Statement of Research Interests

Yuxuan Zhang, University of Pennsylvania

I'm interested in understanding and mitigating the front-end bottleneck caused by cloud workloads. In particular, my research focuses on investigating and mitigating the front-end bottleneck at runtime. Prior research [1][2] has pointed out that the capacity of the instruction-cache of server processors cannot meet the requirement of the growth of today's data center applications and their growing instruction working set. Several profile-guided optimization (PGO) solutions [3][4][5] are proposed to address the gap between the instruction cache capacity and the code size. However, they all require halting the program to re-deploy the optimized application. After the terminated online service reboots, the new tasks assigned to the service may be different and prefer a different optimized code layout. Hence, to the re-deployed application, the profile data is always stale, which may lead to a sub-optimal or even a worse code layout being generated. Moreover, some of the PGO solutions lack flexibility, as they profile all cloud workloads and analyze their memory access behavior as a whole, rather than processing them individually. To address these problems, it's necessary to move offline analysis and optimization to run online instead. Therefore, I set my research goal to be designing and implementing an online PGO-based code layout optimization solution with low overhead.

My current research investigates the feasibility of changing code layout at runtime, and it includes two parts: (1) the Build Accelerator and (2) Online COde LayoUt optimization System (OCOLUS). The work for the Build Accelerator has proved that, for batch processing jobs, only a small subset of profiling data would be enough to produce an optimized binary for runtime optimization, and the ongoing project OCOLUS investigates the feasibility of replacing code layout at runtime by inserting the optimized machine code into a target running process. In this statement, I summarize my prior works, and detail my future research plans.

1 Ongoing Research - Replace Code Layout at Runtime

The Build Accelerator: With the development of continuous method of software, software engineers use Continuous Integration/Continuous Deployment (CI/CD) pipelines to avoid too much human intervention from the development of new code until its deployment. However, in CI/CD pipelines, each committed version of software gets compiled from scratch and tested, leading to a lot of cycles spent in the compiler. Also, compilers suffer heavily from front-end stalls, which further exacerbates the problem of wasting too much time on the compilation in the CI/CD pipelines. So, the Build Accelerator, which accelerates the compilation by changing code layout of the compiler, was chosen as my first step to explore the relationship between the quantity of profiling data and the quality of code layout optimization.

Since the nature of building a large project from source is to re-invoke the same compiler executable to turn source files into object files individually and then link them together, it becomes possible to break the profiling of the whole compilation into small pieces due to the reuse of the compiler – each time there is a compiler invocation, corresponding profiling data can be generated. Hence, a number of profiling samples are available long before the overall build finishes. The Build Accelerator takes advantage of a small portion of the available profiling samples to construct a new compiler binary by feeding both the profiling data and the compiler executable itself to the post-link optimizer BOLT [6], a system from Facebook. After BOLT finishes building the optimized binary, the Build Accelerator replaces the original compiler executable with the optimized one. The result shows that building Clang by applying the Build Accelerator takes 108.3 minutes, while the native build takes 135.93, which means applying the Build Accelerator yields a 20.32% speedup when building Clang.

Meanwhile, breaking large profiling data down into pieces provides the opportunity to examine to what degree the profiling is adequate to perform the code layout optimization, since the Build Accelerator can tune the size of profiling samples and test the speedup. My empirical results show that for Clang, we got a 22.65% speedup with the full profiling info, compared to 21.83% speedup when we use just 1% of the profiling data. This result shows that online profiling can be very lightweight.

Online COde LayoUt optimization System (OCOLUS): Online services handle thousands of queries or tasks per hour, and these queries or tasks may change rapidly depending on how the master nodes forward requests or assign tasks. As a result, a newly generated code layout optimization may become stale after a relatively short period of time. So it is necessary to change the code of the running process again once the system detects that the current code layout is not optimal for the incoming queries or tasks anymore. This brings new questions to the table: How do we define and detect whether the code layout optimization is optimal? After we find the code layout is not optimal anymore, what kind of optimization should we apply to the code? Monitoring performance introduces additional overhead to online services, because it requires CPU resources. So, how often should we invoke the performance monitoring?

To answer these questions, I plan to implement OCOLUS, a system that always runs as a daemon process to detect, analyze real world online workloads and optimize them by changing their code layout at runtime. This work will start from investigating the feasibility of replacing code layout at runtime. Then the system will be extended by integrating the performance monitoring mechanism.

To investigate the feasibility of replacing code layout at runtime, I have been implementing the first version of OCOLUS. For now, OCOLUS has a standalone process that profiles target data center applications, produces optimized binaries of the applications and then replaces the code of running executables with that of binaries optimized by BOLT, all without terminating the process. The difference between the Build Accelerator and OCOLUS happens at the replacement phase. For the Build Accelerator, the target compiler binary is repeatedly invoked directly, which makes it possible to change the binary at a new process boundary, after some particular separate compilation. However, for most cloud workloads that run as a service, it is infeasible to wait until the service finishes and then change it. Therefore, replacing the code region of the running application is inevitable. Considering the fact that changing of code at runtime can easily crash the running process, I chose to conservatively replace code at function granularity and avoid changing functions that reside in the call stack. I have tested OCOLUS on levelDB and rocksDB, both of which are embedded persistent key-value store for fast storage, and the experimental results show that this design decision was feasible. My work now moves towards collecting the performance results from workloads that employ OCOLUS.

For future work on OCOLUS, I plan to apply more aggressive changes to the code of online cloud workloads, such as reordering functions and splitting cold code from hot code. All such approaches have been proved to have more performance gain in the offline settings. So they would also benefit our online version, although they require a more substantial modification on the code layout when the machine code insertion happens. After this, I'm going to integrate the performance monitoring mechanism, which should be able to decide when to invoke the performance measurement and identify whether the current code layout is optimal.

2 Other Future Research Directions

Online PGO-based instruction-cache layout optimization: Prior research has demonstrated very interesting insights about applying PGO on instruction cache layout optimization, especially on manipulating instruction-cache prefetching. By offline post-processing the profiling data, AsmDB [3] is able to build a full control-flow graph and identify two major sources of front-end stalls: *cache fragmentation* and *distant branches and calls*. Based on the results from AsmDB, they proposed a novel compiler optimization that automatically inserts prefetching instructions only into performance-critical execution paths within an application's binary. Leveraging dynamic miss profiles, I-SPY [4] injects *conditional prefetching instructions* to ease the tension between prefetching accuracy and coverage and *coalesces prefetches* to reduce the static code footprints of the injected prefetching instructions. Ripple [5] identifies a *cue block* that has the highest probability to be evicted by the ideal cache replacement policy and injects a "cache line eviction" instruction to the selected spot at link time to mitigate the wasteful evictions.

Since all the above approaches rely on offline profiling, it would be interesting to see how well they cooperate with OCOLUS to perform online prefetching instruction injection. For example, current hardware-based prefetching mechanisms for online workloads are based on inaccurate understanding of program behavior, because most server processors employ only simple next-line prefetchers. However, if we leverage I-SPY to drive *online* analysis of instruction cache miss behavior, and replace the code region with the newly constructed executable that has conditional prefetching instructions injected, we might be able to see a lower instruction-cache miss ratio. So, one of my future research directions will be leveraging these existing offline instruction-cache layout optimization solutions to build a corresponding online version.

References

[1] G. Ayers, J. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory Hierarchy for Web Search", in *High Performance Computer Architecture*, 2018

[2] S. Kanev, J. Darago, K. Hazelwood, and P. Ranganathan, "Profiling a warehouse-scale computer", in *International Symposium on Computer Architecture*, 2015

[3] G. Ayers, N. Nagendra, D. August, H. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley and P. Ranganathan, "AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers", in *International Symposium on Computer Architecture*, 2019

[4] T. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz and B. Kasikci, “I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing”, in *International Symposium on Microarchitecture*, 2020

[5] T. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz and B. Kasikci, “Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications”, in *International Symposium on Computer Architecture*, 2021

[6] M. Panchenko, R. Auler, B. Nell and G. Ottoni, “BOLT: A Practical Binary Optimizer for Data Centers and Beyond”, in *International Conference on Compiler Construction*, 2019