Interleaved Function Stream Execution Model for Cache-Aware High-Speed Stateful Packet Processing

Ziyan Wu, Yang Zhang, Feng Tian, Minjun Wu, Antonia Zhai, Zhi-Li Zhang*
University of Minnesota, Twin Cities
Minneapolis, USA
{wu000598, zhan3248, tianx399, wuxx1354, zhai}@umn.edu
*zhzhang@cs.umn.edu

2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS) | 979-8-3503-8605-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDCS60910.2024.00056

***Abstract*—The evolving network infrastructure, particularly the 5G core network, is increasingly adopting cloud technologies. This shift brings to the forefront the challenge of meeting the demanding per-packet processing requirements posed by multi-hundred Gbps Ethernet NICs (network interface cards). While traditional NFV (network function virtualization) platforms are effective on older hardware, the per-packet run-to-completion (RTC) execution model for per-packet processing suffers from stalling on state access due to L1/L2 cache misses. Although previous work applying software prefetching can mitigate the issues, their applications are fundamentally limited by the nature of a single execution stream, hence limiting them to batch lookups, suffering from control-flow divergence, and requiring manual tuning. To address the limitations, we introduce a novel interleaved function stream execution model that exploits the function-level parallelism through memory-level parallelism, targeting feature-rich network functions such as 5G Core. To provide the visibility into network functions, we introduce a novel programming model based on the principle of *Granular Decomposition*, which provides deep visibility into the state access by decoupling the state in a *more fine-grained* manner compared to traditional modular approaches. We integrate these two innovative designs into a new open-source NF platform, which we refer to as *GuNFu*. We have tested GuNFu on widely deployed network functions such as 5G UPF (User Plane Function), 5G AMF (Access Management Function), NAT (Network Address Translator) and others. Extensive evaluations reveal that GuNFu can achieve throughput ranging from 1.5 to 6 times over the traditional modular approach.**

***Index Terms*—NFV, execution model, mobile core**

I. Introduction

With growing demands for programmability and scalability, networking infrastructure is increasingly virtualized or "cloudified". This is epitomized by emerging 5G networks. For example, the 5G core is organized as a set of network functions (NFs), such as User Plane function (UPF), access and mobility management (AMF) function. All 5G core NFs are examples of *stateful* NFs where packet processing logic depends on a specific state (e.g., user or session contexts) maintained per user device and/or user session. Many of the NFs widely used in (backend) data centers [1, 2, 3] such as load balancers (LBs), network address translators (NATs), and firewalls (FWs) are also stateful as they maintain state *per flow*.

Using the zero copy principle [4, 5] and sharing-nothing principle [6], previous approaches achieve significant performance improvements. They *nonetheless* face significant scaling challenges with emerging workloads. Today, 100/200 Gbps Ethernet NICs (network interface cards) are commonplace and affordable, leading to *exponentially* decreasing per packet processing budget. As will be further argued in §II-A, it is crucial to ensure that the stateful NF operations on the packets and states are *L1/L2 cache-bound*. Experiments in §II-C show that the existing per-packet *run-to-completion* (RTC) execution model employed in existing NF frameworks [7, 8] – and feature-rich open source 5G projects [9, 10, 11, 12] under active development that use the RTC model – cannot avoid the cache miss penalty due to the increasing complexity of the per-flow state of the network function and the ability to leverage the increasing concurrency in network workloads. Despite the fact that software prefetching is employed to minimize cache misses, it is generally applied to a single stream of NF execution within a limited set of data structures (e.g., table lookups) [13, 14], and suffers from control-flow divergence [15, 16].

We propose a novel *interleaved function stream execution model* (Figure 1): a method that concurrently runs different function streams to exploit function-level parallelism through memory-level parallelism. In our work, a function stream is defined as a sequence of functions (NFAction in §IV), and each stream corresponds to the processing of a packet belonging to a network flow, a sequence of packets matched by a particular predicate, which might be a set of fields in the packet headers or payload. In this model, the states (NFState in §IV) of functions from independent execution streams can be retrieved concurrently through software prefetching. The interleaved function stream execution model takes advantage of the inherent parallelism to be processed per core and takes advantage of the *fine-grained* software prefetching technique to optimize the utilization of the L1 / L2 cache.

However, the realization of the proposed interleaved function stream execution model effectively presents several challenges. To perform software prefetching efficiently, we need insights into the operations of network functions to make intelligent decisions about what states are needed next and what actions must be performed on a given NF function stream.

2575-8411/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDCS60910.2024.00056

Unfortunately, the information is not visible to the runtime in current open-source network function platforms [7, 8] or open-source network function implementations [9, 10, 11, 17] and makes it hard to integrate software prefetching effectively. We also need a light-weighted runtime system to support efficient scheduling among different execution streams.

To address these challenges, we propose *GuNFu* (§III), a *granular*, *cache-aware* network function platform for stateful packet processing at line rate. We advocate for a *Granular Decomposition* programming model to provide *deep visibility* into fine-grained components defined by *NFState*, *NFActions* in our NF Model (§IV), which is more *fine-grained* than traditional modular network functions. To support flexible scheduling in runtime, we develop an abstraction called NF-Tasks to support flexible scheduling among different function streams, which significantly outperforms the per-packet RTC model [18] that is currently used in today's NF platforms and NF implementation [9, 12, 17, 19, 20]. We introduce several additional compilation optimization techniques (§VI) to further enhance performance by taking advantage of the *deep visibility*. Evaluation (§VII) shows the effectiveness of GuNFu in cache utilization and performance improvement ranging from 1.5 to 6 times over the conventional per-packet RTC model.

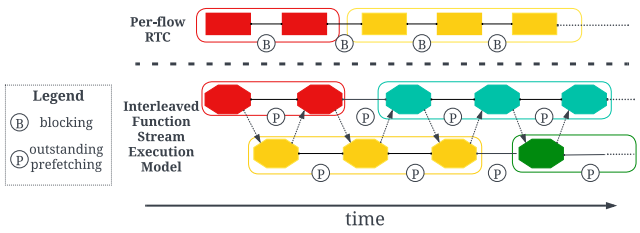
II. Challenges in Accelerating Stateful NFs

We argue why per-packet RTC cannot address the issues on state accessing in network functions and the proposed interleaved function stream execution model is a better candidate.

A. Growing Network, CPU and Memory Gaps

The network speed has grown tremendously in the last 20 years, evident from the growth of the Ethernet speed, which has grown 200 times from 1 Gbps in 1997 to 200 Gbps in 2017 and is expected to reach 800 Gbps/1.6 Tbps in the near future [21]. We have calculated the *(average) per packet time budget* to process 64B packets with a processor clocked at 3.4 GHz, as the line speed increases from 10 Gbps to 400 Gbps. In particular, to keep up with 100 Gbps line rate, the average *per-packet* processing time is only about 6.7 ns (about 23 clock cycles at 3.4 GHz). And the access latency of the memory hierarchy is about 1.2 ns, 4.1 ns, 13-20 ns and 70-125 ns respectively for L1-cache, L2-cache, Last-level cache, and DRAM.

The relatively high cost of accessing the lower layers of memory hierarchy shows the importance of cache utilization in NF software design and implementation. Ideally, we want the state needed for NF processing to reside within the L1/L2 caches *before it is accessed*. Otherwise, if the core has to wait for the NF state to be fetched from the LLC, or worse, from the DRAM, precious CPU cycles are wasted. With the increasing concurrency of network workloads that each core must process, contention for L1 / L2 cache resources causes a rise in cache misses, resulting in *unavoidable* LLC/DRAM memory accesses. This is particularly the case when the NF state required for packet processing is large/complex.



(Different Colors represent different Execution Streams)

Fig. 1: An illustration of the comparison of per-flow RTC and the interleaved function-stream execution model on a single core. Our framework employs Granular Decomposition to explicitly decouple states that may cause cache misses from functions, thereby enabling efficient prefetching.

B. Empirical Study for Per-packet RTC Execution Model

The idea behind the per-packet RTC execution model is that within one network function module, the underlying runtime system processes the packet from one network flow without yielding until it is done. The per-packet RTC model is ubiquitously used in the open-source state-of-the-art modular network function framework [8, 22, 23] and standalone network function projects [9, 10, 11, 17] under active development.

The local state access becomes an obstacle for scaling of stateful network functions under per-packet RTC execution model. Take the NAT from BESS or FastClick [22, 24] as example, they orchestrate the batched packet processing computation in a loop and in each iteration they run the packet processing to completion, including the matching for individual packets, accessing the per-flow states which all cause memory-access stalling during the high-concurrency workload. For more intricate network functions such as AMF, UPF from the open source implementations [9, 10, 11, 12], the per-packet RTC execution model is also applied: for each packet, they first match the user context information, then based on the states, do a series of computations based on the user context, without yielding until done.

To illustrate the issues for cache utilization for per-packet RTC execution model empirically, we used two network functions in the mobile core, 5G UPF and 5G AMF.

Testbed Setup: The testbed used for our experiments consists of two 48-core servers (Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz), each equipped with a dual port Mellanox ConnectX-6 EN 100Gbps DPDK capable NIC. One server acts as the device under test (DUT) and the other as the traffic generator.

Methodology: For both network functions, we examine the implementation from L25GC [17]. To generate the workload, we extend the Telco Pipeline Benchmarking System [15, 25] of MGW (Mobile Gate Way) use cases and vary the number of PFCP sessions and the number of PDRs (Packet Detection

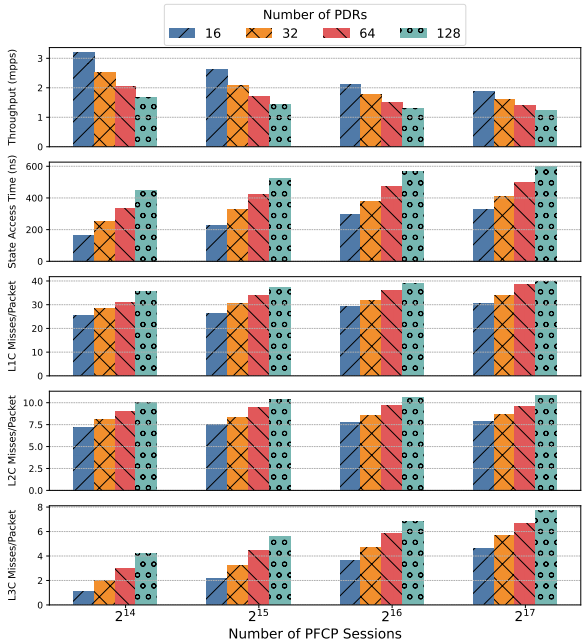


Fig. 2: Impact of concurrency on the performance on UPF.

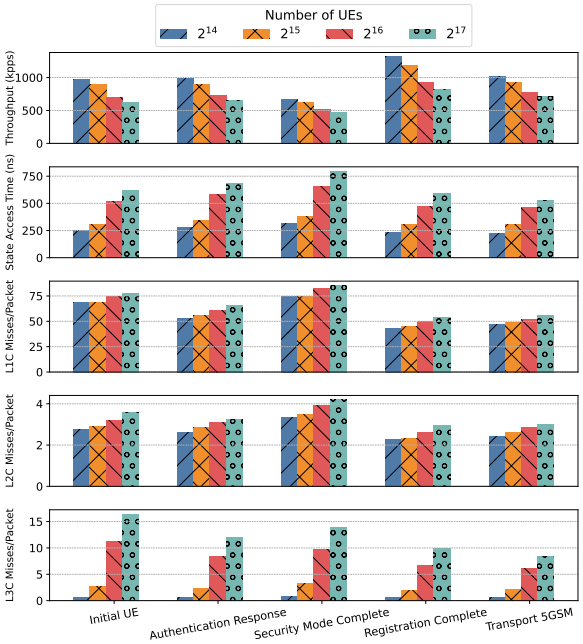


Fig. 3: Impact of state complexity for state-intensive messages in UE Initial Registration on performance.

Rule) to see its impact on state access. For AMF, we take the test cases provided by Free5GC [9], vary the number of UEs and examine the messages that require state access involved in the initial registration. As we focus on the state access performance, we extract related call flows and state access involved in the process and port them to DPDK. For both experiments, we tested single-core performance. For microarchitecture metrics and state access time, we use the perf utility from Linux to measure PMU events and CPU time by the state accessing functions.

(EXP A) Impact of Growing Concurrency: Figure 2 illustrates the impact of growing concurrency on the performance. From the results of the experiment, we can see that when the number of PFCP sessions and the number of PDR increases, the performance of the 5G UPFs degrades. The further profiling shows that the root cause of the performance degradation is due to the increased per-packet process time because of increased flow matching and state access overhead. The increase in overhead is because with the increased size of flow table, the matching process and the access to the per-flow state (PFCP sessions) and sub-flow state (PDR) is more likely leading to a cache miss as the corresponding entry in the flow table and the per-flow state is unlikely in the cache when the flow arrives.

(EXP B) Impact of State Complexity: Figure 3 shows the impact of state complexity on performance. Compared to UPF, the per-flow state is much larger for AMF, as the information for keeping track of the UE states is richer. The size of the per-flow state in AMF is greater than 20 cache lines. As a result, state access takes up a large portion of the total packet processing time of AMF. The cache profiling metrics reflect the impact of demanding per-flow state access with much higher L1/L2 cache, LLC misses per-packet. It is worth noting that, for some messages, the actual state accessed is much smaller than the touched cache lines, resulting in the program asking for more cache lines than it actually needs. This inefficiency further degrades performance.

The empirical study shows that the per-packet RTC model is not sufficient to tackle efficient cache utilization under the emerging constraints of higher concurrency and complexity.

C. Case for the Interleaved Function-stream Execution Model

As illustrated by the Figure 1 and the empirical study, for the per-packet RTC, within a network function module, it is likely that the functions or groups of instructions will block when accessing the states that cause cache misses. Although previous work [13, 14] has shown that software prefetching can improve instruction-level parallelism, the critical downside is that they are subject to a single execution stream, resulting in the limitation to homogeneous workloads and homogeneous functionalities, i.e., batch lookup. The adaptive batching solution [15] suffers from state access latency for the first few packets when scheduling downstream modules, which can lead to SLA violation for queueing delaying. Furthermore, none of the works solves the problem of state access delay of the per-packet processing within the module after the batch lookup. Applying prefetching techniques naively suffers heavily from control flow divergence and requires the programmers on fine-tuning of the insertion of prefetching instruction [26], instead of focusing on the application logic. These limitations make it hard to integrate effective software prefetching beyond

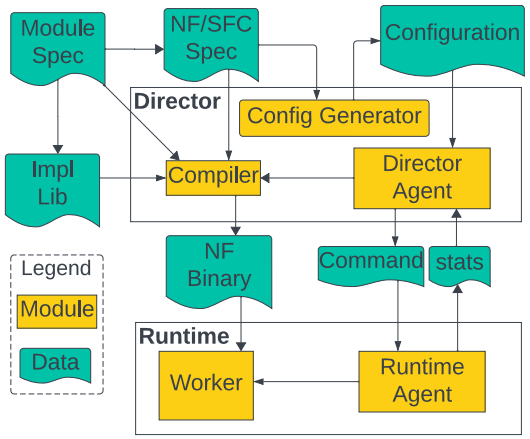


Fig. 4: GuNFu System Architecture Overview

the batch lookup data structure into more intricate network function projects.

The *interleaved function stream execution model* we propose does not suffer from this limitations. Instead of stalling on the state access, the interleaved function streams can enable the runtime system to prefetch the states that correspond to the next function for function-level parallelism and switch to the execution stream in which the corresponding state is available. It enables more efficient consolidation of different modules, since runtime has *visibility* on the function of the next module to be executed. Compared to per-packet RTC, it also applies to *heterogeneous workloads/functionalities*, which are common in feature-rich network functions dealing with different user behaviors, hence different state lookup methods, application logic executed and states to be accessed. As a *general* cache optimization solution, it not only applies to state lookup but also improves general state access.

However, systematically integrating the novel interleaved function stream execution model into the network function platform is not without challenges. Current modular network function implementation does not provide an explicit relationship between the state (which may cause cache misses) and functions with *fine granularity*, hence our need to redesign a programming model that allows us to refactor the network functions to provide that deep visibility into the network functions in terms of the states accessing for each function. As a result, the runtime can take advantage of the information to perform prefetching effectively and transparently to the developers. Besides, as the current network function platforms only support per-packet RTC, we need a light-weight runtime for efficient function stream scheduling. We lay out our systematic solution in the next section.

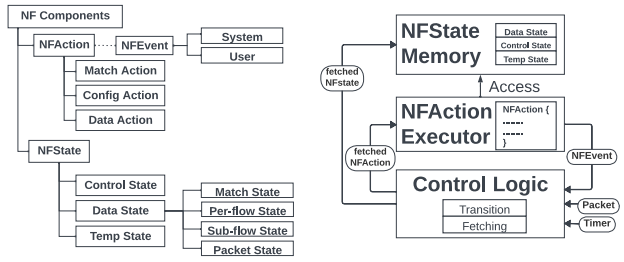
III. System Overview

GuNFu is the first to provide a *systematic* solution to the challenges of applying interleaved function stream execution model. To provide deep visibility for runtime, our programming model is designed based on the paradigm of *granular decomposition* – a collection of design principles that effectively decompose the modules of network functions into even more finer-grained components – to provide deep visibility. This paradigm will be presented in detail in §IV. With visibility into network functions, our novel *interleaved function stream execution model* enables our NFV runtime to timely prefetch the required data and dynamically switch between multiple execution streams to mask the inevitable and costly LLC/DRAM access overheads, thus significantly enhancing *per core* CPU utilization. Our design is partially motivated by the following key insights. *1. Inherent Parallelism in NF Flow Processing.* We note that the most frequent operations, processing of different NF flows in a stateful network function, are independent. For example, for the UPF, the process of encapsulation of a packet depends only on its per-flow state and subflow state (PDR), which is a private operation. This means that before the network function begins expensive memory operations accessing its PDR/packet header, it should issue a prefetching instruction and switch to other execution streams for different flows for useful work, instead of blocking on the L2 cache miss or LLC miss. *2. Critical Per-Flow Operations.* Cache misses are frequently caused by per-flow state access, flow matching, and packet header access in the context of network functions. We therefore optimize these operations by intelligently and systematically applying software prefetching.

High-level Workflow: In Figure 4, we present an overview of the system data flow graph. The system consists of two main parts: the director and the runtime. The director works as the orchestration and control plane, and the runtime works as the data plane of the NFV system. The architects provide the program specification of modules using the principle of Granular Decomposition (see §IV). Developers provide implementation according to the specification. Architects or operators can write an NF/SFC specification to specify the network function, service function chain that must be deployed in the infrastructure according to a functional or operational requirement. The configuration generator will generate a configuration template that needs to be filled out by operators and be used to initiate/configure the network functions. The director compiler will take the specifications and corresponding NF implementations to generate the NF binary (see §VI). The director will deploy and start the runtime (see §V) on the hosting machine. During execution, the runtime agent will receive configuration commands from the director through the director agent for initialization and dynamic configuration, and it will exchange operational statistics with the director.

IV. Granular Decomposition

To give visibility to the network functions, we need to design a programming model to explicitly separate the code and data with finer granularity than traditional modular network functions and to describe how the data is consumed by the NF



(a) NF Components (b) NF Execution Model

Fig. 5: Detailed categories of NF components and an illustration of logical view of NF Execution Model

modules. In addition, we propose the Granular Decomposition Property for deciding the granularity of the decomposition.

A. NF Components and NF Execution Model

We present the key elements (Figure 5(a)) in our NF computational model and present the Granular Decomposition Principle and the execution model.

NFEvents: NFEvents are notifications of changes that the network functions handle. These notifications can be categorized into two main types: system events and user events. System events are triggered by or sent to the outside environment of NF, such as the arrival of a network packet or a request from user-defined actions to the hardware. On the other hand, user events are inner events triggered by user-defined actions and can lead to subsequent actions within a module.

NFState: NFStates are the data that need to be stored and accessed during the packet processing life cycle. We classify states into the following: data states, control states, and temporary states. Data states include match state, per-flow state, sub-flow state, and packet state. Match state are data structure used for flow classifying that map the packet to its per-flow state or sub-flow state, typically using a hash table or tree searching structure. Control states are states that persist during the lifetime of an instance of a network function and are shared among the flows. Since the control state tends to be small and shared among the flows (its space complexity is not related to the number of flows), it is likely always in the L1/L2 cache. Temporary states are states that need to persist among the invocation of actions for a particular packet. These states are used to store intermediate results and are typically discarded after the packet has been processed. The performance of accessing the NFStates depends on whether they are in the L1/L2 cache. Motivational experiments show that the access to the match state and per-flow state will lead to L1/L2 cache misses with high concurrency workload.

NFAction: NFActions are NFEvent handlers. NFActions are categorized into match action, data action, and config action based on the states with which they interact. Match actions interact with match states to locate the per-flow/sub-flow states. Data actions are actions that interact with the data states. Config actions interact with the control state. The performance of each NFAction depends on the NFState they are interacted with whether in the L1/L2 cache.

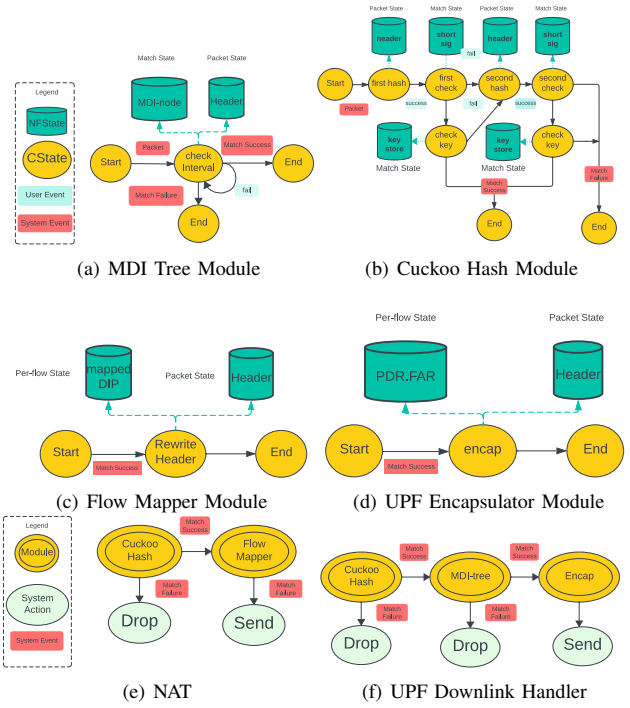
Granular Decomposition Property: Since a module can be decomposed into NFActions with arbitrary granularity (as a result, the granularity of the correspondence of the data and code), deciding the granularity of the decomposition is important, and we found any decomposition satisfying the following property to be useful: whether or not the access of data state in NFAction should not depend on the computation within the NFAction. The implication is that the variables in the data states (per-flow state, part of the packet, or match state) needed should be decided before the execution of NFAction. The rationale behind the principle is the following: Before the execution of an NFAction, all the portions of the data state related to it should already be in its NFState memory (where we will map to L1/L2 cache later). If a part of the portion is not actually accessed because of the dynamic behavior within the action, then the portion wastes valuable L1/L2 cache space as it is not needed. We call the decomposition of an NF satisfying this property "Granular Decomposition".

NF Execution Model: Figure 5(b) illustrates a "virtual machine" that runs an NF processing one packet based on the notion of granular decomposition. The virtual machine consists of an NFState memory, an NFAction executor, and NF Control Logic. NFState Memory is the data storage area for different states that the network function might need. It would store the portion of data states, control states, and temporary states that the packet needs to access. We need to ensure that all the data, when mapped to the host environment, should be in the L1/L2 cache when executing the action. NFAction Executor is the component responsible for the actual execution of NF actions. It takes fetched NFActions and executes them, presumably altering or utilizing the NFState as needed. Control logic determines the selection of an action based on various types of events (generated by the system or the user). Importantly, it is also responsible for preparing the necessary NFState for the actions to access. After the NFAction is executed, a new NFEvent is generated, and the control logic may select further action to process the packets.

Formalizing Execution Model as FSM: Under the execution model, the control logic of an NF can be modeled as an FSM (finite state machine). Let $CS = \{CS_1, CS_2, \ldots, CS_n\}$ be a set of control logic states to represent the stages to process a packet. The set A_{nf} is a collection of NFActions, and S_{nf} is a collection of NFStates. The set of NFEvent to which the network function can respond is defined as E_{nf}. The $CSFSM$ can be defined as (CS, Δ, F).

The transition function $\Delta : CS \times E_{nf} \rightarrow CS$ describes how the network function transitions between control logic states in response to NFEvents.

$$\Delta(cs_i, e) = cs_j, \quad cs_i, cs_j \in CS, \quad e \in E_{nf}$$



(a) MDI Tree Module (b) Cuckoo Hash Module

(c) Flow Mapper Module (d) UPF Encapsulator Module

(e) NAT (f) UPF Downlink Handler

Fig. 6: Module/NF Specification Diagrams

A fetching function $F : CS \rightarrow A_{nf} \times 2^{S_{nf}}$ is defined to determine the appropriate action and subset of states that the control logic should fetch, based on the current control state.

$$F(cs_i) = (a_i, S_i), \quad cs_i \in CS, \quad a_i \in A_{nf}, \quad S_i \subseteq S_{nf}$$

Two network functions NF_1 and NF_2 with compatible transition functions Δ_1 and Δ_2 can be composed to form a more complex network function $\text{NF}_{\text{composite}}$. Let CS_1 and CS_2 be the control logic states of NF_1 and NF_2, respectively. A_{nf1} and A_{nf2} be their respective sets of actions, and S_{nf1} and S_{nf2} be their respective sets of states. The control logic state set $CS_{\text{composite}}$ for $\text{NF}_{\text{composite}}$ could be defined as $CS_1 \times CS_2$. The fetching function $F_{\text{composite}} : CS_{\text{composite}} \rightarrow A_{nf1} \cup A_{nf2} \times 2^{(S_{nf1} \cup S_{nf2})}$ and the transition function $\Delta_{\text{composite}} : CS_{\text{composite}} \times E_{nf} \rightarrow CS_{\text{composite}}$ can be defined in terms of Δ_1, and Δ_2.

B. Programming Examples

Programming consists of three parts: module specification, NF/SFC specification, and action implementation. We use yaml as the language for the specification language due to its readability. For the implementation of the action, we design a C-like DSL, NF-C, to describe the NFAction logic, allowing developers to describe the application logic in an SPMD model.

We use a 5G UPF (downlink handler) and NAT as examples to illustrate how to provide a specification. The UPF consists of three modules, the Cuckoo-hash table (Figure 6(b)) to map

Listing 1: Flow Classifier Specification

```
# Flow Classifier Specification
category: StatefulClassifier
parameters: # for init, config
- entry_num
- header_type
transitions:
Start,packet->get_key
get_key,get_key_done->hash_1
hash_1,hash_done->check_1
check_1,MATCH_SUCCESS->End
check_1,check_failure->hash_2
hash_2,sec_hash_done->test_2
check_2,MATCH_SUCCESS->End
check_2,MATCH_FAIL->End
NFAction_NFState:
- hash_1:
  - {header_type} # packet state
- bucket_check_1:
  - bucket # match state
- hash_2:
  - {header_type}
- bucket_check_2:
  - bucket
- key_check_1:
  - key_store # match state
- key_check_2:
  - key_store # match state
```

Listing 2: Flow Mapper Specification

```
# Flow Mapper Specification
category: StatefulNF
transitions:
Start,MATCH_SUCCESS->flow_mapper
flow_mapper,packet->End
NFAction_NFState:
- flow_mapper:
  - ip # mapped ip
  - port # mapped port
```

Listing 3: NAT Specification

```
# SFC Specification
0:receive_packet,packet->1:
        flow_classifier
# name of declared modules omitted
1,packet->2:flow_mapper
```

Listing 4: Flow Mapper Implementation in NF-C

```
// Implementation Using NF-C
NFAction(flow_mapper) {
Packet.src_ip=PerFlowState.ip;
Packet.dst_ip=PerFlowState.port;
Emit(Event_Packet);
}
```

Fig. 7: Example specification (YAML) and code (NF-C).

the teid to the PFCP session (per-flow state), multidimensional interval tree (MDI tree, Figure 6(a)) to map the five-tuple to the PDR (subflow state) and the UPF encapsulator (Figure 6(d)) to tunnel the packet to the RAN. In the stateful matching operations in Cukcoo Hash table and the interval tree, significant overheads are caused by the pointer chasing operations associated with accessing the match state, which include bucket accessing or accessing the next child node will likely lead to cache misses. The UPF encapsulator utilizes the FAR (Forwarding Action Rules) from PDR. Figure 6(f) shows how to use the composability of each component to compose the functionality of the UPF downlink handler. For the NAT, we compose (Figure 6(e)) the Cuckoo Hash table and the flow mapper (Figure 6(c)) that rewrites the header based on the mapping rule.

To provide the specification, the designer needs to write a YAML file for different modules and the composition of modules. Listing 1,2 shows the example specification for the Cuckoo hash table and the flow mapper. For the composition, the designer needs to write the transition from one module to the other. To provide the implementation for NFAction, developers need to write a function in NF-C, our extension for C. The NF-C provides newly introduced keywords such as Packet, PerflowState, SubflowState, MatchState, and TempState to have access to various states in NFState memory. The example code for a flow mapper is provided in List 4.

V. Runtime Architecture

Runtime provides the environment for the execution model explained in the previous section. The key innovation in the runtime is effectively applying the interleaved function stream execution model for stateful network functions, an extension of the NF execution model. Using the interleaved function stream execution model, runtime can concurrently run multiple execution streams for better parallelism. To enable efficient

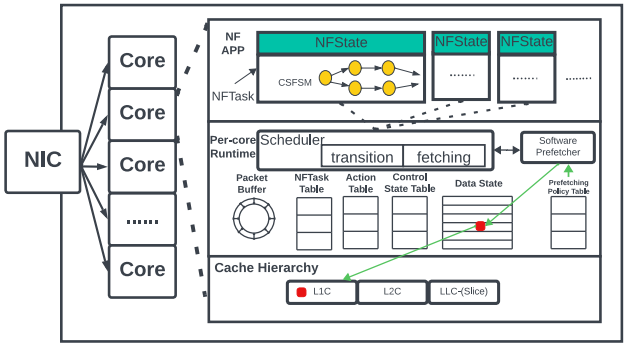
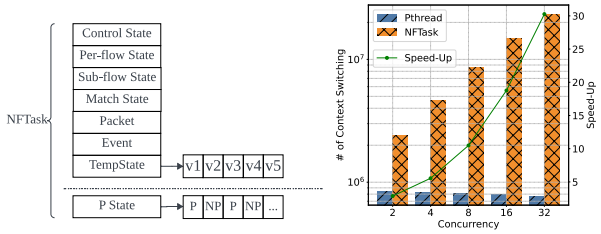


Fig. 8: Runtime Architecture



(a) NFTask Data Structure (b) Scalability of NFTask

Fig. 9: Evaluation of the maximum number of context switching per second of NFTasks of one core. Compared to pthread by Linux Kernel, our NFTask can take advantage of the parallelism.

scheduling, we leverage an abstraction called NFTask to keep track of NFAction streams for the processing of different flows.

Runtime Overview: Figure 8 illustrates the runtime architecture. Our system has a per-core runtime, which we call worker, and each worker has exclusive management of the resources on its assigned core. The core runtime is responsible for orchestrating the execution of NFs of different execution streams with its scheduler. During scheduling, it automatically executes the prefetching instruction to timely put the related NFState to the L1/L2 cache for better performance.

NFTask: NFTask is an abstraction of a lightweight (see Figure 9(b)) execution environment for an execution stream of NFActions for a packet. The key data structure used to manage it is illustrated in figure 9(a). Each field in the NFTask corresponds to the components described in the NF Model; thus, an NFTask maintains all the context that needs to process the packet. When the NFAction of the network function accesses the NFState, it essentially utilizes the reference of the fields to have access to the data. Although all the NFTask share the same memory space, NFTask also provides isolation, as the action cannot access a memory address other than the one referenced in an NFTask through the compilation check.

Cache Management: To ensure efficient usage of prefetching, we utilize a field in NFTask instruction called the P state to keep track of whether the NFState corresponds to a control logic state already prefetched, and hence it is likely in the L1/L2 cache. When it does need prefetching, it will reference the prefetching policy (generated by the correspondence of

Algorithm 1 Runtime Scheduler

```
NFTasks ← NFTasks(max_interleaved)
while True do
    PacketBuffer ← Receive packets
    Initialize NFTasks, initial transition, and fetching for all
    n_t ← 0                                   ▷ Current NFTask number
    while Packet processing is incomplete do
        if not isPrefetched(NFTasks[n_t].p_state) then
            Prefetch(NFTasks[n_t], prefetch_policy)   ▷ Prefetch NFState
            Continue
        end if
        ActionExecutor(NFTasks[n_t])          ▷ Execute the NFAction
        if NFTasks[n_t] is complete then
            Initialize NFTasks[n_t]
        end if
        Transition(NFTasks[n_t])              ▷ Execute Control Logic
        Fetch(NFTasks[n_t])                   ▷ Prepare NFState and NFAction
        n_t ← Next(NFTasks,n_t)               ▷ Switch to the next NFTask
    end while
end while
```

NFAction and NFState) to prefetch the specific cacheblocks of the corresponding NFState.

NF Management: For NFAction management, we use an action table to store all the NFActions deployed in the runtime, each entry contains a function pointer to the function. When the runtime executes an NFAction in an NFTask, it will use the function pointer to call the function, with all the NFStates referenced in the NFTask as parameters. NFState management requires different ways to manage different kinds of state. For the control state, we maintain a control state table that contains the list of references to each control state of NFs in the SFC. The control state and the match state will be allocated during the initialization stage. For the packet state, as it arrives in the packet buffer in the host, we store its reference into the NFTask to achieve zero-copy during its lifetime. For the per-flow state and sub-flow state, we pre-allocate datablocks, whose size is the size of each entry times the maximum concurrency that the core plans to support configured by the operators. When the matching is done, the matching result, which is an offset to the table, will be stored to the Per-flow state/sub-flow state field in the NFTask data structure, which enables the NFAction to access their per-flow/sub-flow state.

Scheduler: Our scheduler is responsible for achieving interleaving between NFTask and inserting the prefetch instruction in a timely manner. Our scheduler runs a round-robin policy to switch among different NFTasks. The scheduler essentially extends the control logic of the NF Model introduced in the previous section as it is responsible for the transition and fetching of control logic of multiple streams. After the transition and fetching step is done, the NFStates of the NFAction need are determined, and the software prefetcher can use the visibility to prefetch them as needed. The Algorithm 1 illustrates the scheduler process. At the beginning of the loop, it will receive a batch of packets. After the batch of packets is ready, the worker will initialize a set of NFTask, loading packets into the NFTask and setting up the initial transition and fetching triggered by the ”Packet” system event. Then it will

enter another loop, processing each NFTask and re-initializing finished NFTask if they are done.

VI. Compilation and Optimization

In this section, we discuss the compilation process that maps the program into the form that can be executed by the runtime. We also apply several optimization techniques, including data-packing and redundant matching removal.

A. Compilation

Our compiler takes two inputs: one is the specifications and one is the NFActions involved from the NF Implementation Library provided by the programmer. The output will be the binary of the NF Application bundled with the runtime.

NF-C Compilation: Our compiler takes the implementation of NFActions using NF-C and generates the C code. First, we translate the function signatures into a C compatible one that takes pointer to **struct NFTask** (Figure 9(a)) as a parameter. Second, we replace the operations related to the extended keywords of NFState with the pointer-reference operation of NFTask because different NFStates are members of the NFTask data structure. Third, the compiler collects all the temporary variables used and will be used to allocate the temporary variable fields of NFTask.

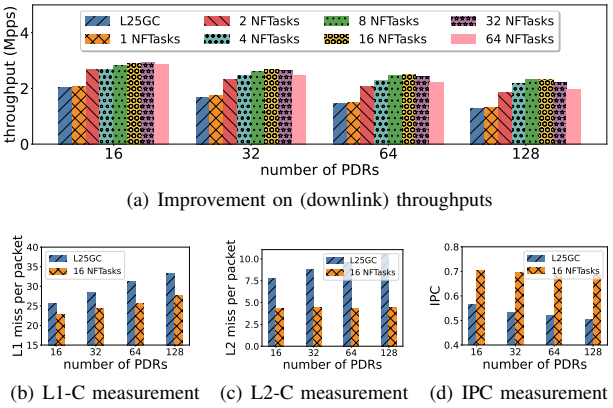
NF Binary Compilation: The compiler transforms the Module/NF specification into a Finite State Machine (FSM) for control logic, ensuring valid transitions. It generates action and control state tables from the specification, links action implementation libraries, and produces initialization functions in C. During deployment, the director agent uses this configuration file to initialize the network function through the runtime agent.

B. Compilation Optimization

The Granular Decomposition not only provides visibility for the scheduling in the runtime but also for the compilation optimization in the compilation time.

Redundant Matching Removing for SFC: Removal of redundant matching reduces the number of matchings required for SFC. For multiple stateful network functions chained together, naive consolidation will match multiple times and manage their per-flow state separately. We remove redundant matching to eliminate redundant operations that are full of pointer-chasing operations, by simplifying the control logic, and reuse the matching result of the first matching action for consecutive network functions that use the same key to locate the session states.

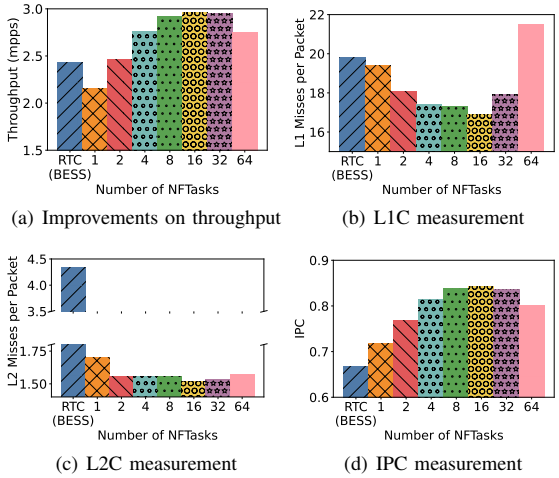
Redundant Prefetching Removing: For each action, the compiler will analyze the execution stream that leads to the NFAction and check whether the NFAction has already prefetched the NFState before the execution of previous NFAction. If it is prefetched before, it will remove the part of the NFState in the prefetching policy.



(a) Improvement on (downlink) throughputs

(b) L1-C measurement (c) L2-C measurement (d) IPC measurement

Fig. 10: Single-core performance improvement of UPF.



(a) Improvements on throughput (b) L1C measurement

(c) L2C measurement (d) IPC measurement

Fig. 11: Performance improvement of granular decomposition for NAT on GuNFu

Data Packing: To further minimize the cache footprint, we apply a data packing algorithm [27] to group the states most frequently accessed. We estimate how frequently two state variables are contemporaneously accessed based on the corresponding relationship between state variables and actions. For state variables that are accessed contemporaneously together, we put them into a single cache line to maximize the efficiency of prefetching. Considering SFC, because per-flow states of the consecutive network functions are highly correlated temporally, we put them in the same cache line if possible.

VII. Evaluation

GuNFu includes a programming model to provide Granular Decomposition that gives visibility on network function modules to the runtime and compiler, and also a runtime

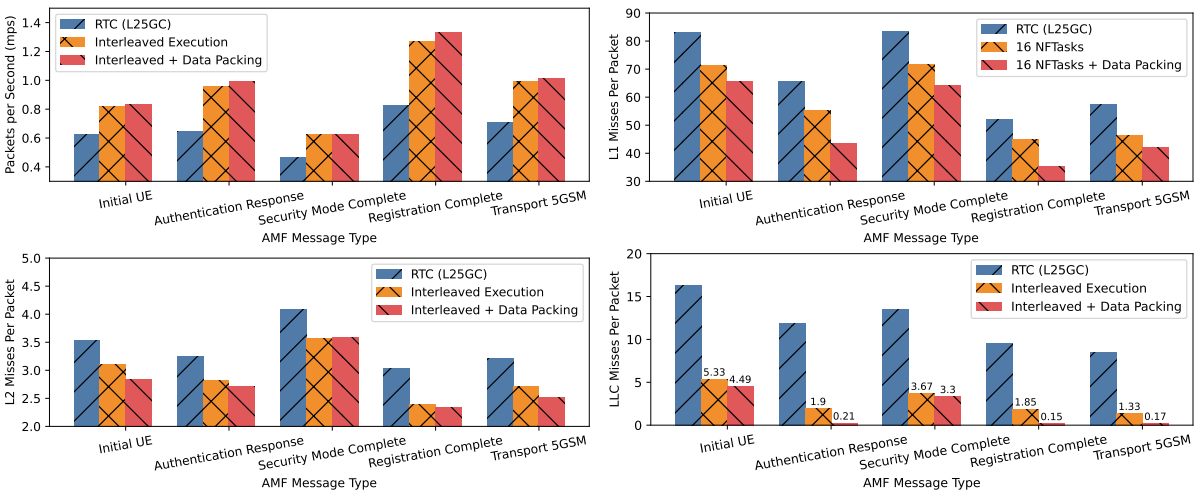


Fig. 12: The effectiveness of interleaved execution on granularly decomposed AMF on GuNFu

design supports flexible scheduling for interleaved execution streams for timely and automatic software prefetching. In the evaluation, our aim is to answer the following questions: (1) What is the performance gained from the interleaved function-stream execution model? (2) What is the performance gained from applying the compilation optimization techniques? (3) What is the scalability of GuNFu for the real-world workload?

Implementation: We have implemented a set of important and representative proof-of-concept network functions in our prototype system for evaluation purposes: User Plane Function (UPF) and Access and Mobility Management Function (AMF), Stateful Load Balancer (LB), Network Address Translation (NAT), Stateful Firewall (FW), Network Monitor (NM). We implement a prototype of UPF and AMF based on L25GC [17] and Free5GC [9].

A. *Impact of Execution Model*

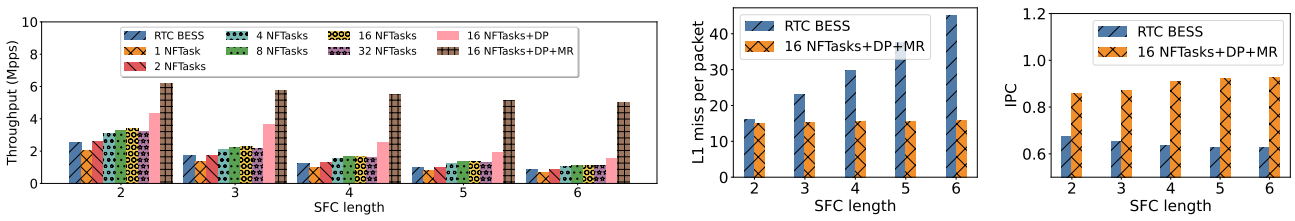
We use the same setup as the motivating experiments in §II to evaluate the improved performance on UPF and NAT of GuNFu. Evaluation test cases are taken from the MGW and NAT use cases from the Telco benchmarking System [25].

Evaluation on UPF: We evaluate the improvement in downlink throughput of our granularly decomposed UPF compared to the one that performs per-packet RTC from L25GC [17]. This evaluation measures the effects of the number of interleaved NFTasks and the number of second-level rules (PDR) on performance. The UPF achieves optimal performance with 16 or 32 interleaved NFTasks, but the performance degrades because of the cache contention if the number is 64. Figure 10(b) and Figure 10(d) measures the micro-architecture metrics for 16 interleaved NFTasks with a varying number of rules. For per-packet RTC model, the utilization of L1-C degrades (Figure 10(b)) when the number of second-level rules increases, since there are on average more pointer-chasing operations in tree structure lookups which are very likely leading to L1/L2 cache misses, hence worse performance. With the granularly decomposed version, the utilization of L1 cache is relatively stable even when the number of rules increases. This is because instead of stalling on the slow DRAM/LLC access, the NFTask immediately switches to another for useful work, and when it switches back, the tree node should already be in the L1/L2 cache.

Evaluation on NAT: We show NAT (Figure 11) because it is representative in terms of performance for other network functions such as LB, NM, FW, which perform a simple operation based on the per-flow state, which is only a few bytes. In GuNFu, similar to UPF, the optimal performance of NAT can be achieved using 16 NFTasks. The performance of using only one NFTask is worse than RTC due to the overhead of the scheduler, but the benefits of interleaved execution streams become apparent when the number is greater than 4. Similar to the one in UPF evaluation, the performance degrades if the number is 64 because the prefetched data may get pushed out from the cache if too many NFTasks contend for the resources. Based on the microarchitecture profiling metrics, L1 / L2 cache utilization of 16 NFTask is better than other alternations, leading to its better performance.

B. *Impact of Compiler Optimization*

To illustrate the benefits of compiler optimization of complex network functions with complex states and a longer processing chain, we use AMF and the composition of stateful network functions (LB, NAT, NM, FW) into an SFC as examples. We compare the performance of network functions with various length (2-6) using granular decomposed version with interleaved methods against those using the RTC model. For lengths greater than 4, we add FW to the SFC with different firewall policies. Additionally, we evaluate the performance improvement of applying the data packing and redundant matching elimination algorithms.



(a) Throughput, SFC length and number of interleaved NFTasks (b) L1-C measurement (c) Instructions per cycle (IPC)

Fig. 13: The impact of interleaved execution model, data packing, redundant matching removal on SFC.

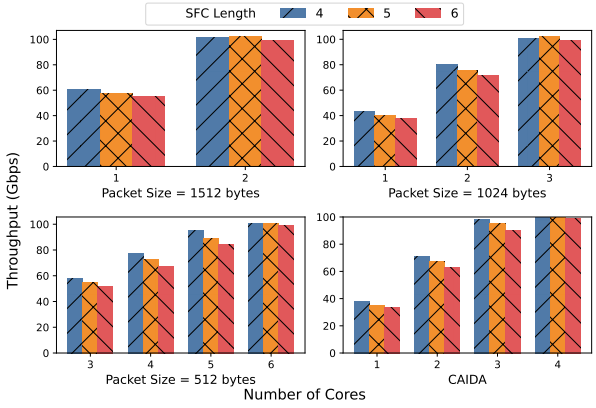


Fig. 14: Performance of SFC on GuNFu with 130K flows

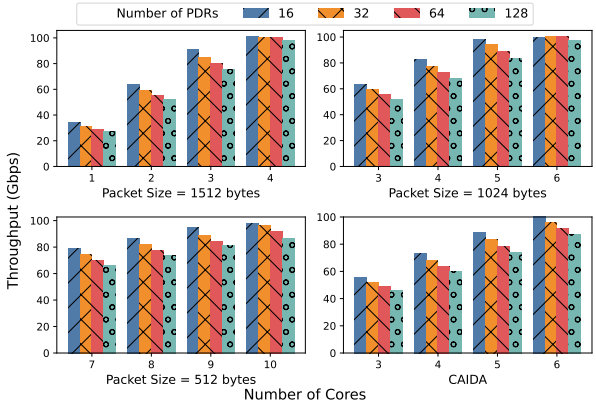


Fig. 15: Performance of UPF on GuNFu with 130K PFCP sessions

Evaluation on AMF: we compare our granularly decomposed AMF with 16 NFTasks (which is optimal) against the L25GC usign RTC model with the assumption of 2^{17} flows. Our implementation can achieve an improvement of 60% improvement of processing messages in initial registration (Figure 12(a)). Our implementation can achieve better throughput because it can have better utilization of the L1 / L2 cache by achieving fewer L1 / L2 cache misses per packet. Prefetching also significantly reduces the number of LLC cache misses per packet. In addition to the benefits of prefetching, our data packing techniques provide an additional 5% performance improvement because fewer cache lines needed for the same amount of state needed.

Evaluation on SFC: From Figure 13(a), the configuration of 16 interleaved NFTasks will lead to optimal performance. In addition, that data packing (DP) can improve performance significantly if we pack the temporally correlated per-flow states into a minimum number of cache lines. The redundant matching removal (MR) has a significant improvement (∼6x for SFC of length 6) in the performance on top of the interleaved methods and the data packing algorithm, since it eliminates all cache misses in pointer-chasing-heavy matching operations. The IPC measurement (Figure 13(c)) shows that with 16 interleaved NFTasks, data packing, and removal of redundant matching have superior efficiency because it is less influenced by memory stalling issues caused by cache misses. The evaluation shows that, thanks to the fine-grained data-code decoupling of the granular decomposition, the platform can leverage visibility into the network functions for efficient composition that improves cache utilization.

C. Scalability

To understand the scalability of our system, we study the performance of our system in a diverse traffic environment. In addition, we use CAIDA traffic as real-world traces to test our system.

Evaluation on SFC: in Figure 14, for all packet sizes, GuNFu scales linearly with the core number. It reduces the number of cores needed to reach the line rate or near line rate due to the effectiveness of prefetching, redundant matching removal, and data packing. In comparison, for the SFC of length 6, the implementation used by BESS can only achieve 18, 18, 17 , 20 Gbps respectively.

Evaluation on UPF: in Figure 15, for all packet sizes, GuNFu can scale linearly with the number of cores until it reaches the line rate. For packet size with 1512 bytes, the UPF can achieve line-rate or near line rate with 4 cores, 1024 bytes with 6 cores, 512 bytes with 10 cores, and CAIDA traffic with 6 cores. In comparison, for the case of 16 PDRs, the per-packet RTC model of UPF used by L25GC can only achieve 65, 64, 52, 41 Gbps, respectively, with the same number of cores. It shows that granularly decomposed NFs of GuNFu can be

effective when scaled to multiple cores and have better core utilization than monolithic implementations.

VIII. Related Works

More on Execution Model: Besides the works discussed in §II that use per-packet RTC model. There are execution models proposed to tackle inefficiency in higher abstraction layers than ours: the pipeline model that place different modules on different cores[28, 29, 30], the interleaved processing among networking modules and application modules [31], the adaptive batching technique for scheduling among modules[15], network function parallelism through module-level parallelism [32, 33, 34]. Within the module execution, they still apply per-packet RTC. In comparison, our method exploits the parallelism of the processing of different networking flows [35]. Their methods can be combined with ours to further improve the performance.

NF Programming Abstraction: The development of programming abstractions for network functions is crucial to enhancing their visibility within the hosting platform. For previous network function platforms, modularization based on functionality [8, 23, 29, 36, 37] yields significant benefits in terms of programmability, manageability, and scheduling when interfaced with traditional hardware. For the stand-alone network function implementations, the idea of modularization is also applied [9, 10, 11, 12]. Inheriting modular design, GuNFu further breaks down the module employing the principle of Granular Decomposition into a collection of NFAction and NFState with *fine-grained* decoupling.

Cache Optimization Applied in NF: Leveraging programming abstraction for transparent cache optimization is the key motivation for the design of interleaved function stream execution model. [13, 14, 26, 38] apply the software prefetching to accelerate the lookup data structure. This approach is also applied to network-intensive functionalities such as key value stores and databases [16, 39]. Another cache optimization is data packing [27, 40] to increase the utilization of cache lines by adjusting the placement of state variables. Through the programming model that we propose, both cache optimization methods are performed transparently.

Separation of Data and Code: The principle of separating data and code is a widely adopted design strategy on NF platforms [41, 42, 43, 44], improving fault tolerance and offering flexibility during scaling-in/out events. Within the mobile core, network functions such as 5G UDSF [45] have been developed to decouple data storage from computation, fortify fault tolerance, and enable adaptability in scaling scenarios [46]. Our NF model delineates the correspondence between *every* NFState and *every* NFAction with finer granularity, as opposed to the general per-flow state and NF. Our strategy overcomes the conventional trade-off between performance and reliability by improving L1 / L2 cache utilization, thereby improving performance without compromising reliability.

IX. Conclusion

The rapid expansion of virtualized network infrastructure, exemplified by the emerging 5G core network, necessitates innovative approaches to optimize per-core network function processing performance. This paper addresses the limitations of the prevalent per-packet RTC execution model using a novel interleaved function stream execution model to achieve more efficient state access. To address challenges on lack of visibility and inflexibility of scheduling of traditional modular network function implementations and platforms, we advocate a new programming model with the principle of Granular Decomposition for deep visibility. We design a lightweight runtime that leverages visibility to prefetch timely and precisely, and compilation optimizations that can further improve cache utilization. The evaluation results reveal significant performance gains, with an improvement ranging from 1.5 to 6 times per core throughput compared to traditional monolithic network functions with the per-packet RTC execution model. Our prototype of GuNFu is available at [47].

Acknowledgement

We appreciate the feedback by the anonymous reviewers. The research was supported in part by NSF under Grants CNS-2106771, CCF-2123987 and CNS-2321531.

References

[1] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, 2016.

[2] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, Renton, WA, April 2018. USENIX Association.

[3] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: building switch software at scale. pages 342–356, 08 2018.

[4] DPDK. http://dpdk.org/, 2017.

[5] Luigi Rizzo. netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[6] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. {mTCP}: a highly scalable user-level {TCP} stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, 2014.

[7] Berkeley Extensible Software Switch. http://span.cs.berkeley.edu/bess.html, 2017.

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

[9] free5gc/free5gc: Open source 5g core network base on 3gpp r15. https://github.com/free5gc/free5gc.

[10] a c-language open source implementation of 5g core and epc. https://github.com/open5gs.

[11] Openairinterface software alliance. https://github.com/openairinterface.
[12] srsran project open source ran. https://www.srslte.com/.
[13] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 97–108, 2013.
[14] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. Raising the bar for using GPUs in software packet processing. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 409–423, Oakland, CA, May 2015. USENIX Association.
[15] Tamás Lévai, Felicián Németh, Barath Raghavan, and Gabor Retvari. Batchy: batch-scheduling data flow graphs with service-level objectives. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 633–649, 2020.
[16] Onur Kocberber Ecocloud, Babak Falsafi Ecocloud, and Boris Grot. Asynchronous memory access chaining. *Proceedings of the VLDB Endowment*, 9, 2150.
[17] Vivek Jain, Hao-Tse Chu, Shixiong Qi, Chia-An Lee, Hung-Cheng Chang, Cheng-Ying Hsieh, KK Ramakrishnan, and Jyh-Cheng Chen. L25gc: a low latency 5g core network based on high-performance nfv platforms. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 143–157, 2022.
[18] Peng Zheng, Arvind Narayanan, and Zhi-Li Zhang. A closer look at nfv execution models. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, APNet '19, page 85–91, New York, NY, USA, 2019. Association for Computing Machinery.
[19] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, JR. Gerald Q. Maguire, and Rebecca Steinert. Metron: High-performance nfv service chaining even in the presence of blackboxes. *ACM Trans. Comput. Syst.*, 38(1–2), jul 2021.
[20] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proc. OSDI*, GA, 2016.
[21] Terabit ethernet. https://en.wikipedia.org/wiki/Terabit_Ethernet, 2023.
[22] Bess nat example. https://github.com/NetSys/bess/blob/master/core/modules/nat.cc, 2023.
[23] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proc. of HotMIddlebox*, 2016.
[24] Fastclick nat example. https://github.com/tbarbette/fastclick/blob/main/elements/flow/flowipnat.cc, 2023.
[25] Tamás Lévai, Gergely Pongrácz, Péter Megyesi, Péter Vörös, Sándor Laki, Felicián Németh, and Gábor Rétvári. The price for programmability in the software data plane: The vendor perspective. *IEEE Journal on Selected Areas in Communications*, 36(12):2621–2630, 2018.
[26] Hyunseok Chang, Fang Hao, TV Lakshman, Sarit Mukherjee, and Limin Wang. Cache-friendly ip reassembly network function. In *Proceedings of the Symposium on SDN Research*, pages 69–75, 2020.
[27] Trishul M Chilimbi, Bob Davidson, and James R Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 13–24, 1999.
[28] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In *Proc. CoNEXT*, 2016.
[29] Guyue Liu, Yuxin Ren, Mykola Yurchenko, KK 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*, pages 504–517, 2018.
[30] Ziyan Wu, Tianming Cui, Arvind Narayanan, Yang Zhang, Kangjie Lu, Antonia Zhai, and Zhi-Li Zhang. Granularnf: Granular decomposition of stateful nfv at 100 gbps line speed and beyond. *ACM SIGMETRICS Performance Evaluation Review*, 50(2):46–51, 2022.
[31] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. {IX}: a protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014.
[32] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. Nfp: Enabling network function parallelism in nfv. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 43–56, 2017.
[33] Yang Zhang, Bilal Anwer, Vijay Gopalakrishnan, Bo Han, Joshua Reich, Aman Shaikh, and Zhi-Li Zhang. Parabox: Exploiting parallelism for virtual network functions in service chaining. In *Proceedings of the Symposium on SDN Research*, pages 143–149, 2017.
[34] Sihao Xie, Junte Ma, and Jin Zhao. Flexchain: Bridging parallelism and placement for service function chains. *IEEE Transactions on Network and Service Management*, 18(1):195–208, 2020.
[35] Ziyan Wu, Yang Zhang, Wendi Feng, and Zhi-Li Zhang. Nflow and mvt abstractions for nfv scaling. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 180–189. IEEE, 2022.
[36] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.
[37] Shihabur Rahman Chowdhury, Haibo Bian, Tim Bai, Raouf Boutaba, et al. A disaggregated packet processing architecture for network function virtualization. *IEEE Journal on Selected Areas in Communications*, 38(6):1075–1088, 2020.
[38] Junji Takemasa, Yuki Koizumi, and Toru Hasegawa. Data prefetch for fast ndn software routers based on hash table-based forwarding tables. *Computer Networks*, 173:107188, 2020.
[39] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.
[40] Krishnendu Chatterjee, Amir Kafshdar Goharshady, Nastaran Okati, and Andreas Pavlogiannis. Efficient parameterized algorithms for data packing. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.
[41] Murad Kablan, Blake Caldwell, Richard Han, Hani Jamjoom, and Eric Keller. Stateless network functions. In *Proceedings of the 2015 ACM SIGCOMM workshop on hot topics in middleboxes and network function virtualization*, pages 49–54, 2015.
[42] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 299–312, 2018.
[43] Fabricio B. Carvalho, Ronaldo A. Ferreira, Italo Cunha, Marcos A.M. Vieira, and Murali K. Ramanathan. Dyssect: Dynamic scaling of stateful network functions. *Proceedings - IEEE INFOCOM*, 2022-May:1529–1538, 2022.
[44] Jingpu Duan, Xiaodong Yi, Shixiong Zhao, Chuan Wu, Heming Cui, and Franck Le. Nfvactor: A resilient nfv system using the distributed actor model. *IEEE Journal on Selected Areas in Communications*, 37(3):586–599, 2019.
[45] Unstructured data storage services, 2023. [Online; accessed 17-Aug-2023].
[46] Umakant Kulkarni, Amit Sheoran, and Sonia Fahmy. Towards a low-cost stateless 5g core. In *2022 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–2. IEEE, 2022.
[47] Gunfu prototype. https://github.com/GuNFuNFV, 2024.