling data.

References

1. Chen, G.J., et al.: Realtime data processing at Facebook. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD 2016, New York, NY, USA (2016)
2. Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., Strum, M.: Optimizing space amplification in RocksDB. In: Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR 2017). Chaminade, California (2017)
3. Annamalai, M., et al.: Sharding the shards: managing datastore locality at scale with Akkio. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI 2018, Berkeley, CA, USA (2018)
4. Wicht, B., Vitillo, R.A., Chen, D., Levinthal, D.: Hardware counted profile-guided optimization. CoRR, vol. abs/1411.6361 (2014). http://arxiv.org/abs/1411.6361
5. Transparent Hugepage Support. https://www.kernel.org/doc/Documentation/vm/transhuge.txt
6. Intel HasWell Architecture. https://ark.intel.com/content/www/us/en/ark/products/codename/42174/haswell.html
7. Advanced usage of last branch records. https://lwn.net/Articles/680996/
8. HFSort. https://github.com/facebook/hhvm/tree/master/hphp/tools/hfsort
9. Ottoni, G., Maher, B.: Optimizing function placement for large-scale data-center applications. In: Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Piscataway, NJ, USA (2017)
10. Data pipelines at Facebook. https://www.meetup.com/DataCouncil-AI-NewYorkCity-Data-Engineering-Science/events/189614862/
11. Barrigas, H., Barrigas, D., Barata, M., Furtado, P., Bernardino, J.: Overview of Facebook scalable architecture. In: Proceedings of the International Conference on Information Systems and Design of Communication, ISDOC 2014 (2014)
12. Buck: A high-performance build tool. https://buckbuild.com/
13. NM utility. https://sourceware.org/binutils/docs/binutils/nm.html
14. Chen, D., 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, CGO 2016, New York, NY, USA (2016)

15. Tallam, S., Coutant, C., Taylor, I.L., Li, X.D., Demetriou, C.: Safe ICF: pointer safe and unwinding aware identical code folding in gold. In: GCC Developers Summit (2010)
16. Panchenko, M., Auler, R., Nell, B., Ottoni, G.: Bolt: a practical binary optimizer for data centers and beyond. In: Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, pp. 2–14. IEEE Press, Piscataway (2019)
17. Binary Optimization and Layout Tool. https://github.com/facebookincubator/BOLT
18. Luk, C.-K., Muth, R., Patil, H., Cohn, R., Lowney, G.: Ispike: a post-link optimizer for the Intel Itanium architecture. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO 2004, Washington, DC, USA (2004)
19. Nowak, A., Yasin, A., Mendelson, A., Zwaenepoel, W.: Establishing a base of trust with performance counters for enterprise workloads. In: Proceedings of the 2015 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC 2015, Berkeley, CA, USA, pp. 541–548 (2015)
20. Scaling server software at Facebook. In Applicative 2016, Applicative 2016, speaker-Watson, Dave (2016)
21. RocksDB: A persistent key-value store. https://rocksdb.org/
22. Ouaknine, K., Agra, O., Guz, Z.: Optimization of RocksDB for Redis on flash. In: Proceedings of the International Conference on Compute and Data Analysis, ICCDA 2017, New York, NY, USA (2017)