SHORT-PAPER

POSTER: Bring I-Cache to Light in Data Plane Applications

YIHAN DANG, Xi'an Jiaotong University, Xi'an, Shaanxi, China

ZE XIA, Xi'an Jiaotong University, Xi'an, Shaanxi, China

HAO LI, Xi'an Jiaotong University, Xi'an, Shaanxi, China

Open Access Support provided by:

Xi'an Jiaotong University

PDF Download
3672202.3673724.pdf
22 December 2025
Total Citations: 0
Total Downloads: 376

Published: 04 August 2024

Citation in BibTeX format

ACM SIGCOMM Posters and Demos '24: ACM SIGCOMM 2024 Conference: Posters and Demos
August 4 - 8, 2024
NSW, Sydney, Australia

Conference Sponsors:
SIGCOMM

POSTER: Bring I-Cache to Light in Data Plane Applications

Yihan Dang
Xi'an Jiaotong University
Xi'an, China

Ze Xia
Xi'an Jiaotong University
Xi'an, China

Hao Li
Xi'an Jiaotong University
Xi'an, China

Abstract

Modern data plane applications are usually with large code, causing the i-cache to miss frequently during execution. To this end, we apply a new execution model on these applications that splits them into i-cache-friendly stages, and reuse the code of each stage on multiple packets. Experiments on snort3 show that i-cache misses are reduced by 72.2%, and throughput increases by 25.0%.

CCS Concepts

• **Networks → Middle boxes / network appliances**; • **Software and its engineering → Software performance**.

Keywords

Instruction Cache, Data Plane

ACM Reference Format:
Yihan Dang, Ze Xia, and Hao Li. 2024. POSTER: Bring I-Cache to Light in Data Plane Applications. In *ACM SIGCOMM 2024 Conference (ACM SIGCOMM Posters and Demos '24), August 4–8, 2024, Sydney, NSW, Australia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3672202.3673724

1 Introduction

One major hinder to the performance of data plane applications is the cache inefficiency [13]. To this end, researchers has proposed a wide variety of optimizations, targeting on lower cache miss rate in accessing *data*, *e.g.*, packet data [8, 12, 14], flow states [6], and critical data structures [4, 5]. However, data plane applications in production are essentially bottlenecked in accessing *code*. Table 1 profiles three popular data plane applications, where the portion of "Backend Bound", *i.e.*, the amount of time when the processor is waiting for accessing data, is quite limited; instead, the factual bottleneck is the "Frontend Bound", *i.e.*, the CPU stalls during instruction fetching and decoding.

We observe that such inefficiency is mainly caused by excessively large-sized code, which is common in feature-rich data plane applications. As a result, even the execution path of a single packet can cause the i-caches (*e.g.*, L1 i-cache and iTLB) to thrash, causing the processor front-end to stall frequently. Our insight on this problem is to adopt an i-cache-friendly execution model, splitting the execution path into smaller stages, and schedule a batch of packets to finish a stage before moving to the next. In this way, the processor effectively reuses code blocks that can fit in i-cache, and substantially relieves from the front-end bottleneck.

We propose NanoPipeline to achieve the aforementioned goal. By offline profiling the execution path, we can find the appropriate size and location of each stage, and then mark clear boundaries between them. By runtime scheduling the packets, we can make a packet to hand control to other packets when it reaches stage boundaries, preventing cramming too much code into i-cache. Combining both points together, we are able to integrate the i-cache-friendly execution model into the control flow of existing data plane applications. Experiment on snort3 [1] shows that NanoPipeline can reduce the stalls caused by front-end by 53.8%, leading to a 25.0% throughput improvement.

The contribution of this paper is showing that the common front-end bottleneck of popular data plane applications can be mitigated by NanoPipeline's methodology.

Table 1: Execution time breakdown of complex data plane applications reported by Intel vTune.

Program	Frontend Bound	Bad Spec.[a]	Retiring[b]	Backend Bound
Snort3	**50.6%**	0.9%	34.6%	13.9%
Nginx	**61.7%**	0.7%	29.4%	8.2%
HAProxy	**31.1%**	2.6%	37.9%	28.4%

[a]The amount of time spent on wrongly-predicted branches.
[b]The amount of time when the processor is running at full speed.

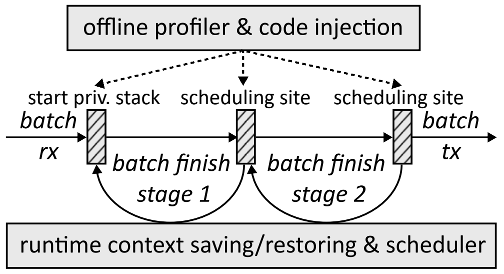

Figure 1: Workflow of NanoPipeline. Dashed arrows signify the offline tasks, and concrete arrows represent runtime application control flow.

2 Initial design

As in Figure 1, NanoPipeline consists of offline and runtime components. The detailed design is as follows:

Offline profiling and scheduling site injection. To decide the location of each processing stage, instruction-level information about an execution path is required. We leverage Intel PT [2] to dump the complete instruction sequence during program execution. Then, we remove any loop presented in the path, ensuring that instructions inside a loop will not be counted multiple times. After

Table 2: Experimental results

Metrics	snort3	snort3-opt	ratio
Frontend stalls (M)	4787	2213	-53.8%
L1 icache misses (M)	176	49	-72.2%
Total micro-ops (M)	3296	3402	3.2%
Throughput (Mbps)	501.9	627.4	25.0%
Frontend Bound	50.6%	28.9%	

that, we can calculate the instruction size as well as the number of cachelines of the path, which serves as the guideline in creating stages. Note that data dependency is also a major factor to consider in choosing stages, see Section 4.

When the decision for stages has been made, we leverage a static binary rewriter to inject a function call as scheduling site at the end of each stage, which transfers the control flow to the runtime scheduler. Additionally, we inject a call to create private stack for each packet before packet processing begins, so that context of a packet can be saved when it yields control.

Runtime scheduling. Despite data applications have different purposes and designs, most of them follow the common pattern of *batch receive*, *process each packet inside a loop*, and *batch send*. This implies that a packet will not be sent out until all packets in its batch are processed, allowing the processor to temporarily suspend it and process other packets instead.

Switching between packets requires saving and restoring contexts when a packet yields and regain control flow. As discussed above, we allocate a private stack for each packet to save its execution context. In this case, switching out from packet processing context is as simple as saving private stack location and callee-saved registers, which is reported to only incur minor overhead [7, 10]. Restoring the context is the exact reverse of saving.

A runtime scheduler is responsible of making decision on which stage and which packet to run next. To avoid disrupting instruction cache, all packets in a batch should finish a stage before a packet touches the next stage. Special cases occur at the beginning and the end of packet processing: the scheduler needs to fetch the next packet in batch when there are still packets not entering the pipeline. This is achieved by jumping to the beginning of processing loop as if current packet were finished, and then the next packet is fetched. Similarly, when the scheduler discovers all packets finishing all stages, it will directly jump out of the loop, conforming to the original control flow when a batch ends.

3 Preliminary Evaluation

We apply NanoPipeline on snort3, one of the most popular and feature-rich open-source IDS, and run it on an Intel® Xeon® Gold 6248R CPU with 32 KB L1 i-cache. To focus the performance on a single execution path, we filter out all HTTP requests from an ISP traffic trace. On our testbed, this path is required to be split into 6 stages to fit in i-cache. We run the original snort3 and the optimized version on the same trace, and report the number of critical hardware counters and end-to-end performance in Table 2.

Experimental results show that NanoPipeline effectively meets its design goal. First, the i-cache-oriented design reduces the time of i-cache misses by 72.2%, and consequently reduces the time the processor is bounded by front-end. Note that the front-end stalls do not decrease proportionally to i-cache misses, because i-cache misses only comprise part of the reason for front-end stalls. Second, context saving and restoring during execution, as well as runtime scheduler logic, lead to a 3.2% increase in the amount of micro-operations (the decoded form of instructions). Finally, throughput observes a 25.0% gain, which indicates that even with additional runtime logic, the benefits of i-cache optimization can still lead to a significant improvement in end-to-end performance. Also note the percentage in improving front-end stalls is larger than that of throughput. This is because NanoPipeline only optimizes the processor front-end, which is the most significant part, but not all of performance hinders.

4 Conclusion and Future Work

We have shown that instruction cache appears as a significant performance bottleneck in feature-rich data plane applications, and by carefully scheduling packets to run in sub-i-cache-sized stages, this bottleneck can be greatly relieved. To make this insight robust, we are currently solving on the following challenges:

Data dependencies. Instruction size is not the only consideration when injecting scheduling sites. Consider the case where a packet reads a global variable in phase 1 and then write to it in phase 2. If we schedule all packets to execute phase 1 first, all packets except the first one will read an outdated value, violating the original semantics. In essence, this is because the access pattern on certain variables is changed when runtime scheduling is enabled. To overcome this challenge, we are currently leveraging whole program dependency analysis [11] to find all variables possibly breaking semantic equivalence, and refining the strategy in injecting scheduling sites. A fallback solution can always function after we find all accesses to these variables. When a packet is going to access variables not up-to-date, the packets prior to it in the batch are executed till completed so that the current packet always sees the updated variables.

Multi-path scheduling. When a batch of packets belong to different execution paths, scheduling policies will have a great impact on performance. For example, scheduling a packet of an unseen path will disrupt the existing content in i-cache, diminishing the benefits of scheduling. Our insight on this challenge is that despite the applications have huge code size, most packets will take a limited number of paths in a specific use case. Therefore, we can focus on the interplay of popular paths, developing the optimal scheduling policy among them, and allow the outlier packets to finish without any scheduling. This minimizes negative effects of multiple paths.

Multi-core scaling. We have discussed how a single core schedules packets, and there remains two options to scale to multiple cores: 1) Every core processes a subset of all flows. 2) Every core processes a subset of all stages. In either case, the scheduler should carefully balance the load among all cores to avoid busy waiting or dropping packets. For the former, some flows should be migrated from heavy-loaded core to light-loaded ones [3]. For the latter, some CPU time on the light-loaded core should be migrated to the tasks on heavy-loaded ones [9].

Acknowledgments

This paper is supported by the National Key Research and Development Program of China (No. 2022YFB2901403) and NSFC (No. 62172323). Hao Li is the corresponding author.

References

[1] 2024. Snort. https://www.snort.org/.

[2] Intel® 64 and IA-32 Architectures Software Developer Manuals. 2024. *Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*.

[3] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. 2019. RSS++: Load and State-Aware Receive Side Scaling. In *ACM CoNEXT*.

[4] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2021. PacketMill: Toward per-Core 100-Gbps Networking. In *ACM ASPLOS*.

[5] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, and Dejan Kostić. 2019. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 8, 17 pages. https://doi.org/10.1145/3302424.3303977

[6] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr., and Dejan Kostić. 2022. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *USENIX NSDI*.

[7] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX ATC*.

[8] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *USENIX NSDI*.

[9] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. 2017. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *ACM SIGCOMM*.

[10] Hao Li, Yihan Dang, Guangda Sun, Guyue Liu, Danfeng Shan, and Peng Zhang. 2023. LemonNFV: Consolidating Heterogeneous Network Functions at Line Speed. In *20th USENIX Symposium on Networked Systems Design and Implementation*.

[11] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. ACM, 265–266.

[12] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *USENIX NSDI*.

[13] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44. https://doi.org/10.1109/ISPASS.2014.6844459

[14] Yifan Yuan, Mohammad Alian, Yipeng Wang, Ren Wang, Ilia Kurakin, Charlie Tai, and Nam Sung Kim. 2021. Don't Forget the I/O When Allocating Your LLC. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 112–125. https://doi.org/10.1109/ISCA52012.2021.00018