USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Mitigating CPU Frontend for Complex Data Plane Applications

Yihan Dang, Hao Li, Ze Xia, Jiajun Luan,
and Peng Zhang, *Xi'an Jiaotong University*

https://www.usenix.org/conference/nsdi26/presentation/dang

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Mitigating CPU Frontend for Complex Data Plane Applications

Yihan Dang, Hao Li, Ze Xia, Jiajun Luan, Peng Zhang

Xi'an Jiaotong University

Abstract

The functionality and requirement of modern networks are becoming increasingly complex, giving rise to Complex Data Plane Applications (CDPA) with rich semantics but often limited performance. However, many existing optimizations fail to improve the performance of CDPAs. This is because CDPAs usually come with excessively large code size, which is often two orders of magnitude larger than today's L1 instruction cache (I-cache) size, causing the CPU to frequently stall on accessing instructions, thus presenting a distinct performance profile that bounds on CPU frontend. This paper proposes NanoPL, an I-cache-friendly new execution model that shuffles the packet processing logic to efficiently mitigate CPU frontend for CDPAs. Stemming from the common execution pattern of CDPAs, NanoPL analyzes its code to ensure semantic consistency after shuffling. By collecting performance profile of CDPAs over underlying traffic, NanoPL partitions CDPAs into execution stages and conducts I-cache-friendly shuffling policy. Experiments show that NanoPL can achieve 17.2%~30.2% higher throughput over real world CDPAs due to the reduction of I-cache misses by up to 86.4%.

1 Introduction

Data plane applications form the backbone of modern communication systems, ranging from simple routing to more complex tasks such as DNS systems, intrusion detection systems (IDS), proxies, and beyond. The latter group, referred to as Complex Data Plane Applications (CDPAs), operates on packets in a much finer and deeper manner, *e.g.*, tracking flow state, reassembling segments, scanning payloads, and matching security rules. While indispensable, the single-core throughput of CDPAs is often less than 1Gbps [4, 38], and even with multi-core scaling [9, 17, 43], they frequently fall short of the performance targets required in realistic deployment scenarios like service gateways and edge proxies.

A natural intuition is that CDPAs are slow simply because they perform more work per packet. However, our measurements show that the workload alone does not explain the slowdown. We compare two simple DPAs with two CDPAs. The simple DPAs are from FastClick [10], including a simple firewall that drops packets by 5-tuple and a simple router that forwards packets by the destination IP. For CDPAs, we involve two typical IDS, snort3 [38] and Suricata [40]. As shown in Table 1, when per-packet instructions increase from 1,252–2,789 in simple DPAs to 11,980–40,670 in CDPAs, the instruction per cycle (IPC) dramatically drops from averagely 2.35 to 1.42. This *superlinear* performance degradation indicates a microarchitectural cause for the low CPU efficiency.

Table 1: IPC drops in a superlinear way in CDPA.

Application	Per-packet instruction	Per-packet cycle	Instruction per cycle
Simple Firewall [10]	1,252	655	**1.91**
Simple Router [10]	2,789	999	**2.79**
snort3 [38]	11,980	7,987	**1.50**
Suricata [40]	40,670	30,350	**1.34**

Given this microarchitectural bottleneck, we ask where the cycles go. CDPAs are complex along two axes: heavy statefulness that inflates data access, and intricate control logic that expands code. Extensive profiling shows that time spent accessing packets and associated state, *i.e.*, CPU *backend*, is small; in the `nginx` proxy, only 8.2% of execution time is spent waiting for data (see Figure 1), implying an Amdahl upper bound of at most 8.9% even if all data accesses hit in the L1 D-cache. In contrast, instruction delivery, *i.e.*, the CPU *frontend*, accounts for the majority of stalls [44]. For example, 67.3% percent of CPU cycles are spent on CPU frontend in `Apache2` web server, giving a 3x speedup if optimized out. Our measurement also confirms this analysis: D-cache optimizations improve the throughput of CDPAs in Table 1 by only about 4%. Overall, low CPU efficiency in CDPAs stems from inefficient instruction delivery, not data-access latency. More evidence is presented in §2.1.

Unfortunately, improving the efficiency of instruction fetching is hard in CDPAs: due to the complex logic, a single packet traverses a long, input-dependent path, so the runtime instruction working set exceeds L1 I-cache and keeps thrash-

ing it. A hardware approach is to design a better prefetcher that more accurately fetch upcoming instructions into the I-cache [19–21], but these require costly CPU modifications. Profiling-guided optimization (PGO) that reorganizes the code layout can reduce the conflict misses in the I-cache and I-TLB [31,35], while the long code path would still cause inevitable capacity misses.

Our insight is to pursue *reusing*, rather than *selection* or *re-layout*: instead of trying to control which instructions are loaded, we exploit those already resident and maximize their reusing. We partition packet processing into small, independent stages that each fits within the I-cache, and schedule multiple packets to execute the same stage back-to-back. In other words, such time-domain reusing creates several *pipelines* on a single core, which shrinks the effective instruction working set, reduces I-cache misses, and raises IPC.

Building on this insight, we propose *NanoPipeLine (NanoPL)*, a framework designed to mitigate frontend bound in CDPAs. Concretely, we make the following contributions:

- Proposing an I-cache oriented execution model that partitions application logic into smaller stages, which largely reduces CPU frontend bound caused by excessive I-cache misses;
- Designing the semantics and performance policy to partition program stages, so that the CDPAs can run efficiently as well as sticking their original behavior;
- Implementing the prototype of NanoPL, which is able to be integrated into various real-world CDPAs. Experiments show that NanoPL can increase end-to-end throughput by up to 30.2% and reduce I-cache misses by up to 86.4% on different CDPAs.

2 Background and Motivation

In this section, we first profile various complex data plane applications to show their frontend bottleneck, and reveal that I-cache miss is the dominant factor of it (§2.1). Then, we discuss the existing solutions to mitigate the I-cache misses, and explain why they fail to fully address this problem (§2.2).

2.1 Frontend as the Bottleneck

Frontend is the performance bottleneck of CDPA. Modern processors mainly experience three major parts of performance penalty [44]: (1) *frontend bound*, where CPU stalls when the *instruction* is unavailable, (2) *backend bound*, where CPU stalls when the *data* is unavailable, and (3) *bad speculation*, where CPU mispredicts a *branch* and has to discard the speculatively executed instructions. Obviously, the performance bottleneck will vary largely since applications have distinct instruction, data and branch working sets. However, CDPAs present a surprisingly similar performance profile, where frontend bound appears as the largest performance bottleneck. As shown in Figure 1, we profiled 8 well-known CDPAs under traces captured from our campus network (detailed in §7.1), calculating the CPU time portion of inefficiency, and the time CPU is running without stalling, *i.e.*, *retiring* micro-operations. Among all tested applications, frontend bound appears as the largest performance penalty, ranging from 29.6% to 67.3%. In contrast, bad speculation takes up 0.8%–17.6% of stalls, and bad speculation contributes to 8.2%–30.0%, which is 32× and 3.5× less than frontend bound in average.

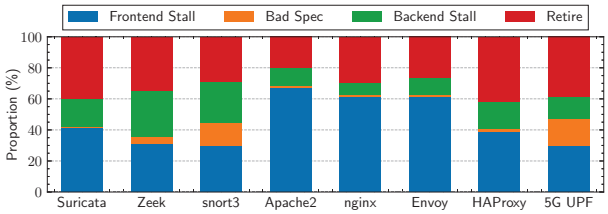


Figure 1: Runtime CPU time breakdown of well-known data plane applications under representative configurations. Frontend bound (45.1% in average) dominates the CPU time.

Optimizing backend is not effective for CDPA. Such frontend dominant profiles will sometimes affect or even nullify the effectiveness of existing optimizations on network systems that target on different bottlenecks. For example, Reframer [15] proves that the performance of network functions can be significantly improved by buffering and aggregating same-flow packets, because their major performance bottleneck is on per-packet flow state accesses. To show the optimization effects on different applications, we apply Reframer to the simple firewall and the simple router in Table 1, and observe that the end-to-end throughput gains are 18% and 61%. However, with the same packet buffering time, Reframer can only obtain 2.5% and 4.9% improvement on the two CDPAs listed in Table 1, because their bottleneck is on instruction instead of data access.

I-cache miss is the root cause. Frontend bound happens when instructions are not readily available from memory. This could be caused by several reasons, *e.g.*, I-cache misses, iTLB misses and branch misprediction correction. To further find out the root cause of the frontend bottleneck, we conduct a breakdown analysis using the method suggested in [5]. The results show that in 7 out of 8 applications we profiled, I-cache misses is the largest contributing factor of frontend bound, which incurs 1.8× and 6.1× more stalls than iTLB misses and branch corrections. The highlights the needs to design optimizations targeting on reducing I-cache misses.

2.2 I-Cache Miss as a Pending Problem

Existing works tackle the I-cache miss problem from three perspectives: (1) improving the instruction fetching policy to *refine* the working set, (2) optimize the binary to *rearrange* the working set, or (3) using modular unit to *reimplement* processing logic as well as the working set.

Improving prefetcher precision. Modern x86 processors are usually equipped with a hardware instruction prefetcher [20, 33], which looks several cachelines ahead of the current control flow [46] and fetches the upcoming instructions when the processor backend is finishing calculations required by the current instruction. This works perfectly until there are branches in the control flow, which requires the branch predictor to decide the prefetching location. However, a mistake of the branch predictor will fetch useless instructions into I-cache, wasting CPU time as well as polluting the I-cache. To solve this problem, researchers focusing on CPU microarchitecture have proposed various hardware improvements to predict branches more precisely [20, 34]. Yet iterations over CPU microarchitecture not only takes year to finish, but also pose a heavy burden to enterprise users. As a result, it is impractical to leverage such method to optimize CDPAs.

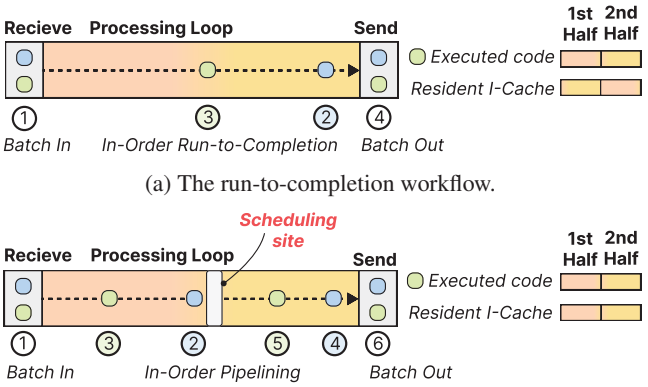
Profiling-guided optimizations (PGO). After collecting runtime statistics, PGO can rearrange branches and functions to optimize application binaries [28, 30]. This rearrangement aggregates common instructions and modules inside the binary, which efficiently utilize the I-cache lines and instruction memory pages. However, since such optimizations must not change the original application control flow, they still fails to prevent the excessively long execution paths from exhausting and thrashing hardware resources. Our experiments show that such rearrangement can only deliver minor improvement and sometimes even harm the performance (see §7.2).

Modular processing units. Many works have explored providing common building blocks to compose data plane applications [6, 10, 16]. This allows applications to schedule their processing logic flexibly by leveraging the modular processing units, therefore improving cache utilization. VPP [6] is the representative work of this kind. By allowing a batch of packets going through a processing node at a time, instruction working set is effectively limited to the size of that node. However, such works force the applications to be implemented and configured using their own features and syntax, which is at odds with the existing huge code base and custom semantics of CDPAs.

Takeaway. For CDPAs, their instruction working set cannot be refined because of the high cost of upgrading CPUs, cannot be rearranged due to its irreplaceable core logic, nor can be reimplemented as a result of huge porting efforts. In conclusion, existing work cannot fill the gap between CDPA code and CPU I-cache.

3 NanoPL Overview

As shown by existing work in §2.2, the key to mitigate frontend bound is to reuse existing contents in the I-cache, which is too hard if treating CDPAs as opaque boxes. However, we argue that the control flow of CDPA has a large potential of reusing instructions and thus preventing I-cache misses.



(a) The run-to-completion workflow.

(b) NanoPL partitions the workflow into stages.

Figure 2: The workflow of NanoPL. The gradient colored rectangle illustrates a processing loop larger than the I-cache capacity, and blue and green rounded rectangles are two packets received in order. (a) The original workflow runs each packet to the end before processing the next. Then the green packet continually thrashes I-cache (right side). (b) NanoPL partitions the workflow into two stages with a *scheduling site*. When the blue packet finishes the first half, NanoPL would suspend it and push the green one to the same code. This lets the green packet always reuse the resident I-cache (right side).

When multiple packets share part of the processing logic, we can achieve code reusing by processing them consecutively. In this way, we effectively *redefine* the working set by identifying and enabling potential code reusing.

In this section, we illustrate a workflow on how NanoPL achieves the code reuse (§3.1), and pose the two key challenges behind this seemingly simple notion (§3.2).

3.1 Workflow of NanoPL

Many packet-processing loops are larger than the I-cache, so a worker that executes packets in a run-to-completion style repeatedly shifts its instruction working set. Figure 2a follows two packets that arrive in order. The first packet (blue) enters the loop, executes the first half, then progresses into the second half. When the next packet (green) starts, it re-enters the first half while the cache still holds code from the second half. The I-cache must reload the first-half instructions, and when the green packet resumes the second half it reloads those routines again. This back-and-forth refill pattern leaves adjacent time slices executing different code regions, which prevents a stable resident set from forming. The result is persistent I-cache misses, wasted front-end bandwidth, and a throughput loss that grows with loop size and control-flow variation.

Such execution pattern is widely adopted by CDPAs, and serves as a clear boundary of each packet's processing, enabling domain specific optimizations. Based on that, NanoPL introduces a scheduling mechanism that reshapes the batch-

processing loop so that adjacent time slices execute the same code region. The key idea is to identify the processing paths of each packet, partition them into stages whose instruction footprints fit in the I-cache, and insert scheduling sites that suspend and resume packet execution at stage boundaries.

Batch processing in a pipeline manner. We first locate the batch-processing loop and the per-packet execution paths within it, which gives a precise boundary for each packet's tasks and a natural place to apply code reuse. Using these boundaries, NanoPL partitions the loop body into stages smaller than the I-cache. Packets within the same batch then traverse the stages in order, but at any moment the worker runs many packets on the same stage, allowing the current I-cache contents to be reused across packets.

Suspending packets with scheduling site. The key enabler of NanoPL is the *scheduling site*, which is able to suspend the processing of the current packet, and allows the CPU to process the next for code reuse. As shown in Figure 2b, a scheduling site cuts the loop into two stages. When the blue packet reaches the site (②), the next instruction after it is saved for the blue packet's continuation. The control flow is then redirected to the loop head to fetch the green packet and execute the same first-half code. The worker repeats first-half for multiple packets in the batch (③). During this process, only the red region of code is touched, remains resident in I-cache and is being reused by all packets in the batch. When the batch is small or with latency-sensitive packets, scheduling sites are temporarily disabled and the packets will follow their original workflow.

Resuming packet from where it suspends. Resumption is symmetric. After a batch of packets completes a stage, previously suspended packets resume from the saved address as if returning from a function, so their control flow and per-packet execution paths are preserved. In Figure 2b, after the batch finishes the first-half, the two packets resume execution at their saved addresses to run second-half (④–⑤). This in-order pipelining does not change external semantics: packets still enter and leave the loop in order, while within each stage the instruction working set remains stable. During this stage, only the yellow region is touched and remains resident in I-cache.

In general, with NanoPL, I-cache content changes only at stage transitions, not at every packet boundary, which avoids the back-and-forth refills that thrash the I-cache and shifts the bottleneck away from the fetch front end.

3.2 Ongoing Challenges

The above workflow shows that partitioning code into stages has the potential to reuse instructions already in I-cache. However, arbitrarily placing scheduling sites would raise correctness issue and/or degrade expected improvements. To this end, NanoPL must address the following two challenges.

Semantics consistency. NanoPL guarantees to resume execution of each packet from where it is suspended; however, the scheduling sites could still violate the original semantics. This is because the "out-of-order" execution of an packet could overwrite the variables of others. For example, if a TCP FIN packet is reordered to be processed earlier, it may free flow states and invalidate the pointers held by subsequent packets. How to precisely define and enforce semantic consistency of the application, while enforcing the efficient execution model, is a critical challenge of NanoPL.

I-cache reusability. The workflow in Figure 2 considers an ideal case, where all packets take the identical execution path, perfectly sharing the instruction of each stage one after another. In fact, given a wide range of input packets, they will have drastically different paths. This not only increases the difficulty of aggregating packets of the same paths, but also complicates the process to make runtime scheduling decisions. How to ensure code reusing even for packets taking various paths, is a key challenge of NanoPL.

4 Enforcing Semantics Consistency

In this section, we formally define the execution consistency through the instruction dependency (§4.1). The key of preserving such consistency is to avoid variable conflicts (§4.2), and we present how NanoPL safely composes the scheduling sites respecting those constraints (§4.3).

4.1 Consistency via Dependency Preservation

A straightforward baseline for semantic consistency is to preserve the exact order of execution over all packets. However, just as CPUs reorder independent instructions, NanoPL can still be semantically correct while rescheduling execution, as long as it only reorders operations that are *independent*.

In the following, we formally define events, the dependency relation, and a dependency-preserving consistency criterion for NanoPL.

Events and executions. The original execution induces a reference total order $(E_{\text{ref}}, \leq_{\text{ref}})$ over an event set E. Each event is

$$e = (\text{op}, v, \text{pkt}),$$

where $\text{op} \in \{\text{R}, \text{W}\}$ (read, write), v is a program variable, and pkt is the triggering packet. Since NanoPL does not create or delete any event, it actually uses the same event set E but produces a partial order $\leq'$.

Dependencies. We use a dependency relation $\rightarrow_D \subseteq E \times E$ to capture what must not be reordered.

(D1) *Program order:* For two events e_1 and e_2 from the same packet, if e_1 precedes e_2 in the reference, e_2 depends on e_1.

(D2) *Variable conflicts:* For two events e_1 and e_2 that access the same variable v, with at least one being a write, if e_1 precedes e_2 in the reference, e_2 depends on e_1.

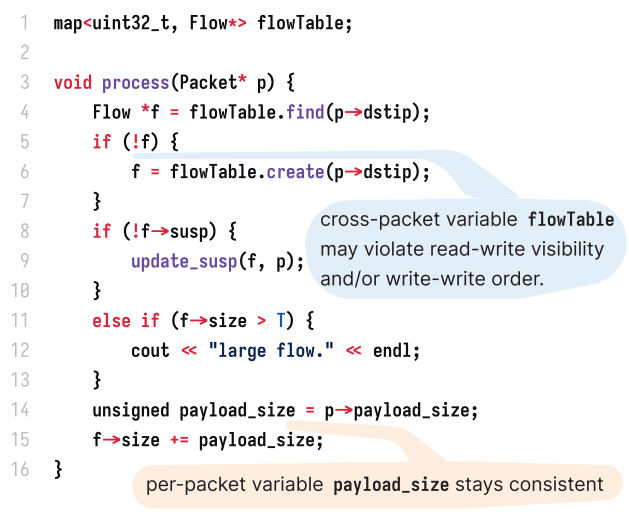


Figure 3: Example DPA code with two candidate scheduling sites. The red site that separates the read-write to a per-packet variable is safe, while the blue one that splits accesses of a cross-packet variable may cause inconsistency.

Intuitively, $\rightarrow_D$ is the must-happen-before relation induced by code order and per-variable conflicts.

Definition 1 (Dependency-preserving consistency)**.** *A NanoPL execution $(E, \leq')$ is consistent with the reference if it preserves all dependency relations ($e_1 \rightarrow_D e_2 \Rightarrow e_1 \leq' e_2$).*

Note that NanoPL does not modify the event set and always resumes packets exactly where they are suspended. This means the program order for any certain packet inherently preserves, *i.e.*, NanoPL ensures (D1) by design. As a result, the following focuses on the variable conflicts (D2).

4.2 Variable Conflicts

Consider a simple DPA example shown in Figure 3: for each incoming packet, it executes `process()` (Line 3), which looks up (or creates) the corresponding flow by destination IP (Line 4–7), checks the suspicious payload (Line 8–13), and accumulates the payload size into the flow record (Line 11–15). In this code, temporaries such as `payload_size` belong to the current packet only (*per-packet variables*), while `flowTable`, `f->susp`, and `f->size` are shared across packets (*cross-packet variables*).

Per-packet variables. Per-packet variables are visible only to the current packet. Because other packets never access them, they do not create (D2) conflicts. For example, let NanoPL places a scheduling site between Line 14–15 (red site), and two packets p_1 and p_2 are suspended after Line 14. When resuming p_1, it reads and writes its own `payload_size`, not that of p_2. To this end, we call the red site is *safe*.

The remaining risk is stack pollution when interleaved packets reuse the same stack frames. We avoid this with a simple "copying" scheme: NanoPL gives each in-flight packet a private stack, switches the stack pointer to that stack on entry to `process()`, and on a context switch saves callee-saved registers and restores the destination packet's stack pointer and saved registers. Private stacks are recycled between batches. With this isolation, per-packet variables remain conflict-free.

Cross-packet variables. Cross-packet variables can be read or written by different packets in an interleaved manner. Unlike per-packet variables, these states cannot be protected by per-packet isolation because they are inherently shared. As a result, NanoPL must preserve write–write (WW) order and read-write (RW) visibility: the relative order of writes to a variable v must match the reference, and each read in $\leq'$ must see the latest preceding write.

For example, suppose NanoPL places a scheduling site after Line 5 (blue site). Let a packet p_1 misses the lookup at Line 4 and is suspended at the scheduling site. Then, p_2 that has the same destination IP would fail to observe the write that p_1 would perform at Line 6, and also miss at Line 4, violating WR. When both packets later resume from Line 5, they would create duplicated entry, violating WW. As a result, the blue site is not a safe scheduling site.

4.3 Cross-Packet Variable Consistency

We first identify all accesses to cross-packet variables and then compose safe scheduling sites to enforce RW and WW.

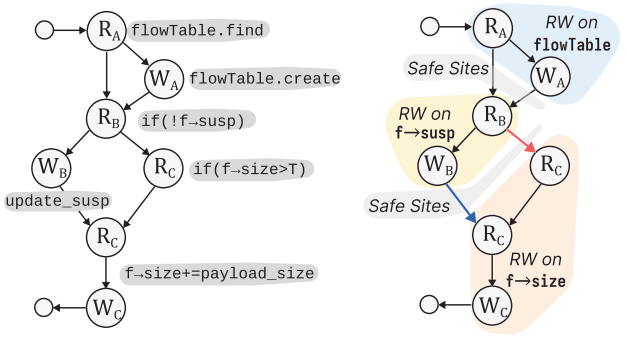
Identifying cross-packet accesses. We enumerate all cross-packet variables, and all instructions that may read from or write to those variables.

Cross-packet variable is any variable whose lifetime spans multiple packets and remains reachable from subsequent packet executions. The classification is simple: variables defined outside the processing loop, *e.g.*, on the heap or in `.data` section, but used inside the loop, are cross-packet variables.

To enumerate accesses, we leverage the classic alias analysis techniques. Concretely, given a cross-packet variable, the analysis process traces its value-flow through all program assignments, and computes a sound set of pointers that may be an alias to that cross-packet variable. Pointers allocated on the stack but holding reference to cross-packet variables are also identified during this process. We then collect all instructions that take those pointers as operands. Those instructions are identified as accesses to this cross-packet variable, and we repeat this process for all cross-packet variables.

Safe scheduling over control-flow graph. Cross-packet interleavings are dynamic, but the program path is static. We convert the dynamic consistency requirement into a local path property over a static control-flow graph (CFG) (See the dynamic case in §8.2). This lets us decide safe scheduling sites over the edges on the graph.

Each node in CFG represents an instruction in the processing loop. Specifically, we use op_v to denote a static access to a cross-packet variable, where $\text{op} \in \{\text{R}, \text{W}\}$, and v is a cross-



(a) Control-flow graph. (b) Commitment windows.

Figure 4: Avoid variable conflicts on cross-packet variables. (a) The CFG for code in Figure 3, where A, B, C denote the cross-packet variable `flowTable`, `f->susp` and `f->size`, respectively. Small circles are entry and exit nodes. (b) Marking per-variable commitment windows by RW visibility and WW order. Safe sites must be a cut in-between windows.

packet variable. Figure 4a shows the CFG for code in Figure 3, where A, B, C denote `flowTable`, `f->susp` and `f->size`, respectively. Note that Line 15 both reads and writes `f->size`, so it translates to two nodes in the graph (the bottom R_C and W_C). For simplicity, we exclude irrelevant instructions, *e.g.*, accesses to per-packet variables and I/O operations.

To ensure (D2) on the CFG, we mark a *commitment window* for each variable. A commitment window is a contiguous path interval bounded by two static accesses such that any scheduling between them may violate RW visibility or WW order. Specifically, we mark $[W_1..W_2]$ for any required write order $W_1(v) \Rightarrow W_2(v)$ (WW order), and $[R_v..W_v]$ for $R(v) \Rightarrow W(v)$ (RW visibility). Figure 4b shows the marked commitment windows, where the three windows are for three cross-packet variables respectively. Note that for `f->size`, the window is marked from the earliest read to the latest write.

Finally we define a safe scheduling site over the commitment windows. A scheduling site is a cut to the graph, and is safe if and only if: (1) it lies outside all commitment windows; (2) it separates the entry and exit node. The second is to ensure the cross-window WW order and RW visibility. For example in Figure 4b, consider two consecutive packets p_1 and p_2, which follows the red and blue paths respectively. If NanoPL sets a scheduling site only on the red path, then p_1 would be suspended before R_C, while p_2 is allowed to trigger the bottom W_C. Then the two writes on `f->size` are inverted, and hence inconsistent. Note that this rule also requires the resumed packets must traverse the scheduling site in the exact order they are suspended, *e.g.*, p_1 on the red path must be resumed before p_2 suspended at the blue path.

5 Maximizing I-Cache Reuse

In this section, we first pose the challenge of locating scheduling sites that deliver optimal performance (§5.1). After presenting our basic idea, we deeply describe how NanoPL fits each stage into I-cache (§5.2).

5.1 Challenges and Basic Idea

The intuition is that if the code executed between two scheduling sites can be largely retained in the L1 I-cache, packets in the same batch will repeatedly traverse the same instructions, which reduces the I-cache misses. Our goal is to select a set of scheduling sites that partition the code into stages, each of which fits in the I-cache.

One concern is that the commitment window may already exceed the I-cache capacity, making it impossible for any single scheduling site to enforce I-cache residency. Fortunately, we observe that complexity in CDPAs often stems from the composition of multiple modules rather than a single monolithic path, and the cross-packet variables are mostly independent for different modules. For example, an IDS comprises parsers across L2–L7, each maintaining independent cross-packet variables such as a TCP flow table or an HTTP session table. This suggests that the extremely large commitment window may be a rare case in practice. Our experiments show that even the largest commitment window spans only 14.8%–34.6% L1 I-cache capacity (see §7.3 for details).

Even when commitment windows are within the I-cache, deciding the location of scheduling sites remains difficult. A naive policy is to ensure instructions between scheduling sites fitting into I-cache. However, this will create a large number of scheduling sites because the union of all possible execution paths is large. Under such policy, the control flow will frequently switch between packets, increasing scheduling overhead and D-cache misses (detailed in §7.3).

Our key observation is that CDPA control flow is highly skewed. A few paths implement the mainline logic with compact code such as fixed-format IP parsing or continuous TCP segmentation. Other paths are heavy but rare and handle exceptional cases such as unusual IP options or TCP out-of-order reassembly. Paths therefore contribute unevenly to the I-cache footprint. In short, we should partition according to runtime contribution rather than static size. This avoids overpaying to cover rare paths and reduces the number of per-packet scheduling events.

5.2 Fitting Stages into I-Cache

We next describe how NanoPL (1) collects runtime information, (2) assigns weights to basic blocks, and (3) places scheduling sites accordingly.

Collecting runtime information. We rely on offline replay with sampled traffic. The mainline logic of CDPA is relatively stable across deployments, so profiling on representative samples is sufficient to capture execution characteristics. NanoPL injects sampled traffic into the target CDPA and records the

executed path of each packet by instrumenting basic blocks in the IR and logging the block sequence along the path.

Recall that commitment windows cannot be separated by scheduling site, so when we merge blocks belonging to the same commitment windows. This allows us to locate safe scheduling sites by separating the blocks. Specifically, for each basic block B, NanoPL records:

- Visits(B): the number of times execution enters B.
- $U_B^{(v)}$ for sampled visits v: the number of unique I-cache lines touched within B during visit v.
- InstrExec(B): the total dynamic instructions spent in B.

Weighting the block. We focus on two per-block quantities: the per-block cost, *i.e.*, how much I-cache would be occupied for this block; and the per-block heat, *i.e.*, how much workload could be covered for this block.

Per-block cost. We define a block's cost as the typical I-cache footprint according to the collected information.

$$s(B) \;=\; \mathbb{E}[U_B] \;\approx\; \frac{1}{\mathrm{Visits}(B)} \sum_{v=1}^{\mathrm{Visits}(B)} U_B^{(v)}$$

Intuitively, $s(B)$ reflects how much cache space B typically needs when it is active. This choice avoids over-provisioning for rare heavy paths, helping us pack more blocks and reduce scheduling sites.

Per-block heat. We define a block's *heat* as its expected performance cost per request. Heat represents how much benefit we gain by keeping B resident, and it is the counterpart to $s(B)$ in the budgeting objective.

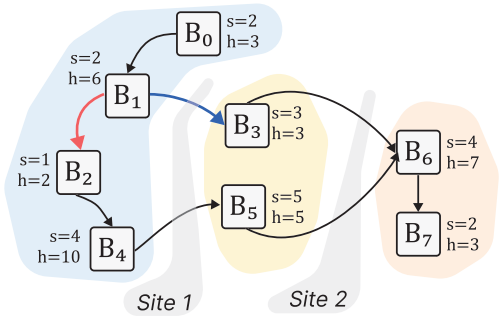
$$h(B) \;=\; \frac{\mathrm{InstrExec}(B)}{N},$$

where N is the total number of replayed packets. Intuitively, $h(B)$ is the expected instructions spent in B per request, capturing how "valuable" it is to allocate cache budget to B. The "hot" blocks are more likely executing the mainline logic, and thus should be included in the current stage. In this way, the number of scheduling sites through the mainline logic can be decreased.

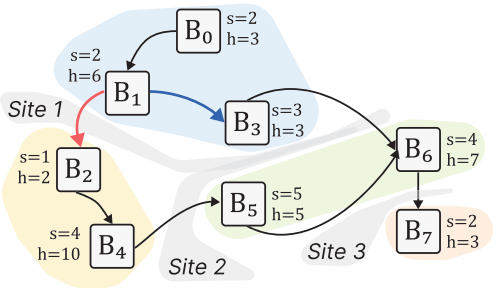
Partitioning according to the weights. Given $\{s(B), h(B)\}$, NanoPL places scheduling sites by searching for a good cut and then committing everything before it. A cut separates the current head from the rest of the CFG. The stage cost is the sum of $s(\cdot)$ over blocks before the cut, and the stage benefit is the sum of $h(\cdot)$.

The search is lightweight with look-ahead. We start with the cut right after the head, and push them forward along likely branches. Each push moves the cut past one edge, and will fallback if the cut is not safe. To guide the search, we score each cut with a simple optimistic bound

$$\mathrm{UB}(F) = H(F) + \big(Bud - S(F)\big)_+ \rho_{\max}, \qquad \rho_{\max} = \max_B \frac{h(B)}{s(B)},$$



(a) Optimistic-bound cut.



(b) Greedy cut.

Figure 5: Finding optimal cuts on CFG with I-cache capacity. (a) The two sites partition the CFG into three stages. When pushing the frontier at B_1, NanoPL would choose the red path (to B_2) rather than the blue path (to B_3), because the red one has higher optimistic bound for including the hot block B_4. (b) If pushing frontier with greedy strategy over $h(B)$, it may compose more stages, and thus more runtime overhead.

where $S(F)$ and $H(F)$ are the current cost and benefit under cut F, Bud is the remaining cache budget, and $(x)_+ = \max(x, 0)$. The intuition is to take the already gained $H(F)$ and pretend the remaining budget is filled with the best observed benefit-per-cost $\rho_{\max}$; this overestimates what is achievable, so it is safe for ranking.

We always advance the cuts with higher scores. When a cut is nearly full or its bound barely exceeds its current benefit, we record it as a candidate. Among feasible candidates (within budget), we pick the one with the highest benefit, place the site, move the head to that cut, and repeat.

A walkthrough example. We use the CFG in Figure 5a to illustrate how NanoPL advances the frontier and places a scheduling site under a strict I-cache budget $C = 9$. Starting at entry we include B_0 and B_1, yielding $S = 4$ and $H = 9$. At the split B_1 we rank candidate advances with the optimistic bound. The blue branch would add B_3 ($s = 3$, $h = 3$), producing $S = 7$ and $H = 12$ with only two units of capacity left and no access to a high-payoff block before the next site, so its bound is weak. The red branch adds B_2 ($s = 1$, $h = 2$), then exposes B_4 ($s = 4$, $h = 10$); taking B_4 uses the remaining budget to reach $S = 9$ and $H = 21$.

To this end, although $h(B_2) < h(B_3)$, the red direction at-

tains a higher bound because the leftover capacity can be converted into heat near ρ_{max} by capturing B_4, whereas the blue path saturates early without it. We therefore advance on red to just before B_5 and place the first site at B_1–B_3 and B_4–B_5, yielding the segment $\{B_0, B_1, B_2, B_4\}$ with $S = 9$. We apply the same rule toward Site 2, ultimately partitioning the CFG into three stages that each fit the I-cache budget and preserve high payoff.

We finally show the scheduling sites if we solely use $h(B)$ to push the frontier, *i.e.*, pushing the blue path instead of the red, in Figure 5b. In this case, we have four stages, and the mainline logic, *i.e.*, the paths with higher H ($B_0 \to B_1 \to B_2 \to B_4 \to B_5 \to B_6 \to B_7$) will meet three scheduling sites instead of two, which results in more runtime overhead.

6 Implementation

We implement NanoPL in roughly 5.3K lines of C++ code under LLVM [22] v14.0.0, including a profiling pass instrumenting program IR to find and analyze instruction traces during packet processing, an analysis pass that finds the correct and performant location in IR for scheduling sites, an injection pass that integrate scheduling sites into program IR, and a runtime scheduler responsible for managing the control flow. We highlight several key workflows involving intricate collaboration of these modules.

Scheduling site analysis. To achieve sematic consistency for cross-packet variables in §4.3, NanoPL has to perform static program analysis on the whole program. This is done by first compiling the program into a whole-program LLVM IR using wllvm [32]. Then, NanoPL leverages SVF 2.7 [41] to construct a complete inter-procedural control flow graph (ICFG), and labels the packet receiving interfaces. In our prototype implementation, we support packet receiving interfaces including kernel event-based APIs (*e.g.*, `epoll` and `poll`), Intel DPDK [2], and libDAQ [39] used by snort3. Calls to these interfaces help the analysis pass find the packet processing entrance function, which serves as a separation mark in the path profiling results, as well as the starting point of static analysis. After collecting the initial set of cross-flow variables from the program IR, NanoPL adopts flow sensitive analysis [7, 8] to find all their aliases, traversing the ICFG to mark the edges illegal for scheduling sites, and uses the results as output in injection pass.

Scheduling site injection. To precisely get I-cache usage during runtime, we use a modified version of Cachegrind [1] to dump the instruction sequence in runtime. Then, stages are partitioned according to I-cache capacity (set index of each cacheline is not decided yet). The scheduling site location found is the address in binary text section, and is converted into LLVM IR location by llvm-addr2line [3]. After that, the injection pass injects stub function calls to the designated location, and the implementation of scheduling sites and runtime scheduler is linked against the instrumented IR to form the optimized binary. During runtime, the control flow sticks with the original behaviors until it reaches the stubs, and switches to the scheduler. Then, the scheduler picks the next packet from the scheduling site with the most packets, and finally hands the control back to application.

7 Evaluation

In this section, we evaluate the performance benefits by deploying NanoPL in real-world applications. In particular, we are interested answering in the following questions:

(1) Can NanoPL be integrated into various CDPAs correctly as well as yielding a decent end-to-end performance boost? Experiment results show that NanoPL is able to achieve a 17.2%–30.2% throughput improvement on tested applications (§7.2), while consistently stick to their original semantics.

(2) Where does the performance gain of NanoPL come from and how does it change with different parameters selection? Experiment results prove that NanoPL improves performance by reusing instructions with the same batch, and the importance of scheduling site selection (§7.3).

7.1 Experimental Setup

Testbed. We test NanoPL on an Intel Xeon Gold 6248R CPU, running at 3.0 GHz with hyperthreading and turbo boost off. This CPU has 32KiB L1 instruction cache on each core, and 35.75 MiB LLC on each socket. The device under test is installed with Ubuntu 20.04.1, using kernel version 5.15.0, and all hardware counters are recorded using perf 5.15.168. For network interfaces, the server is equipped with an Intel 82599ES 10 GbE NIC, which is connected back-to-back with another identical NIC on a dedicated packet generator server.

Applications. We involve four CDPAs for our evaluation, including two IDSes (snort2 and snort3), a web proxy (nginx) and a network function with complex stacks (5G UPF). For IDSes, we use the ruleset shipped each applications, containing rule bindings on a wide range of ports and application layer protocols including HTTP, DNS and FTP. snort3 is selected to show the effect of significantly larger code and less predictable control flow caused by C++. For nginx, we configure it to run as an `epoll`-based HTTP proxy. For 5G UPF, we collect the core modules from the Aether SD-Core UPF [29], and compose a network function based on the protocol stacks generated by Rubik [23], which receives packets from the core network, enforces fine-grained QoS control, and encapsulates them for the mobile network.

Workload. For packet level workload, we use a trace captured from our school network. This trace is composed of a wide range of application layer protocols (22 identified in total,

Table 2: The end-to-end profiling and top-down analysis for vanilla version and NanoPL.

	I-cache MPKI			IPC			Frontend (%)		Retire (%)		Bad Spec (%)		Backend (%)	
	Vanilla	NanoPL	Ratio	Vanilla	NanoPL	Ratio	Vanilla	NanoPL	Vanilla	NanoPL	Vanilla	NanoPL	Vanilla	NanoPL
snort2	33.9	4.60	-86.4%	1.40	1.76	+25.4%	16.9	13.4	34.6	43.4	15.0	5.56	33.6	37.7
snort3	20.2	4.09	-79.7%	1.27	1.58	+24.9%	31.4	22.7	28.9	39.7	14.3	11.1	25.4	26.6
nginx	83.1	48.0	-42.2%	1.54	1.77	+14.9%	56.8	46.0	37.2	42.4	0.90	3.60	5.10	8.00
5G UPF	73.9	37.5	-49.3%	1.61	1.92	+19.1%	29.9	11.8	38.8	46.3	17.7	7.14	13.7	34.7

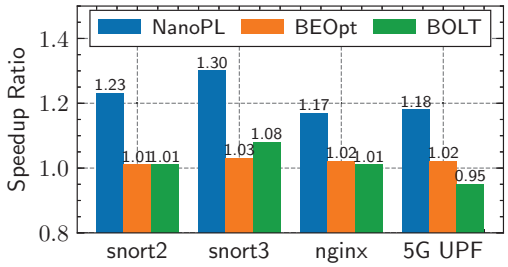


Figure 6: Throughput speedup of different approaches.

including HTTP, SSH, etc.), with its majority traffic type being TCP. In total, this trace contains 11.5M packet, and the average packet size is 666 bytes. All sensitive information is redacted before this trace is given to us. The traffic generator replays this trace at the highest rate without packet loss to ensure identical flow level workloads under various processing speed. For socket level workload, we use wrk2 [42] to generate HTTP requests from 64 concurrent clients accessing random 4KB files, and configure a web proxy to forward the requests to a backend server running on the same machine. To precisely measure the performance profile of applications in the user space, we implement a preload library to proxy socket-related system calls, offloading these calls to non-application threads, so that the applications immediately get return value without stepping into the kernel.

7.2 End-to-End Results

NanoPL improves the end-to-end performance. For NanoPL, we select the scheduling sites based on the policy proposed in §5, and set maximum batch size to 32. Here, NanoPL does not delay packets to make a larger batch, so the per-batch latency is not worsened. Figure 6 shows that NanoPL achieves the largest speedups in all approaches, ranging from 17.2% to 30.2%, and it particularly suitable for feature-rich and compute-intensive IDSes.

We implement a backend optimizer, namely BEOpt, following the basic idea of Reframer [15], which aggregates the data accesses to the same or near memory to reduce the D-cache misses. For packet-based CDPAs, *e.g.*, snort2, snort3 and 5G UPF, BEOpt buffers the packets and flushes the packets from the same flow to the CDPAs; for flow-based CDPAs like nginx proxy, BEOpt aggregates the flow events from the same connection, so as to improve the D-cache hit rate on the flow level. As shown in Figure 6, BEOpt can only achieve 1.7% speedup in average. This echoes the top-down analysis shown in Figure 1, where the backend bound is not the micro-architecture bottleneck.

We use BOLT [31] as the representative PGO approach. We inject 20% packets of the trace into BOLT for training a better layout, and use the rest for measurement. As shown in Figure 6, the overall improvement is minor, ranging from -5.5% to +7.8%. We note that PGO may even harm the performance instead of improving it (5G UPF). This is because the profile data used for PGO may not accurately represent real-world workloads (even they are from the same trace) [26].

NanoPL's gain comes from mitigating the frontend. We collect runtime counters to calculate CPU time breakdown before and after applying NanoPL, as shown in Table 2.

NanoPL largely reduces I-cache MPKI (misses per kilo-instructions) of all involved CDPAs (averagely 64.4%), especially for two IDSes (averagely 83.0%). This is because the packet processing path of IDS systems is particularly long due to the wide range of protocol parsers and the corresponding actions. The mitigation of CPU frontend also improves the IPC of CDPAs with significant ratio, ranging from 14.9% to 25.4%, meaning that the micro-architectural bottleneck has been effectively addressed.

We then deeply dive into these results with a top-down analysis. NanoPL has mitigated the frontend bound of all CDPAs. This enables stalls in CPU pipeline being filled with useful micro-operations, and thus increase the portion of retiring (Retire in the table). NanoPL also helps to reduce bad speculation (Bad Spec in the table) in most cases, because a smaller instruction working set forms a more predictable branch history, improving the prefetching precision [45, 46].

Note that the percentage of frontend bound does not seem to reduce largely compared with the throughput increase. This is because NanoPL reduces the total number of cycles per packet, making directly comparing percentages meaningless. In fact, the actual hardware event counter indicating frontend bound (`idq_uops_not_delivered.core` on our CPU) decreases much more significantly, *e.g.*, 48.0% for snort3. This indicates that we have eliminated a decent portion of frontend

bubbles even under complex path conditions.

Another observation is that the backend bound increases. The reason is two-fold. First, the mitigation of frontend stalls reduces the processing cycles for each packet, making the backend stalls take up a larger portion. For example, the backend stall slots of snort2 and snort3 are actually 9.4% and 14.1% less after applying NanoPL, but the backend bound percentage appears larger. Second, NanoPL schedules multiple packets at the same time, which can increase D-cache misses. This is because more per-packet states take up more space in D-cache, and causes some states to be evicted from D-cache before their packets are scheduled again. 5G UPF falls into this case, showing 1.28× more backend stall slots. To further show the effect, we expand the length of TCP data packets to MTU, making application accesses as much state as possible. In this case, 5G UPF only presents 1.31× more backend stall slots, which is only a marginal increase, because its majority state accesses are per-stage instead of per-packet. We argue that the trend of backend stall depends on the application profile, and NanoPL will still gives net performance improvement because it targets on applications with heavy frontend bound.

NanoPL is semantically consistent. For each tested application, we first adopt a "verification run", which configure the application to print as many logs and statistics as possible. We find that the NanoPL-instrumented version outputs exactly the same statistics as the original version, while the logs are identical in number and contents but in a different order. We also test under a trace captured under an enterprise network, which consists of 11.2M packets with a wide variety of protocols encapsulated into GTP, and confirm the outputs are same (with 20.9% speedup on average on applications shown in §7.1). This proves that NanoPL can achieve semantic consistency while shuffling the instruction order.

7.3 Microbenchmarks

To understand and break down the performance benefits of NanoPL, we list the parameters that are critical to performance, and evaluate how performance changes when they differ. For the sake of brevity, we mainly involve snort2 and 5G UPF as the representative CDPAs in this section.

NanoPL reuses instructions within batch. Since NanoPL reuses I-cache within packets of the same batch, the setting of batch size is vital to performance. Packets are buffered until a batch with specific size is received. As in Figure 7, throughput grows as expected with increased batch size. At batch size 32, throughput boost of both applications is maximized, reaching 22.9% and 17.4%. The performance gain mainly comes from a mitigated CPU frontend as in Table 2. Similarly, L1 I-cache misses decrease logarithmically with batch size, which is a clear sign of I-cache reusing among packets in the same batch. Consequently, when batch size is set to 64, NanoPL achieves the lowest I-cache miss rate (-86.4% on snort2).

Table 3: Commitment window analysis. Commitment windows only occupy a small portion of code, providing sufficient space of scheduling site placement.

	#CP Vars	Largest CW	Ratio on I-cache	Total CW	Ratio on Codebase
snort2	38	1185 insts	14.8%	1607 insts	1.27%
5G UPF	103	893 insts	10.9%	2187 insts	2.66%

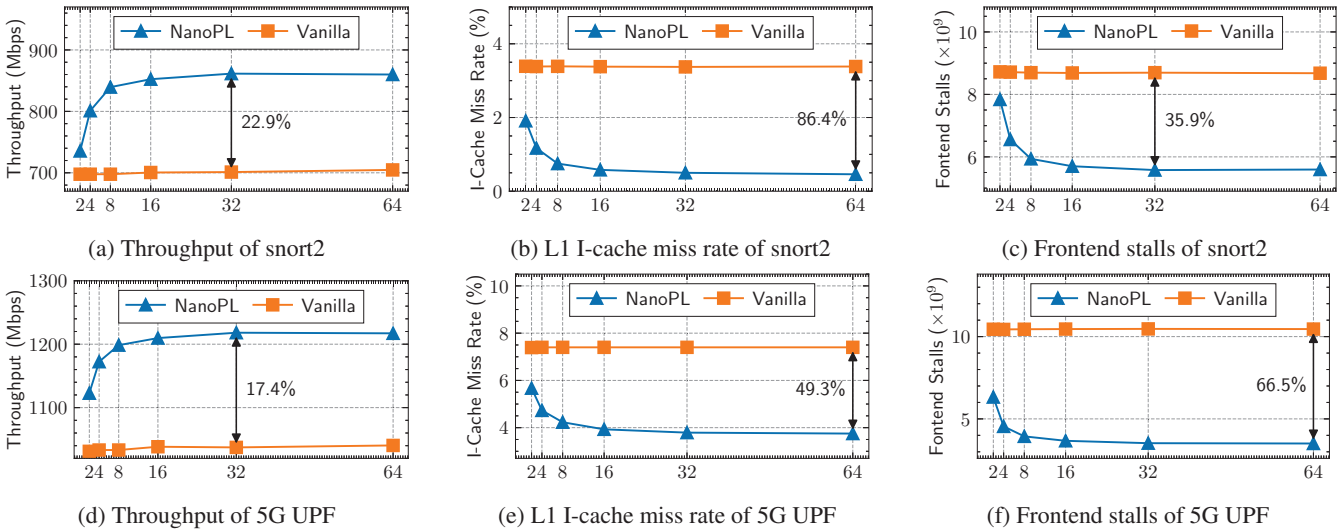
CP Vars: *Cross-packet variables* **CW:** *Commitment window*

While there is no technical limit, when batch size goes up to 32 and beyond, throughput starts to stay stable. This is caused by two reasons: (1) Code is already reused by many packets, leaving only marginal benefit; and (2) under large batch size, the working set of active variables also enlarges, hindering the improvement with more D-cache misses.

NanoPL benefits applications under larger I-cache. The most recent processors are equipped with larger I-cache to help complex applications. To test its effect, we migrate the applications to an ECS instance running on vCPU upon Intel Xeon 6982P-C, which has 64KB I-cache per-core. The average throughput speedup reduces from 22.3% to 16.9% in this setting. This is because a larger I-cache observes less misses and mitigates the frontend bound problem. For example, snort3 reports that its I-cache MPKI reduces from 20.2 to 8.0 in vanilla setting. However, this does not eliminates the design space of NanoPL because it further reduces I-cache MPKI to 1.64 (79.5% less) and still gains a 18.1% speedup. This shows that a larger I-cache indeed help applications by reducing I-cache misses, but still cannot solve frontend bound on CDPAs, emphasizing the need to deploy NanoPL.

NanoPL effectively identifies reusable code regions. We analyze the result of hot-path-based scheduling site selection policy proposed in §5. As a comparison, we also implement a conservative policy that restrict all possible paths between any two sets of scheduling sites to fit in I-cache. As shown in Figure 8a, the speedup ratio of hot-path aware policy is 2.9× and 1.6× higher than the conservative one on snort2 and 5G UPF respectively, due to reduced scheduling sites in Figure 8b. Consequently, fewer scheduling sites makes room for more instructions among scheduling sites. As in Figure 8c, the average instruction size of each stage under hot path aware policy is 1.9× and 2.1× more than the conservative policy, without a single stage exceeding the I-cache capacity.

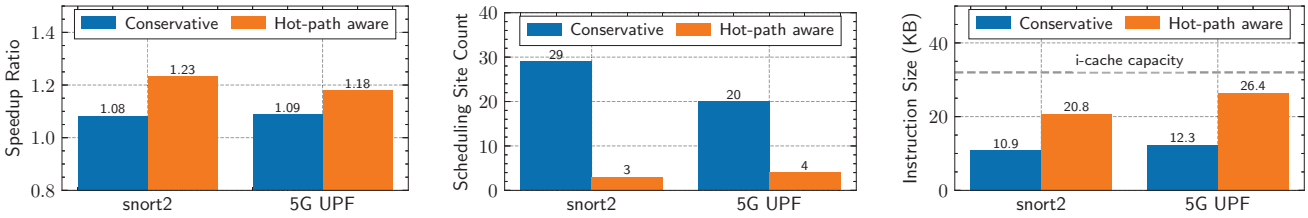
Since excessively large commitment window could exceed I-cache solely by itself, we measure the size of commitment windows in Table 3. It is shown that largest commitment windows cover several thousands of instructions. Considering a typical I-cache size of 32KB, which can store roughly 8K instructions, the largest commitment window takes 10.9%–



(a) Throughput of snort2 (b) L1 I-cache miss rate of snort2 (c) Frontend stalls of snort2

(d) Throughput of 5G UPF (e) L1 I-cache miss rate of 5G UPF (f) Frontend stalls of 5G UPF

Figure 7: The performance results with increasing batch sizes.



(a) Throughput speedup (b) Number of scheduling sites (c) Average instruction size of stages

Figure 8: The performance result of different scheduling site selection policies.

14.8% of the I-cache. Moreover, the total instruction numbers of commitment windows are minor, which only amount to less than 3% of the total codebase. These results indicate there are plenty rooms for NanoPL to safely place the scheduling site, and have high potential to fit the stages into I-cache.

NanoPL introduces manageable overhead. The overhead of NanoPL is twofold: offline analysis and online scheduling. As for offline analysis, snort2 (1.3 MB binary) takes 1624 seconds and UPF (915 KB binary) takes 1056 seconds for a single run. Though static analysis is a well-known extremely time-consuming task [36, 37], NanoPL only requires the value-flow of a restricted set of cross-packet variables, greatly reducing the analysis time. We observe that all applications in our experiments can be analyzed within an hour, which is fast enough for most use cases.

As for online scheduling, we evaluate the time spent during a single scheduling site invocation, which is on average 31 cycles. This is reasonable since the procedure of scheduling sites is as simple as saving necessary callee-saved registers and rebasing the PC register. However, even the overhead of a single scheduling site is minor, the performance degradation with increased scheduling sites is still large as in Figure 8, due to increased D-cache misses as discussed in §7.2. We observe that the per-packet L1 D-cache misses of conservative policy is 1.8× and 1.5× more on two applications, emphasizing the need to perform hot-path aware analysis to reduce scheduling sites.

NanoPL under configuration changes. We randomize the patterns in matching rules of snort2 and snort3, and run the unmodified binary with the new rules. The results show that NanoPL still gives a 20.2% and 29.6% performance improvement, which keeps the most portion of the original speedup. This is because the content of the matching rules only has a minor influence on the program path. Despite it is generally recommended to run the offline analysis again when the settings of applications change, NanoPL still preserves a large portion of performance improvement in common cases like rule updating, saving the analysis overhead if desired.

NanoPL is multi-threading compatible. To test the performance of NanoPL under multi-core environments, we split the traffic to 8 cores using S-RSS, and pin a 5G UPF worker thread to each core. We measure that the vanilla case scales the performance to 7.95×, and NanoPL improves the performance by 8.74×, which calculates to a 16.8% speedup. Recall that the performance benefit on a single core is 17.4%, which means that NanoPL is still able to retain 96.8% of the speedup under 8 cores. This is reasonable since L1 I-cache is exclusive to each CPU core, and NanoPL can independently mitigate the bottleneck on each core.

8 Discussions

We discuss the related work and limitations of NanoPL.

8.1 Relevant Optimizations

Batching optimizations. Batching is widely used by a vast majority of network applications [10, 14, 15, 25], amortizing the fixed overhead of NIC interaction, module initialization, etc. In fact, the benefit of batching can go beyond that: Batchy [25] models and fine tunes the batch size, realizing higher level of SLO requirements than naive batching. Reframer [15] discovers that shuffling packet order in batches can be critical to data cache utilization and thus performance. Similarly, NanoPL improves performance by finding an I-cache friendly execution pattern within a batch. Such methodology can further help a wider range of batching-enabled applications, *e.g.*, databases [11, 13].

Workload-aware optimizations. Existing literature has explored specializing applications under certain workloads to remove redundancy and improve performance. For example, applications might have duplicated packet and flow classification [23, 24], unreachable program modules [12], or hot and cold paths [26, 27]. However, due to the complexity of CDPAs, their packet processing paths will stay large even after specialization, causing the frontend bound to persist. In the setting of §2.1, snort3 accesses a total of 3833 cachelines during execution, which is over 7 times larger than I-cache. In contrast, NanoPL takes a similar workload-aware method, but redesigns the execution pattern for better I-cache efficiency.

Hardware-based optimizations. Besides I-cache, modern CPU frontend is a complex orchestration of a large set of hardware components, including iTLB, micro-operation cache, loop stream detector (LSD), branch target buffer (BTB), etc [18]. Despite the constant effort of reverse engineering [45, 46] and optimizing them [20, 34], the precise parameters and implementations remain undisclosed. The users are merely exposed to performance counters that are insufficient to understand and model their behaviors, complicating software optimization. Instead of going further on the hardware structures, NanoPL decides to solve this problem from software side. By rearranging the instruction execution pattern, the hardware components can have a smaller working set, and get optimized without being demystified.

8.2 Limitations

Worst case traffic. Since the benefit of NanoPL comes from code reusing within batch, its performance benefit will be limited on a small batch size. In this case, the scheduling overhead still persists and outweighs the I-cache reusing benefits, resulting in potential performance drop. A typical example is where packets come in large intervals, *e.g.*, behind a traffic shaper, and thus do not form a large enough batch. Following the configuration in Figure 7, we limit the batch size to 1 to eliminate code reusing, and find the throughput of two applications decreases by 3.9% and 4.5%. We argue that such cases are rare in practice because real-world traffic is mostly bursty and forms large enough batches.

Static analysis soundness. The consistency guarantee of NanoPL is based on the soundness of static analysis. That is, on a static CFG, *i.e.*, no modification after compilation, static analysis always finds all possible accesses of each variable. Note that this includes the case of dynamic paths. For example, whether the branch on Line 5 of Figure 3 will be taken or not depends on the runtime traffic patterns, but is both considered by static analysis.

On the other hand, when the CFG is modified dynamically during runtime, it may introduces variable accesses bypassing static analysis and cause conflicts. This happens in specific cases such as shared objects loading, just-in-time compilation, and self-modifying program. Even in these cases where analysis soundness cannot be guaranteed, we argue that NanoPL still provides critical hints. For example, a typical hot code update is implemented by dynamically loading a shared library and redirecting existing function pointers. The new functions bypassing static analysis will possibly violate semantics consistency under existing scheduling sites. Despite that, application developers can still manually modularize their code, while scheduling sites provided by NanoPL serving as I-cache friendly module boundaries.

Application-specific thread models. §7.3 shows that NanoPL can retain the speedup under symmetric multi-core scaling. However, it remains challenging to work with application-specific thread models where packet processing cannot be marked clearly. We argue that this problem should be solved by integrating more pre-knowledge into program analysis.

9 Conclusion

We present NanoPL, a novel framework to address the CPU frontend bottleneck in CDPAs. NanoPL introduces an I-cache-aware execution model that partitions application logic into cache-friendly stages and enables efficient instruction reuse through runtime scheduling. Our evaluation shows that NanoPL integrates seamlessly into real-world CDPAs, achieving up to 30.2% throughput improvement and reducing I-cache misses by up to 86.4%. These results demonstrate the potential of rethinking execution models to tackle CDPA-specific bottlenecks.

This work does not raise any ethical issues.

Acknowledgements. We would like to thank Guyue Liu for comments on the early version of the paper, the anonymous NSDI reviewers for constructive reviews, and our shepherd Ming Liu for valuable revision feedback. Hao Li is the corresponding author. This work is partially supported by the National Natural Science Foundation of China (No. 62572381 and No. 62272382).

References

[1] Cachegrind: a high-precision tracing profiler. https://valgrind.org/docs/manual/cg-manual.html, 2025.

[2] Dpdk. https://www.dpdk.org/, 2025.

[3] llvm-addr2line - a drop-in replacement for addr2line. https://llvm.org/docs/CommandGuide/llvm-addr2line.html, 2025.

[4] nDPI. https://www.ntop.org/products/deep-packet-inspection/ndpi/, 2025.

[5] Perfmon. https://github.com/intel/perfmon, 2026.

[6] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. High-speed software data plane via vectorized packet processing. *IEEE Communications Magazine*, 56(12):97–103, 2018.

[7] Mohamad Barbar and Yulei Sui. Compacting points-to sets through object clustering. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27, October 2021.

[8] Mohamad Barbar, Yulei Sui, and Shiping Chen. Object Versioning for Flow-Sensitive Pointer Analysis. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 222–235, Seoul, Korea (South), February 2021. IEEE.

[9] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, and Dejan Kostić. RSS++: load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT)*, pages 318–333, New York, NY, USA, 2019. ACM.

[10] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5–16, 2015.

[11] ClickHouse. Real-time data analytics platform. https://clickhouse.com/clickhouse.

[12] Bangwen Deng and Wenfei Wu. NFReducer: Redundant Logic Elimination for Network Functions with Runtime Configurations. In *IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–10, May 2021. ISSN: 2641-9874.

[13] DuckDB. A in-process SQL OLAP data management system. https://duckdb.org/.

[14] fd.io. VPP. https://wiki.fd.io/view/VPP, 2025.

[15] Hamid Ghasemirahni, Tom Barbette, Georgios P. Katsikas, Alireza Farshin, Amir Roozbeh, Massimo Girondi, Marco Chiesa, Gerald Q. Maguire Jr, and Dejan Kostić. Packet order matters! improving application performance by deliberately delaying packets. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 807–827, 2022.

[16] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155*, 2015.

[17] Intel. Intel® Dynamic Load Balancer. https://www.intel.com/content/www/us/en/download/686372/intel-dynamic-load-balancer.html.

[18] Intel. Intel® 64 and ia-32 architectures optimization reference manual: Volume 1. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2026.

[19] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. Twig: Profile-Guided BTB Prefetching for Data Center Applications. In *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 816–829, New York, NY, USA, 2021. ACM.

[20] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. Blasting through the Front-End Bottleneck with Shotgun. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 30–42, New York, NY, USA, 2018. ACM.

[21] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. Boomerang: A metadata-free architecture for control flow delivery. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 493–504, 2017.

[22] Chris Lattner and Vikram Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[23] Hao Li, Yihan Dang, Guangda Sun, Changhao Wu, Peng Zhang, Danfeng Shan, Tian Pan, and Chengchen Hu. Programming Network Stack for Physical Middleboxes and Virtualized Network Functions. *IEEE/ACM Transactions on Networking*, 32(2):971–986, April 2024.

[24] Guyue Liu, Yuxin Ren, Mykola Yurchenko, K. K. Ramakrishnan, and Timothy Wood. Microboxes: high performance NFV with customizable, asynchronous TCP stacks and dynamic subscriptions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 504–517, New York, NY, USA, 2018. ACM.

[25] Tamás Lévai, Felicián Németh, Barath Raghavan, and Gábor Rétvári. Batchy: batch-scheduling data flow graphs with service-level objectives. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation (NSDI)*, pages 633–650, USA, 2020. USENIX Association.

[26] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. Domain specific run time optimization for software data planes. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1148–1164, New York, NY, USA, 2022. ACM.

[27] László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis, Levente Csikor, Ferenc Juhász, Attila Kőrösi, and Gábor Rétvári. Dataplane specialization for high-performance openflow software switching. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, page 539–552. ACM, 2016.

[28] Andy Newell and Sergey Pupyrev. Improved Basic Block Reordering. *IEEE Transactions on Computers*, 69(12):1784–1794, December 2020.

[29] omec project. 4g/5g mobile core user plane. https://github.com/omec-project/upf, 2025.

[30] Guilherme Ottoni and Bertrand Maher. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 233–244, February 2017.

[31] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 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)*, page 2–14, 2019.

[32] Tristan Ravitch. travitch/whole-program-llvm. https://github.com/travitch/whole-program-llvm, 2025.

[33] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, page 16–27, USA, 1999. IEEE.

[34] David Schall, Andreas Sandberg, and Boris Grot. Warming Up a Cold Front-End with Ignite. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 254–267, New York, NY, USA, 2023. ACM.

[35] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. Propeller: A profile guided, relinking optimizer for warehouse-scale applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 617–631, New York, NY, USA, 2023. ACM.

[36] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 693–706, New York, NY, USA, 2018. ACM.

[37] Qingkai Shi and Charles Zhang. Pipelining bottom-up data flow analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*, page 835–847, New York, NY, USA, 2020. ACM.

[38] snort. Snort - Network Intrusion Detection & Prevention System. https://www.snort.org/, 2025.

[39] snort3. snort3/libdaq. https://github.com/snort3/libdaq, 2025.

[40] suricata. Suricata: Home. https://suricata.io/, 2025.

[41] SVF-tools. SVF-tools/SVF, January 2025.

[42] Gil Tene. wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2, 2025.

[43] Qiongwen Xu, Sebastiano Miano, Xiangyu Gao, Tao Wang, Adithya Murugadass, Songyuan Zhang, Anirudh Sivaraman, Gianni Antichi, and Srinivas Narayana. State-Compute Replication: Parallelizing High-Speed Stateful Packet Processing. In *USENIX NSDI '25*, pages 937–957, 2025.

[44] Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, March 2014.

[45] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. Half&half: Demystifying intel's directional branch predictors for fast, secure partitioned execution. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1220–1237, 2023.

[46] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom. BunnyHop: Exploiting the Instruction Prefetcher. In *USENIX Security '23*, pages 7321–7337, 2023.