Tapping into NFV Environment for Opportunistic Serverless Edge Function Deployment

Lu Zhang, Weiqi Feng, *Student Member, IEEE,* Chao Li, *Senior Member, IEEE,* Xiaofeng Hou, Pengyu Wang, Jing Wang, *Student Member, IEEE,* and Minyi Guo, *Fellow, IEEE*

Abstract—Even with Network Function Virtualization (NFV), many commodity network servers have spare cycles. Despite that they are small and irregularly occur, spare cycles are fit for deploying short-lived serverless computing functions at the network edge. In this work, we perform detailed analyses of the benefits and limitations of co-locating serverless functions on NFV-ready servers. We propose **NEMO**, a novel platform that enables efficient serverless edge function deployment in the NFV environment. NEMO can intelligently harvest spare cycles of network functions to warm up the serverless functions and speed up the function invocation in an agile manner. Besides, NEMO can judiciously manage the thread conflict in a resource-limited environment. We build a prototype of NEMO. Our thorough evaluations show that NEMO can harvest up to 41% spare cycles and achieve about 12.5~25X performance improvement compared with straightforward co-location.

Index Terms—Network function virtualization, serverless functions, spare cycle

1 Introduction

DRIVEN by the growing needs of serving various smart devices with low latency, the enthusiasm for edge computing continues unabated [13], [19]. Besides, the emergence of serverless architecture greatly facilitates the adoption of edge computing [8]. Today, serverless edge computing represents a more flexible way of edge application deployment (e.g., executing a function at the edge). It allows one to create and host applications that scale well, enabling a high degree of multi-tenancy on limited resources [8].

However, supporting serverless edge computing [2], [8] still requires non-trivial engineering work to set up the right infrastructure if we choose to deploy specific hardware at the network edge. One way to alleviate this issue is to take advantage of existing network facilities. According to a recent survey conducted by Emerson Network Power [16], over half of the companies believe that a great percentage (> 60%) of network facilities will also undertake the role of cloud in 2025. The results reveal the growing importance of network systems as more computing resources need to be pushed closer to end-users in the near future.

The rationale behind tapping into network systems for serverless edge computing is that they are generally virtualized commodity servers with CPU slacks (e.g., spare/idle CPU cycles of servers that are underutilized by network functions). Today, Network Function Virtualization (NFV) technology [9] is redefining the data transmission path for deploying network functions as software instead of specific hardware. It has been shown that servers that deploy network functions (NFV-ready servers) are not fully utilized at the network edge [23], [25]. In this paper we argue that it can be a ***win-win proposition*** to co-locate edge functions (EFs) with network functions (NFs). By taking advantage of the existing system infrastructure, we can avoid the upfront capital expenditure and the lengthy construction lead time of dedicated edge servers.

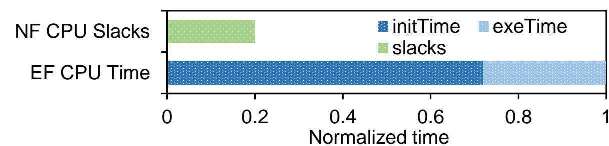


Fig. 1. CPU slacks of NFs normalized to EFs duration

A significant challenge associated with deploying edge functions on network servers is resource allocation. Due to the highly dynamic network traffic, spare cycles are not always available. In addition, with network packets transmitting quickly, spare cycles in the NFV environment are usually highly fragmented [25]. As shown in Figure 1, we measure the average CPU slacks of network functions and the duration of edge functions. There is a big difference between the average CPU slacks and the duration of serverless computing, which implies that the CPU slacks can not be easily utilized by EF without a resource harvesting mechanism. It is challenging to design a resource harvesting mechanism which can efficiently utilize spare cycles of NFV-ready servers for serverless edge computing, not to mention long-lived applications in the traditional cloud.

We explore the possibility of efficient EF execution while maintaining NF performance, rather than aggressively grab resources in the NFV environment. When functions are firstly invoked, the system needs to create and launch a container or a VM, and install the necessary libraries and dependencies, before the function itself can be executed. The initialization phase adds considerable latency for serverless functions. In this case, warming up functions [5], [7] using

• L. Zhang, W. Feng, C. Li, X. Hou, P. Wang, J. Wang, and M. Guo are with the department of Computer Science and Engineering, Shanghai Jiao Tong University, Dongchuan Rd 800, Minhang District, Shanghai 200240, China. E-mail: {luzhang, fengweiqi, xfhelen, wpybtw, jing618}@sjtu.edu.cn, {lichao, guo-my}@cs.sjtu.edu.cn.

Manuscript received April 19, 2005; revised August 26, 2015.
(Corresponding author: Chao Li.)

the harvested resources can improve the performance of EFs. Specifically, we intend to opportunistically utilize the spare cycles to warm up and speed up the execution of edge functions. Since co-locating edge functions with network functions may cause resource contention, it is crucial to keep a watchful eye on the execution of them.

We propose NEMO, a novel system platform that can gracefully manage edge functions that co-locate with network functions. NEMO[1] is short for “**N**etwork function and **E**dge function, **M**anaged and **O**ptimized together”. The basic idea behind NEMO is to harvest spare cycles in the edge NFV environment; meanwhile, NEMO judiciously utilizes the spare cycles to deploy serverless edge functions.

When edge computing requests arrive, NEMO can harvest spare cycles from NFs using a resource harvesting mechanism. NEMO speeds up the processing of edge functions in several different ways. First, it controls the *function pre-warm* thread periodically to utilize spare cycles of a NFV-ready server to warm up edge functions before they are invoked. Second, it uses a *function invocation* thread to intelligently manage the computing resources to boost the execution of serverless functions. Further, NEMO also monitors the two important threads and coordinates them. Doing so allows one to reap the benefits of edge functions while guaranteeing the QoS of NFs.

This paper makes the following key contributions:

- We demonstrate the availability of spare cycles on NFV-ready servers. We show that one can optimize serverless functions with resource harvesting.
- We propose NEMO, a platform that can gracefully manage serverless edge functions when co-locating with NFs. NEMO can speed up edge functions.
- We implement NEMO on a real server. We show that NEMO can harvest 41% spare cycles and achieve about 12.5∼25X performance improvement.

2 Background and Motivation

2.1 Serverless Edge Computing

Faced with a growing reliance on timely information access and efficient data processing, edge computing paradigms have become an evolving necessity [12], [13], [19]. By filtering and pre-processing data locally, one can quickly respond to related events while preventing extraneous data from saturating back-haul links to the cloud.

Emerging serverless computing model [5], [18] is well-suited for light-weight edge computing. In serverless computing frameworks, one can provide services through several functions. Serverless computing is a complement to the edge by enabling a high degree of multi-tenancy with reduced resource occupation [2], [8]. Serverless functions are typically short and can be used to increase the server utilization [18]. Besides, when serverless functions first invoke, creating a container and installing the necessary libraries can lead to long initialization overheads for deployment [5], [7]. Pre-warming function in advance can reduce the initialization overhead [5], [7]. The above features of serverless functions motivate us to improve the performance of serverless functions considering both function execution and function initialization.

1. NEMO is also named after Finding Nemo, a Pixar animated film depicting an energetic clownfish called Nemo. Clownfish and the sea anemone provide a good example of mutualism. Our design aims to gracefully manage NF and EF.

TABLE 1
Characterization Functions Description

Function	Description	Runtime
Network Function		
Firewall	Monitor packets based on rules	C++
Nat	Network address translator	C++
LoadBalancer	Balance packets of applications	C++
Edge Function		
markdown	Render Markdown to HTML	Python
img-resize	Resize image to icons	NodeJS
sentiment	Sentiment analysis of text	Python
ocr-img	Find text in images by OCR	NodeJS/binary

2.2 Virtualized Network Servers

An ideal candidate for hosting serverless edge functions is virtualized network servers that are placed near the user. Today, network function virtualization (NFV) [9] proposes to deploy network functions as software on commodity servers instead of specific hardware. NFV has many advantages such as agile network management and fast network deployment. Many prior works focus on high-performance packet processing in the data center NFV environment, [6], [15] or NF placement at the edge [14], [21], but very few on resource efficiency at the edge [23], [25].

Nevertheless, network servers can face low utilization issues [23], [24], [25], due to the limited and time-varying packet rate. The spare cycles in these NFV-ready servers are wasted if not being utilized properly. Co-locating resource-hungry IaaS applications with network functions is not a good choice. This is because spare cycles on NFV-ready servers are short and network functions have strict QoS requirements. Differently, the execution time of serverless functions is usually short. Thus, it has a great potential to utilize these spare cycles for cost-efficient edge computing.

Note that we do not argue that tapping into NFV-ready servers is the only way of deploying serverless edge computing. We intend to show that opportunistic resource sharing on network facilities provides an ***attractive alternative*** of serving edge user requests without incurring the significant upfront cost. As discussed in the following sections, utilizing these spare cycles is challenging in a highly dynamic environment. Smart resource harvesting/allocation is necessary to maintain the best design trade-off.

3 Characterization

In this section, we analyze the characteristics of spare cycles in the NFV environment. We use Click [10] to build our NFV platform and we choose Openwhisk [4] as our serverless computing platform. Table 1 shows the workload we evaluated, including three click-based network functions which are also used in prior works and four serverless functions presented in recent work [23]. Besides, to understand the behavior of serverless functions, we also implement a subset of functions in the Python Performance Benchmark Suite [20]. We present detailed evaluation platform in Section 5.

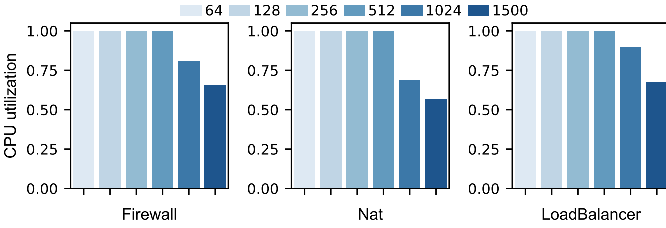


Fig. 2. Average CPU utilization of different network functions

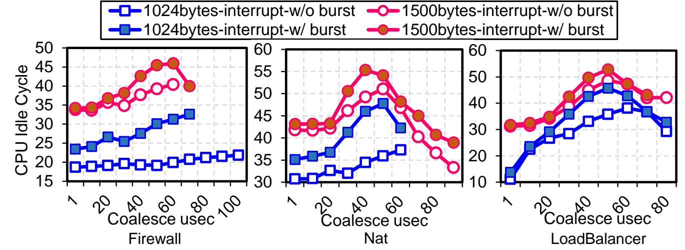


Fig. 3. CPU idle cycles of network functions with interrupt coalescing and packet burst processing

3.1 Understanding NFV Spare Cycles

We set up experiments to explore the characteristics of spare cycles on NFV-ready servers. On the sender server, we use `pktgen` to generate UDP packets with different sizes (64bytes, 128bytes, 256bytes, 512bytes, 1024bytes, and 1500bytes). We mainly focus on two mechanisms that may have a great impact on spare cycles: interrupt coalescing, and burst processing. Interrupt coalescing is a method preventing interrupts from being raised by a device until a specific amount of work is pending [17]. The Network Interface Card (NIC) uses direct memory access (DMA) to copy the packets to RAM and it raises an interrupt request (IRQ) to the CPU. With interrupt coalescing, one can reduce the number of interrupts posted to the processor [17]. Fewer interrupts bring more CPU slacks without compromising the network throughput. Additionally, NFs deployed in Click can further optimize CPU utilization by copying data in burst mode. When NFs are running, Click monitors and copies packets from the kernel space to the userspace. Upon receiving a packet from the sender, Click will immediately copy it to the user space by default. We can modify the number of packets to copy from the kernel to the userspace. With the coordination of interrupt coalescing and burst processing, more spare cycles can be harvested. To measure the CPU utilization of NFs, we monitor NFs every 100ms using the `perf` tool. We measure the CPU utilization for 300 times to report the average value.

3.1.1 Spare cycles on network servers

We use Linux performance counters to estimate the CPU utilization of network functions. Figure 2 shows the average CPU utilization of the evaluated network functions of different packet sizes. We obverse that, NFs occupy almost 100% CPU with packet size lower than 512bytes. This is mainly because NFs are saturated and the CPU core is always busy handling the arriving packets. In this situation, one can not harvest spare cycles. When the packet size becomes larger (i.e., 1024bytes and 1500bytes), idle CPU cycles appear. Therefore, in the following analysis, we mainly focus on large packet sizes when discussing spare cycle harvesting and allocation. Even though there are notable spare cycles in NFV-ready servers, they are too small to be utilized by serverless edge functions.

3.1.2 Implications of system management

To figure out whether we can get more spare cycles with appropriate system management, we experiment with different NIC settings. For each of the evaluated three network functions, the packet size of NFs of 1024bytes and 1500bytes. We first evaluate the impact of interrupt coalescing on CPU utilization and then the impact of burst processing. We modify the value of `rx-usecs` of NIC with `ethtool` tools to coalesce interrupts and the default configuration is `rx-usecs=1`. Larger `rx-usecs` means that there will be more interrupts coalesced. To satisfy the QoS requirement, we only select the `rx-usecs` that does not degrade NF's overall throughput in our experiment.

Interrupt coalescing can harvest more spare cycles from NFV-ready servers. As shown in Figure 3, the available spare cycles increase under interrupt coalescing. When the packet size is 1024bytes, interrupt coalescing brings extra spare cycles by 3% in Firewall, 6.5% in Nat, and 27% in LoadBalancer. In total, there are at most 22% and 40% in Firewall, 37% and 51% in Nat, 38% and 49% in LoadBalancer spare cycles for 1024bytes and 1500bytes, respectively.

Aggressive coalescing will not harvest more spare cycles but only increase the latency of NFs. Once the Linux network stack receives packets from NIC, it will trigger hardware interrupts and SoftIRQ to handle packets. However, interrupt coalescing can only coalesce the hardware interrupt. If we coalesce too many interrupts into one, the SoftIRQ will poll these interrupts, which may cost more CPU cycles. In Figure 3, the inflection point of the plot in Nat and LoadBalancer indicates a balance between hardware interrupt coalescing and SoftIRQ cost. We mainly focus on the region before reaching the inflection point.

In addition, one can also use burst processing to gain more spare cycles. When packets arrive, Click needs to copy packets from kernel space to network functions in userspace. By default, Click transfers one packet at a time, which may not take advantage of interrupt coalescing. Click can enable burst processing to copy multiple packets once. In this case, the number of calling functions to processing packets will decrease including the number of CPU switches between kernel and user mode. As shown in Figure 3, there are more spare cycles harvested by the coordination of interrupt coalescing and Click burst processing.

3.2 Serverless Edge Function Behaviors

We set up experiments to analyze the obstacle to efficient serverless function deployment on network servers. We invoke a group of serverless functions and record their initiation and execution time. Figure 4 shows the latency breakdown of serverless functions. We can see that the initialization time of most functions is even larger than their execution time. On average, initialization accounts for 60% latency of the evaluated serverless functions. The

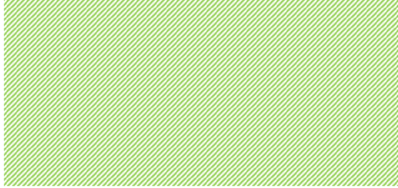


Fig. 4. The latency breakdown of EFs

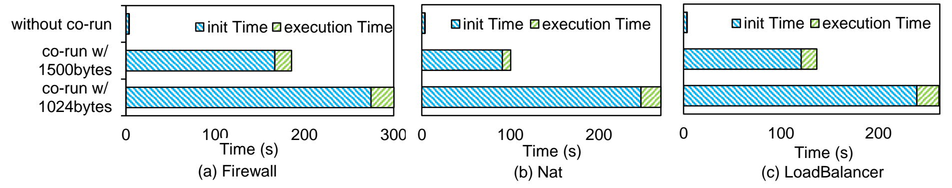


Fig. 5. The latency breakdown of serverless EF w/ and w/o co-locating with NFs

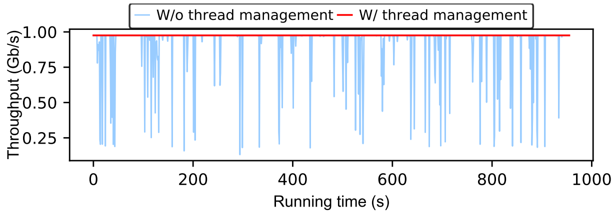


Fig. 6. The throughput of NFs w/ or w/o thread management

initialization cost of *json_loads* can even reach 98.6%. The large proportion of initialization time of serverless functions motivates us to warm up serverless functions in advance to reduce the cold-start cost.

We further conduct experiments to analyze the performance degradation of serverless edge functions while directly co-locating with NFs. We invoke four edge functions described in Table 1 in two scenarios. Firstly, EFs occupy all the CPU resources without co-locating with NFs. Secondly, EFs co-locate with NFs of different packet sizes without resource harvesting. The total initialization cost and execution cost of EFs is shown in Figure 5. Due to inadequate spare cycles of NFs, the initialization and execution time both increase greatly. The above results motivate us to harvest more spare cycles for deploying serverless functions.

3.3 Interference Issue of Colocation

When we deploy serverless edge functions on NFV-ready servers, there are mainly three types of threads running concurrently: *edge function pre-warm thread*, *edge function invocation thread*, and *network function thread*. Simply speeding up edge function invocation and adopting function pre-warm using harvested spare cycles may cause thread conflict due to inadequate resources.

To motivate the necessity of smart thread management, we compare the throughput of NFs with the same configuration (offered load and window size) in two situations: with thread management and without thread management. Figure 6 shows the throughput (Gb/s) in different situations. with thread management, the throughput of NF is stable which guarantees the QoS of NFs (1Gb/s). It is also evident that the throughput of NFs fluctuates severely (even from 1Gb/s to 0.25Gb/s) without thread management. The reason of throughput degradation is that NFs have insufficient resources to process packets due to thread conflict.

Using interrupt coalescing and burst processing, we can harvest more spare cycles without any throughput degradation for NFs. However, resource harvesting may affect the latency of NFs. While our design is throughput-centric, it is important to guarantee that the latency overhead of NFs is within acceptable limits. We model and calculate the latency overhead, which is bounded by the value of `rx_usec`. If the packets follow a uniform distribution, the interrupt coalescing method will increase the latency of NFs by $rx_usec/2$ on average. In other cases like exponential distribution, the latency overhead is expected to be less than rx_usec. In our experiments, the value of rx_usec is usually small than 60 us.

Summary of Design Consideration: The above study shows that there are valuable spare cycles in the NFV environment that can be utilized by serverless edge functions. In particular, with interrupt coalescing and packet burst processing, more spare cycles can be harvested. The benefits of resource harvesting are two-fold: 1) one can speed up the invocation of edge functions; 2) it also accelerates the warm-up process of popular functions. In addition, to avoid thread conflict under limited resources, it is necessary to monitor workload running status and manage them accordingly to guarantee the throughput of the network functions.

4 NEMO Design

In this section, we propose *NEMO*, a novel platform that makes the best use of network facilities and enhances the performance of serverless edge functions.

4.1 Overview of NEMO

Figure 7 gives an overview of NEMO. Our system is mainly composed of two parts. The slack harvester harvests spare cycles according to the feature of NFs. Then the function controller can utilize the harvested spare cycles of NFV-Ready servers to optimize serverless function execution from the following aspects: 1) speeding up the function invocation, 2) accelerating the function pre-warm process. 3) coordinating the threads of both function invocation and function pre-warm for ensuring the QoS of NFs.

Particularly, the thread scheduling between NFs and EFs depends on the Linux scheduler. To guarantee the performance of NF, we configure NF threads with the highest priority. Thus, EFs can get access to the CPU cycles when the platform has spare cycles. Once packets arrive, network functions will get all the CPU to process packets. In this paper, EFs processed on NEMO do not have strict deadlines and if there is a burst of EF requests, NEMO will only process EFs that it can safely accommodate.

4.2 Slack Harvester

In this work, we design a slack harvester to gain more spare cycles when edge functions invoke. The core of slack harvester is a resource management handler called *changeMode*

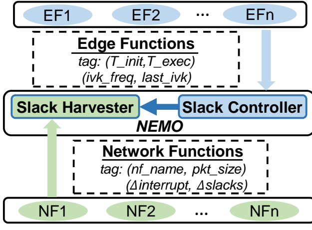


Fig. 7. The overview of NEMO

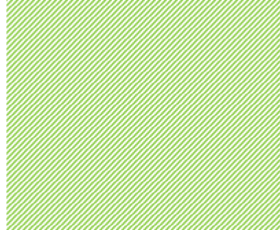


Fig. 8. The slack harvester

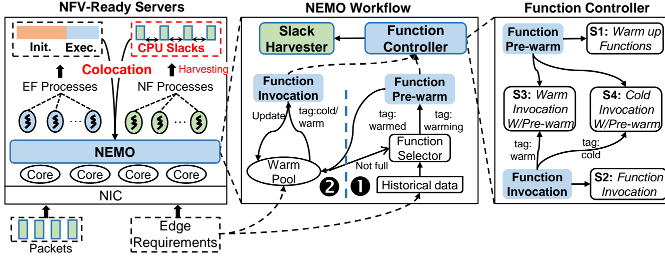


Fig. 9. The workflow of NEMO's function controller

as shown in Figure 8. The *changeMode* handler controls the status of interrupt coalescing and burst processing to harvest cycles. Specifically, it controls the interrupt coalescing using `ethtool rx-usecs` to adjust the size of interrupt coalescing in NIC according to the tag of running NFs. In the meantime, it also transmits messages to the Click receiver to enable burst packet processing through shared memory which causes negligible overhead.

The spare cycles of NFV-ready servers depend on the NF type and packet size. The slack harvester profiles NFs and tags them with (*nf_name*, *pkt_size*) for determining the effectiveness of resource harvesting. Since NFs keep running on the server, it is easy to gather their runtime statistics.

4.3 Function Controller

The goal of NEMO's function controller is to make the best use of the harvested spare cycles for improving edge function performance. The detailed workflow is shown in Figure 9. It maintains two components: ❶ *Function Pre-warm* thread warms up functions in advance to alleviate the cold start latency, and ❷ *Function Invocation* thread invokes serverless functions upon request on the NFV-ready servers. Note that it may cause thread conflict between function pre-warm and function invocation if they run concurrently. In this case, the function controller further manages the contention of different threads.

1) Function Pre-warm: Since the initiation time hinders fast execution of serverless functions, function pre-warm is necessary to prepare the runtime for serverless functions to reduce the initialization cost. NEMO initializes functions in advance to reduce the possibility of cold start using the harvested cycles. NEMO chooses functions based on historical data when warming up containers. The historical data we use in NEMO contains $b_warm, ivk_freq, last_ivk$. Specifically, b_warm is defined as the initialization time divided by the total duration of each function, ivk_freq is the invocation frequency of functions, and $last_ivk$ indicates the last invocation time. Using these historical data, we can

Algorithm 1: Preemptive Thread Management

Input: coming actions $action$, the tag of actions tag_{action}, function pre-warm thread $T_{warming}$
while *True* **do**
  if tag_{action} *is warmed in warm_pool* **then**
    // *S3*
    stop $T_{warming}$;
    run function invocation thread;
    continue $T_{warming}$
  else if tag_{action} *is warming in warm_pool* **then**
    // *S4-1*
    wait for $T_{warming}$ finished;
    invoke $action$;
    update $action$ in warm_pool;
  else if tab_{action} *is not in warm_pool* **then**
    // *S4-2*
    stop $T_{warming}$;
    run function invocation thread;
    continue $T_{warming}$
  else
    // *No thread conflict - S1 or S2*
    run function invocation or function pre-warm thread
  end
  wait for $action$;
end

calculate the pre-warming possibility of each function and choose functions with the biggest possibility to pre-warm.

The function pre-warm is implemented as a daemon thread for checking the system periodically. NEMO maintains a warm pool to store the warmed functions. The size of the warm pool is limited by memory. NEMO monitors the status of the warm pool periodically and it stops functions that are not invoked for a certain period. If the warm function pool is not full, NEMO will choose proper functions to warm based on the historical data. When a function is warming, NEMO enables resource harvesting to speed up the warming process. A function is tagged as $warming$ when the pre-warm thread is running and $warmed$ when the warming process finishes.

2) Function Invocation: Once an edge computing request arrives, NEMO enables the resource harvesting to invoke edge functions which can provide more spare cycles. Doing so allows one to speed up the invocation process.

Given a warm function pool, NEMO first checks whether the pool contains the requested function. If it is found in the warm pool, NEMO will update the information (last invocation time) of the function after this invocation. Otherwise, if the pool is full, NEMO will have to update the warm pool and delete the function whose last invocation time is the oldest. Then, the warm pool will add new functions to the warm pool. After each invocation, NEMO will update the historical data and disable resource harvesting for minimizing the impact on network functions.

3) Thread Management: Without appropriate management, the function pre-warm thread and the function invocation thread may cause thread conflict due to resource scarcity. To avoid the thread conflict, the function controller intelligently coordinates the function pre-warm and func-

tion invocation thread. The function controller manages four states of NEMO as shown in Figure 9: S1) only function pre-warm; S2) only function invocation; S3) warm function invocation with pre-warm thread conflict and S4) cold function invocation thread with pre-warm thread conflict.

The function controller just performs *function pre-warm* and *function invocation* respectively in the state S1 or S2 due to no thread conflict. However, in states S3 and S4, NEMO needs to carefully handle the thread conflict for the QoS of NFs. We propose *Preemptive Thread Management* to avoid the thread conflict, as detailed in Algorithm 1. Since function pre-warm is only an optimization for function invocation, we give high priority to the function invocation thread. Thus, in S3, NEMO needs to guarantee the performance of warm function invocation by pausing the function pre-warm thread. After the invocation, the function controller continues processing the function pre-warm thread. In this way, it also avoids the memory conflict between the function pre-warm and function invocation thread.

Additionally, NEMO handles S4 in two ways: 1) The coming function is contained in the warm pool but tagged $warming$. In other words, the function pre-warm thread and function invocation thread process the same function. In this case, the function invocation thread waits for the function pre-warm thread. Afterward, the function invocation thread invokes the function with a warm start. 2) The incoming function is not in the warm pool. In this state, the function controller will pause the pre-warm thread immediately to ensure the processing of function invocation thread. Meanwhile, it strives to harvest more spare cycles to reduce invocation latency. After invocation, the function pre-warm thread continues processing.

5 Evaluation

5.1 Evaluation Methodology

We evaluate NEMO on our proof-of-concept system. We implement NEMO in C++ with 1649 SLOC. NEMO's configuration file is in JSON format and NEMO parses it to set up the whole system. Users can change the size of the container pool, add/delete edge functions, and the path of the log file by editing the configuration file.

EF Layer: We use the open-source serverless platform named OpenWhisk to run EFs. The EF is from prior work [18] as shown in Table 1. Meanwhile, we rewrite and deploy a subset of Python Performance Benchmark Suite in Openwhisk. We also modify the source code of Openwhisk to bind serverless functions running on the same core with NFs. In Openwhisk, we allocate 2048M memory (the default configuration) for processing edge functions. Openwhisk will allocate 256M memory for each function. As a result, our server node can run at most 8 functions at the same time. We configure the size of the pre-warm function pool which can support at most 6 functions to be warmed. The remaining 512M memory space is used to handle functions with cold invocations. Our light-weight thread management of NEMO runs on a single core with little resource consumption and it will not add contention to the system.

NF Layer: We use Click [10] as our NFV platform and we slightly modifies it by adding 54 SLOC. To adjust the interrupt coalescing rate, NEMO invokes `ethtool` utility to reconfigure the NIC. We change the `rx_usecs` parameter of NIC, which controls the time to delay an interrupt after a packet arrives. In addition, NEMO has a shared memory buffer with Click to enable/disable burst packet processing. There are several works using Click router to implement their NFV platforms such as ClickNF [6] and ClickOS [15]. We use a more conservative Click platform to emulate the NFV environment since we focus on analyzing the resource slacks of underutilized NFV servers at the edge. As shown in Table 1, we choose three click-based NFs. We deploy each NF on a dedicated core by setting the core affinity.

Server Specification: We conduct our experiments on a two-socket Intel Xeon Sliver server with 40 cores. The server has 13.75MB of L3 cache and 64GB of 2666MHz DDR4 RAM. By using `pktgen` (kernel mode), the network sender server can generate packets with a maximum rate of 1Gb/s. We use `Perf` to collect the runtime statistics of NFs and EFs.

Additional Remarks of Evaluation: When we evaluate the overall benefits of NEMO, we design a workload that consists of a sequence of edge functions. The workload consists 10 functions (4 functions from Table 1 and 6 functions from python suite.) and the functions are invoked in an interval following Poisson distribution. We set 3 seconds as the time interval to pre-warm functions. All evaluations include the performance impact of control operations/events like invoking `ethtool`.

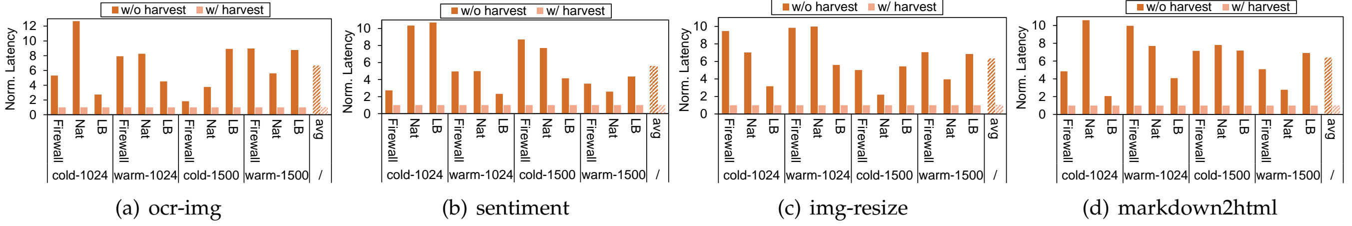
5.2 Effectiveness of NEMO Design

1) Effectiveness of Resource Harvesting: We first evaluate the effectiveness of resource harvesting in NEMO. We colocate four EFs with three NFs and generate two kinds of packet sizes: 1024Bytes and 1500Bytes for each NF. The latency is normalized to the latency without resource harvesting in different scenarios.

Figure 10 shows the normalized latency of EFs in four different scenarios. We observe that resource harvesting brings 1.83∼12.7X performance improvement. For all the situations, it can achieve 6.2x performance improvement on average for edge functions. Note that the availability of spare cycles highly depends on the type of network functions. For example, the performance improvement of EFs co-locating with LoadBalancer is less than the others since fewer spare cycles can be harvested from LoadBalancer.

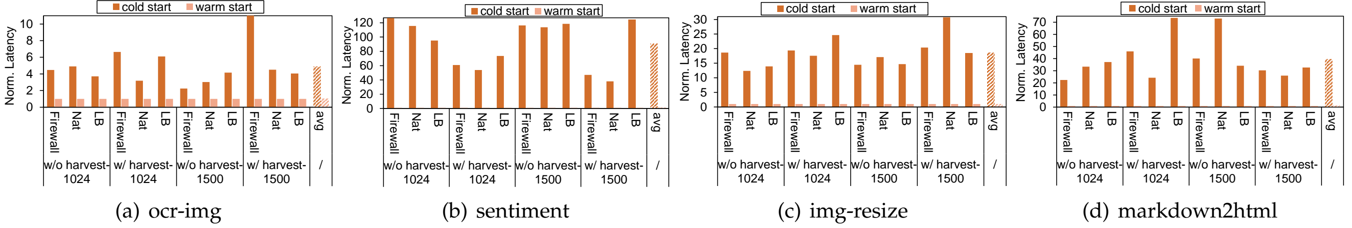
2) Effectiveness of Function Pre-warm: NEMO judiciously warms up containers in advance. Doing so alleviates the initialization overhead which may jeopardize the performance of serverless edge functions. To illustrate the benefits of NEMO's function pre-warm ability, we compare the latency of EF warm invocation with EF cold invocation under different network functions.

We record the latency of edge functions in four different scenarios. Figure 11 shows that all EFs can benefit from warm invocation in different scenarios. Specifically, the actual speedup varies with different EF behaviors. For example, the latency of *ocr-img* is only reduced by about 3∼11X. Differently, The function *sentiment* achieves 47∼127X performance improvement with warm invocation. The reason for such a significant improvement is that the initialization overhead of *sentiment-analysis* accounts for a large portion of its overall latency.



(a) ocr-img (b) sentiment (c) img-resize (d) markdown2html

Fig. 10. The normalized latency of serverless functions when co-locating with NFs w/ or w/o resource harvesting (cold-1024 means functions are cold invocation and packet size of NF is 1024B)



(a) ocr-img (b) sentiment (c) img-resize (d) markdown2html

Fig. 11. The normalized latency of serverless functions when co-locating with NFs w/ or w/o the warm function pool (w/o harvest-1024 means no resource harvesting and packet size of NF is 1024B)

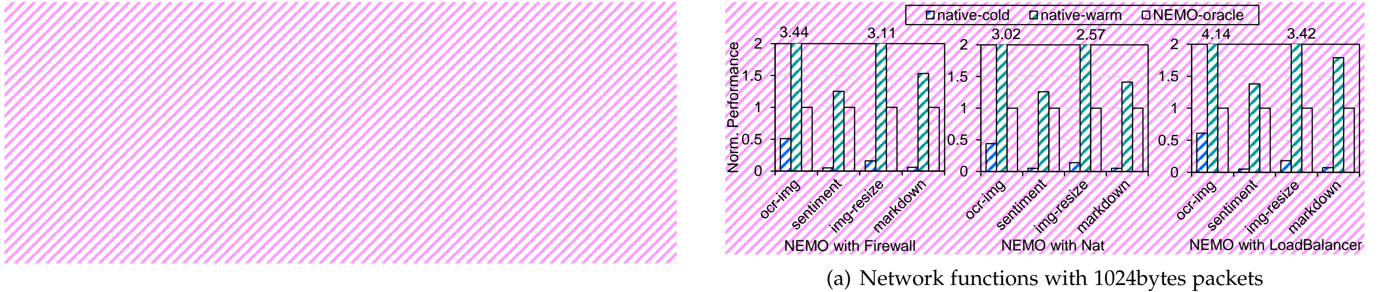
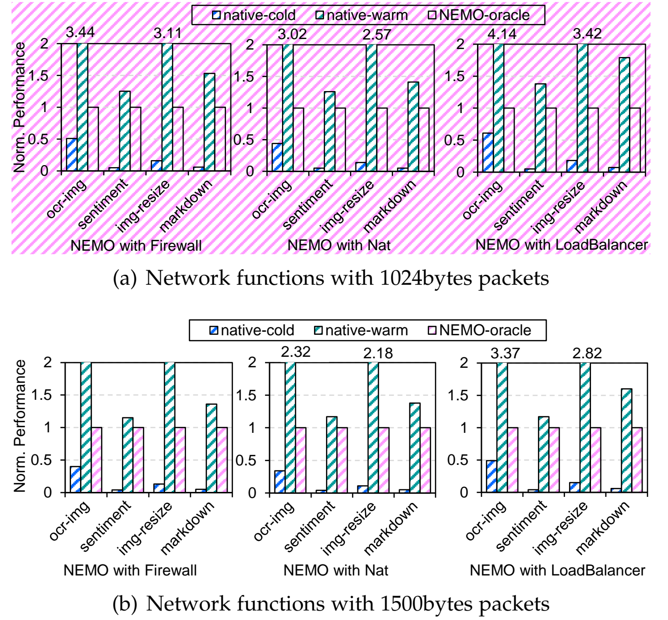


Fig. 12. The normalized performance of functions w/ or w/o NEMO. Best Case: w/ NEMO; Worst Case: w/o NEMO



(a) Network functions with 1024bytes packets

(b) Network functions with 1500bytes packets

Fig. 13. The normalized performance of NEMO and native invocations

3) Joint Effectiveness of Resource Harvesting and Function Pre-Warm: NEMO seeks a synergism of resource harvesting and function pre-warm, and therefore it can achieve significant performance improvement. To demonstrate this, we further evaluate how much speedup NEMO can obtain compared to a straightforward co-location scheme.

In Figure 12, We treat direct co-location (w/o NEMO) as our baseline. It relies on the OS to coordinate EF and NF and therefore shows the worst performance; We observe 16x $\sim$ 628x performance improvement when both resource harvesting and warm invocation are enabled (i.e., w/ NEMO).

5.3 Comparison with Native Processing

Co-locating EFs with NFs unavoidably worsens the performance due to resource contention. To demonstrate the performance impact of NEMO on edge functions, we compare NEMO with the native invocations. In Figure 13, native-cold and native-warm refer to a scenario that the edge function is executed on dedicated hardware without NF interference. NEMO-oracle means that functions are processed with warm start and resource harvesting while co-locating with NFs. All results are normalized to NEMO-oracle.

Since co-location will worsen the performance of EFs, NEMO-oracle can hardly achieve higher performance than native-warm. However, NEMO-oracle greatly outperforms native-cold invocation. NEMO-oracle lowers the performance by 1.15x $\sim$ 4.14x compared with native-warm but obtains 1.64x $\sim$ 25x performance improvement compared to native-cold. This demonstrates the advantage of our design.

5.4 Performance of Mixed Edge Function Stream

When edge computing requests arrive, edge functions may not always be warm invocations. We utilize the workload described in Section 5.1 to evaluate the mixed function invocations. We compare NEMO with two schemes: direct co-location (EFs directly co-locates with NFs without NEMO) and native EF (EFs invokes without co-location).

Figure 14 shows the performance of mixed edge functions. All the results are normalized to NEMO. Compared

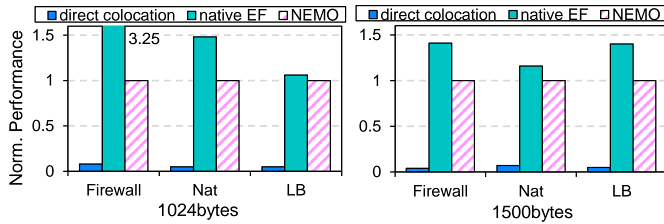


Fig. 14. The overall performance of EFs with native processing and naive co-location normalized to NEMO

with direct co-location, NEMO improves the performance of edge functions by 12.5x $\sim$ 25X. The performance of native edge functions is higher than NEMO as a result of unlimited, abundant resources. The average latency of EFs on NEMO is only increased by 60% compared to native invocations. In a word, NEMO can greatly improve the performance of EFs by opportunistically utilizing the spare resources of NFV-ready servers.

6 Related Work

1) Serverless Computing Platform: Serverless computing is popular at the edge [2], [8]. It has been adopted in various IoT scenarios [1], [3]. To achieve better performance, prior works [5], [7] focus on addressing the cold start problem. Besides, the resource scheduling of serverless functions [22] are also proposed. However, no work has been done in terms of deploying serverless functions on network facilities. Our work aims to fill this void.

2) System Optimization for NFs: Many prior works of NFV aim to provide high performance and design flexibility [6], [15]. There are also works considering the resource utilization of NFV. For example, NFVnice [11] benefit from running multiple NFs on a single core. HyperNF [24] aims at maximizing server performance when concurrently running large numbers of NFs. Moreover, there are also prior works [14], [21] on optimizing network functions at the edge. However, these works mainly emphasize the NF itself. Differently, we focus on opportunistically utilizing the spare cycles on network facilities to host serverless EFs in an efficient, cost-effective manner.

3) Towards NF/EF Colocation: There are some prior works on performing edge computing in the NFV environment [23], [25]. They propose to jointly manage both NFs and EFs. However, these works focus on orchestration for applications. Wang et al. [23] characterize the resource usage patterns of edge NFV which only shows the possibility of deploying EFs on NFV-ready servers. The most relevant work is EdgeMiner [25] which can harvest the idle CPU cycles in a DPDK-based NFV environment. Different from EdgeMiner, NEMO focuses on serverless edge function, which is more suitable for deploying edge applications. NEMO's unique management strategy allows one to better utilize the spare cycles.

7 Conclusion

Looking ahead, the distributed network facilities are an ideal infrastructure for ubiquitous edge computing. The spare cycles on NFV-ready servers at the edge are untapped opportunities for supporting serverless edge functions. In this paper, we demonstrate that the fragmented resources can be harvested for warming up edge functions and speeding up their invocation process. We build NEMO, a system framework tailored to the behaviors of serverless EFs as well as NFs. NEMO shows 12.5$\sim$25X performance improvement compared with direct co-location and only worsen the performance by 60% on average compared to an oracle case. Since NEMO focuses on spare CPU cycles that are almost free, we expect that it will encourage and facilitate the adoption of edge computing applications.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No.61972247). We thank all the anonymous reviewers for their valuable feedback.

References

[1] L. Ao *et al.*, "Sprocket: A serverless video processing framework," in *SoCC*, 2018.
[2] E. de Lara *et al.*, "Hierarchical serverless computing for the mobile edge," in *SEC*, 2016.
[3] S. Fouladi *et al.*, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *NSDI*, 2017.
[4] A. S. Foundation, "Apache openwhisk," https://bit.ly/36veTUX.
[5] A. Fuerst *et al.*, "Faascache: keeping serverless computing alive with greedy-dual caching," in *ASPLOS*, 2021.
[6] M. Gallo *et al.*, "Clicknf: a modular stack for custom network functions," in *ATC*, 2018.
[7] A. U. Gias *et al.*, "Cocoa: Cold start aware capacity planning for function-as-a-service platforms," in *MASCOTS*, 2020.
[8] A. Hall *et al.*, "An execution model for serverless functions at the edge," in *IoTDI*, 2019.
[9] B. Han *et al.*, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, 2015.
[10] E. Kohler *et al.*, "The click modular router," *TOCS*, 2000.
[11] S. G. Kulkarni *et al.*, "Nfvnice: Dynamic backpressure and scheduling for nfv service chains," in *SIGCOMM*, 2017, pp. 71–84.
[12] C. Li *et al.*, "Towards sustainable in-situ server systems in the big data era," in *ISCA*, 2015.
[13] C. Li *et al.*, "Edge-oriented computing paradigms: A survey on architecture design and system management," *CSUR*, 2018.
[14] M. Li *et al.*, "Finedge: A dynamic cost-efficient edge resource management platform for nfv network," in *IWQoS*, 2020.
[15] J. Martins *et al.*, "Clickos and the art of network function virtualization," in *NSDI*, 2014.
[16] E. N. Power, "Data center 2025: Exploring the possibilities," Available: https://Vertiv.com/DC2025, 2014.
[17] R. Prasad *et al.*, "Effects of interrupt coalescence on network measurements," in *PAM*, 2004.
[18] M. Shahrad *et al.*, "Architectural implications of function-as-a-service computing," in *MICRO*, 2019.
[19] W. Shi *et al.*, "Edge computing: Vision and challenges," *IoT-J*, 2016.
[20] V. Stinner, "The python performance benchmark suite, version 0.7.0," Available: https://pyperformance.readthedocs.io.
[21] T. Subramanya *et al.*, "Machine learning-driven service function chain placement and scaling in mec-enabled 5g networks," *Computer Networks*, 2020.
[22] A. Suresh *et al.*, "Fnsched: An efficient scheduler for serverless functions," in *WoSC*, 2019.
[23] J. Wang *et al.*, "Architectural and cost implications of the 5g edge nfv systems," in *ICCD*, 2019.
[24] K. Yasukata *et al.*, "Hypernf: Building a high performance, high utilization and fair nfv platform," in *SoCC*, 2017.
[25] L. Zhang *et al.*, "Characterizing and orchestrating nfv-ready servers for efficient edge data processing," in *IWQoS*, 2019.