E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers

Ming Liu
University of Washington

Simon Peter
The University of Texas at Austin

Arvind Krishnamurthy
University of Washington

Phitchaya Mangpo Phothilimthana
Google

Abstract
We investigate the use of SmartNIC-accelerated servers to execute microservice-based applications in the data center. By offloading suitable microservices to the SmartNIC’s low-power processor, we can improve server energy-efficiency without latency loss. However, as a heterogeneous computing substrate in the data path of the host, SmartNICs bring several challenges to a microservice platform: network traffic routing and load balancing, microservice placement on heterogeneous hardware, and contention on shared SmartNIC resources.

We present E3, a microservice execution platform for SmartNIC-accelerated servers. E3 follows the design philosophies of the Azure Service Fabric microservice platform and extends key system components to a SmartNIC to address the above-mentioned challenges. E3 employs three key techniques: ECMP-based load balancing via SmartNICs to the host, network topology-aware microservice placement, and a data-plane orchestrator that can detect SmartNIC overload. Our E3 prototype using Cavium LiquidIO SmartNICs shows that SmartNIC offload can improve cluster energy-efficiency up to 3× and cost efficiency up to 1.9× at up to 4% latency cost for common microservices, including real-time analytics, an IoT hub, and virtual network functions.

1 Introduction
Energy-efficiency has become a major factor in data center design [81]. U.S. data centers consume an estimated 70 billion kilowatt-hours of energy per year (about 2% of total U.S. energy consumption) and as much as 57% of this energy is used by servers [23,75]. Improving server energy-efficiency is thus imperative [18]. A recent option is the integration of low-power processors in server network interface cards (NICs). Examples are the Netronome Agilio-CX [60], Mellanox BlueField [52], Broadcom Stingray [14], and Cavium LiquidIO [16], which rely on ARM/MIPS-based processors and on-board memory. These SmartNICs can process microsecond-scale client requests but consume much less energy than server CPUs. By sharing idle power and the chassis with host servers, SmartNICs also promise to be more energy and cost efficient than other heterogeneous or low-power clusters. However, SmartNICs are not powerful enough to run large, monolithic cloud applications, preventing their offload.

Today, cloud applications are increasingly built as microservices, prompting us to revisit SmartNIC offload in the cloud. A microservice-based workload comprises loosely coupled processes, whose interaction is described via a dataflow graph. Microservices often have a small enough memory footprint for SmartNIC offload and their programming model efficiently supports transparent execution on heterogeneous platforms. Microservices are deployed via a microservice platform [3–5, 41] on shared datacenter infrastructure. These platforms abstract and allocate physical datacenter computing nodes, provide a reliable and available execution environment, and interact with deployed microservices through a set of common runtime APIs. Large-scale web services already use microservices on hundreds of thousands of servers [41,42].

In this paper, we investigate efficient microservice execution on SmartNIC-accelerated servers. Specifically, we are exploring how to integrate multiple SmartNICs per server into a microservice platform with the goal of achieving better energy efficiency at minimum latency cost. However, transparently integrating SmartNICs into microservice platforms is non-trivial. Unlike traditional heterogeneous clusters, SmartNICs are collocated with their host servers, raising a number of issues. First, SmartNICs and hosts share the same MAC address. We require an efficient mechanism to route and load-balance traffic to hosts and SmartNICs. Second, SmartNICs sit in the host’s data path and microservices running on a SmartNIC can interfere with microservices on the host. Microservices need to be appropriately placed to balance network-to-compute bandwidth. Finally, microservices can contend on shared SmartNIC resources, causing overload. We need to efficiently detect and prevent such situations.

We present E3, a microservice execution platform for SmartNIC-accelerated servers that addresses these issues. E3 follows the design philosophies of the Azure Service Fabric microservice platform [41] and extends key system compo-
Microservices have become a critical component of today’s data center footprint. Microsoft reports running microservices 24/7 on over 160K machines across the globe, including Azure SQL DB, Skype, Cortana, and IoT suite [41]. Google reports that Google Search, Ads, Gmail, video processing, flight search, and more, are deployed as microservices [42]. These microservices include large and small data and code footprints, long and short running times, billed by run-time and by remote procedure call (RPC) [29]. What unifies these services is their software engineering philosophy.

We make the following contributions:

- We show why SmartNICs can improve energy efficiency over other forms of heterogeneous computation and how they should be integrated with data center servers and microservice platforms to provide efficient and transparent microservice execution (§2).

- We present the design of E3 (§3), a microservice runtime on SmartNIC-accelerated server systems. We present its implementation within a cluster of Xeon-based servers with up to 4 Cavium LiquidIO-based SmartNICs per server (§4).

- We evaluate energy and cost-efficiency, as well as client-observed request latency and throughput for common microservices, such as a real-time analytics framework, an IoT hub, and various virtual network functions, across various homogeneous and heterogeneous cluster configurations (§5). Our results show that offload of microservices to multiple SmartNICs per server with E3 improves cluster energy-efficiency up to 3× and cost efficiency up to 1.9× at up to 4% client-observed latency cost versus all other cluster configurations.

2 Background

Microservices simplify distributed application development and are a good match for low-power SmartNIC offload. Together, they are a promising avenue for improving server energy efficiency. We discuss this rationale, quantify the potential benefits, and outline the challenges of microservice offload to SmartNICs in this section.

2.1 Microservices

Microservices have become a critical component of today’s data center footprint. Microsoft reports running microservices 24/7 on over 160K machines across the globe, including Azure SQL DB, Skype, Cortana, and IoT suite [41]. Google reports that Google Search, Ads, Gmail, video processing, flight search, and more, are deployed as microservices [42]. These microservices include large and small data and code footprints, long and short running times, billed by run-time and by remote procedure call (RPC) [29]. What unifies these services is their software engineering philosophy.

Microservices use a modular design pattern, which simplifies distributed application design and deployment. Microservices are loosely-coupled, communicating through a set of common APIs, invoked via RPCs [87], and maintain state via reliable collections [41]. As a result, developers can take advantage of languages and libraries of their choice, while not having to worry about microservice placement, communication mechanisms, fault tolerance, or availability.

Microservices are also attractive to datacenter operators as they provide a way to improve server utilization. Microservices execute as light-weight processes that are easier to scale and migrate compared with a monolithic development approach. They can be activated upon incoming client requests, execute to request completion, and then swapped out.

A microservice platform, such as Azure Service Fabric [41], Amazon Lambda [3], Google Application Engine [4], or Nirmata [5], is a distributed system manager that enables isolated microservice execution on shared datacenter infrastructure. To do so, microservice platforms include the following components (cf. [41]): 1. federation subsystem, abstracting and grouping servers into a unified cluster that holds deployed applications; 2. resource manager, allocating computation resources to individual microservices based on their execution requirements; 3. orchestrator, dynamically scheduling and migrating microservices within the cluster based on node health information, microservice execution statistics, and service-level agreements (SLAs); 4. transport subsystem, providing (secure) point-to-point communication among various microservices; 5. failover manager, guaranteeing high availability/reliability through replication; 6. troubleshooting utilities, which assist developers with performance profiling/debugging and understanding microservice co-execution interference.

A microservice platform usually provides a number of programming models [10] that developers adhere to, like dataflow and actor-based. The models capture the execution requirements and describe the communication relationship among microservices. For example, the data-flow model (e.g., Amazon Datapipe [6], Google Cloudflow [30], Azure Data Factory [56]) requires programmers to assemble microservices into a directed acyclic graph (DAG): nodes contain mi-
microservices that are interconnected via flow-controlled, lossless dataflow channels. These models bring attractive benefits for a heterogeneous platform since they explicitly express concurrency and communication, enabling the platform to transparently map it to the available hardware [69, 71]. Figure 1 shows an IoT thermostat analytics application [55] consisting of microservices arranged in 3 stages: 1. Thermostat sensor updates are authenticated by the API gateway; 2. Updates are logged into a SQL store sharded by a thermostat identifier; 3. SQL store updates trigger data analytic tasks (e.g., spike detection, moving average, and recommendation) based on thresholds. The dataflow programming model allows the SQL store sharding factor to be dynamically adjusted to scale the application with the number of thermostats reporting. Reliable collections ensure state consistency when re-sharding and the microservice platform automatically migrates and deploys DAG nodes to available hardware resources.

A microservice can be stateful or stateless. Stateless microservices have no persistent storage and only keep state within request context. They are easy to scale, migrate, and replicate, and they usually rely on other microservices for stateful tasks (e.g., a database engine). Stateful microservices use platform APIs to access durable state, allowing the platform full control over data placement. For example, Service Fabric provides reliable collections [41], a collection of data structures that automatically persist mutations. Durable storage is typically disaggregated for microservices and accessed over the network. The use of platform APIs to maintain state allows for fast service migration compared with traditional virtual machine migration [20], as the stateful working set is directly observed by the platform. All microservices in Figure 1 are stateful. We describe further microservices in §4.

2.2 SmartNICs

SmartNICs have appeared on the market [16, 52, 60] and in the datacenter [26]. SmartNICs include computing units, memory, traffic managers, DMA engines, TX/RX ports, and several hardware accelerators for packet processing, such as cryptography and pattern matching engines. Unlike traditional accelerators, SmartNICs integrate the accelerator with the NIC. This allows them to process network requests in-line, at much lower latency than other types of accelerators.

Two kinds of SmartNIC exist: (1) general-purpose, which allows transparent microservice offload and is the architecture we consider. For example, Mellanox BlueField [52] has 16 ARMv8 A72 cores with 2 × 100GE ports and Cavium LiquidIO [16] has 12 cnMIPS cores with 2 × 10GE ports. These SmartNICs are able to run full operating systems, but also ship with lightweight runtime systems that can provide kernel-bypass access to the NIC’s IO engines. (2) FPGA and ASIC based SmartNICs target highly specialized applications. Examples include match-and-action processing [26, 44] for network dataplanes, NPUs [27], and TPUs [40] for deep neural network inference acceleration. FPGAs and ASICs do not support transparent microservice offload. However, they can be combined with general-purpose SmartNICs.

A SmartNIC-accelerated server is a commodity server with one or more SmartNICs. Host and SmartNIC processors do not share thermal, memory, or cache coherence domains, and communicate via DMA engines over PCIe. This allows them to operate as independent, heterogeneous computers, while sharing a power domain and its idle power.

SmartNICs hold promise for improving server energy efficiency when compared to other heterogeneous computing approaches. For example, racks populated with low-power servers [8] or a heterogeneous mix of servers, suffer from high idle energy draw, as each server requires energy to power its chassis, including fans and devices, and its own ToR switch port. System-on-chip designs with asymmetric performance, such as ARM’s big.LITTLE [39] and DynamIQ [2] architectures, and AMD’s heterogeneous system architecture (HSA) [7], which combines a GPU with a CPU on the same die, have scalability limits due to the shared thermal design point (TDP). These architectures presently scale to a maximum of 8 cores, making them more applicable to mobile than to server applications. GPGPUs and single instruction multiple threads (SIMT) architectures, such as Intel’s Xeon Phi [37] and HP Moonshot [35], are optimized for computational throughput and the extra interconnect hop prevents these accelerators from running latency-sensitive microservices efficiently [58]. SmartNICs are not encumbered by these problems and can thus be used to balance the power draw of latency-sensitive services efficiently.

2.3 Benefits of SmartNIC Offload

We quantify the potential benefit of using SmartNICs for microservices on energy efficiency and request latency. To do so, we choose two identical commodity servers and equip one with a traditional 10GbE Intel X710 NIC and the other with a 10GbE Cavium LiquidIO SmartNIC. Then we evaluate 16 different microservices (detailed in §4) on these two servers with synthetic benchmarks of random 512B requests. We measure request throughput, wall power consumed at peak throughput (defined as the knee of the latency-throughput graph, where queueing delay is minimal) and when idle, as well as client-observed, average/tail request latency in a closed loop. We use host cores on the traditional server and SmartNIC cores on the SmartNIC server for microservice execution. We use as many identical microservice instances, CPUs, and client machines as necessary to attain peak throughput and put unused CPUs to their deepest sleep state. The SmartNIC does not support per-core low power states and always keeps all 12 cores active, diminishing SmartNIC energy efficiency results somewhat. The SmartNIC microservice runtime system uses a kernel-bypass network stack (cf. §4). To break out kernel overheads from the host experiments, we run all microservices on the
Table 1: Microservice comparison among host (Linux and DPDK) and SmartNIC. RPS = Throughput (requests/s), W = Active power (W), C = Number of active cores, L = Average latency (ms), 99% = 99th percentile latency, RPJ = Energy efficiency (requests/Joule).

<table>
<thead>
<tr>
<th>Microservice</th>
<th>Host (Linux)</th>
<th>Host (DPDK)</th>
<th>SmartNIC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RPS</td>
<td>W</td>
<td>C</td>
</tr>
<tr>
<td>IPsec</td>
<td>821.3K</td>
<td>117.0</td>
<td>12</td>
</tr>
<tr>
<td>BM25</td>
<td>91.9K</td>
<td>116.4</td>
<td>12</td>
</tr>
<tr>
<td>NIDS</td>
<td>1781.1K</td>
<td>111.0</td>
<td>12</td>
</tr>
<tr>
<td>Recommend</td>
<td>3.6K</td>
<td>109.4</td>
<td>12</td>
</tr>
<tr>
<td>NATv4</td>
<td>1889.6K</td>
<td>72.1</td>
<td>8</td>
</tr>
<tr>
<td>Count</td>
<td>1960.8K</td>
<td>68.1</td>
<td>6</td>
</tr>
<tr>
<td>EMA</td>
<td>1696.1K</td>
<td>72.7</td>
<td>8</td>
</tr>
<tr>
<td>KVS</td>
<td>1946.2K</td>
<td>48.6</td>
<td>8</td>
</tr>
<tr>
<td>Flow mon.</td>
<td>1944.1K</td>
<td>70.8</td>
<td>8</td>
</tr>
<tr>
<td>DDoS</td>
<td>1989.5K</td>
<td>111.2</td>
<td>12</td>
</tr>
<tr>
<td>KNN</td>
<td>42.2K</td>
<td>118.3</td>
<td>12</td>
</tr>
<tr>
<td>Spike</td>
<td>91.9K</td>
<td>112.5</td>
<td>12</td>
</tr>
<tr>
<td>Bayes</td>
<td>12.1K</td>
<td>113.9</td>
<td>12</td>
</tr>
<tr>
<td>API gw</td>
<td>1537.6K</td>
<td>108.5</td>
<td>12</td>
</tr>
<tr>
<td>Top ranker</td>
<td>711.9K</td>
<td>119.7</td>
<td>12</td>
</tr>
<tr>
<td>SQL</td>
<td>463.3K</td>
<td>114.7</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 2: Request size impact on SmartNIC RPJ benefits.

host in two configurations: 1. Linux kernel network stack; 2. kernel-bypass network stack [64], based on Intel’s DPDK [1]. Table 1 presents measured peak request throughput, active power (wall power at peak throughput minus idle wall power), number of active cores, (tail-)latency, and energy efficiency, averaged over 3 runs. Active power allows a direct comparison of host to SmartNIC processor power draw. Energy efficiency equals throughput divided by active power.

Kernel overhead. We first analyze the overhead of in-kernel networking on the host (Linux versus DPDK). As expected, the kernel-bypass networking stack performs better than the in-kernel one. On average, it improves energy efficiency by 21% (% column in Table 1) and reduces tail latency by 16%. Energy efficiency improves because (1) DPDK achieves similar throughput with fewer cores; (2) at peak server CPU utilization, DPDK delivers higher throughput.

SmartNIC performance. SmartNIC execution improves the energy efficiency of 12 of the measured microservices by a geometric mean of 6.5× compared with host execution using kernel bypass (× column in Table 1). The SmartNIC consumes at most 24.7W active power to execute these microservices while the host processor consumes up to 113W. IPsec, BM25, Recommend, and NIDS particularly benefit from various SmartNIC hardware accelerators (crypto coprocessor, fetch-and-add atomic units, floating point engines, and pattern matching units). NATv4, Count, EMA, KVS, Flow monitor, and DDoS can take advantage of the computational bandwidth and fast memory interconnect of the SmartNIC. In these cases, the energy efficiency comes not just from the lower power consumed by the SmartNIC, but also from peak throughput improvements versus the host processor. KNN and Spike attain lower throughput on the SmartNIC. However, since the SmartNIC consumes less power, the overall energy efficiency is still better than the host. For all of these microservices, the SmartNIC also improves client-observed latency. This is due to the hardware accelerated packet buffers and the elimination of PCIe bus traversals. SmartNICs can reduce average and tail latency by a geometric mean of 45.3% and 45.4% versus host execution, respectively.

The host outperforms the SmartNIC for Top ranker, Bayes classifier, SQL, and API gateway by a geometric mean of 4.1× in energy efficiency, 41.2% and 30.0% in average and tail latency reduction. These microservices are branch-heavy with large working sets that are not handled well by the simpler cache hierarchy of the SmartNIC. Moreover, the API gateway uses double floating point numbers for the rate limiter implementation, which the SmartNIC emulates in software.

Request size impact. SmartNIC performance depends also on request size. To demonstrate this, we vary the request size of our synthetic workload and evaluate SmartNIC energy efficiency benefits of 5 microservices versus host execution. Figure 2 shows that with small (<128B) requests, SmartNIC benefit of IPsec, NIDS, and DDoS is smaller. Small requests are more computation intensive and we are limited by the SmartNIC’s wimpy cores. SmartNIC offload hits a sweet-spot at 256–512B request size, where the benefit almost doubles. Here, network and compute bandwidth utilization are balanced for the SmartNIC. At larger request sizes, we are network bandwidth limited, allowing us to put host CPUs to sleep and SmartNIC benefits again diminish. This can be seen in particular for IPsec, which outperforms on the SmartNIC due to hardware cryptography acceleration, but still diminishes with larger request sizes. We conclude that request size has a
major impact on the benefit of SmartNIC offload. Measuring it is necessary to make good offload choices.

We conclude that SmartNIC offload can provide large energy efficiency and latency benefits for many microservices. However, it is not a panacea. Computation and memory intensive microservices are more suitable to run on the host processor. We need an efficient method to define and monitor critical SmartNIC offload criteria for microservices.

2.4 Challenges of SmartNIC Offload

While there are quantifiable benefits, offloading microservices to SmartNICs brings a number of additional challenges:

- SmartNICs share the same Ethernet MAC address with the host server. Layer 2 switching is not enough to route traffic between SmartNICs and host servers. We require a different switching scheme that can balance traffic and provide fault tolerance when a server equips multiple SmartNICs.
- Microservice platforms assume uniform communication performance among all computing nodes. However, Figure 3 shows that SmartNIC-Host (via PCIe) and SmartNIC-SmartNIC (via ToR switch) communication round-trip-time (RTT) is up to 83.3% and 86.2% lower than host-host (via ToR switch) kernel-bypass communication. We have to consider this topology effect to achieve good performance.
- Microservices share SmartNIC resources and contend with SmartNIC firmware for cache and memory bandwidth. This can create a head-of-line blocking problem for network packet exchange with both SmartNIC and host. Prolonged head-of-line blocking can result in denial of service to unrelated microservices and is more severe than transient sources of interference, such as network congestion. We need to sufficiently isolate SmartNIC-offloaded microservices from firmware to guarantee quality of service.

3 E3 Microservice Platform

We present the E3 microservice platform for SmartNIC-accelerated servers. Our goal is to maximize microservice energy efficiency at scale. Energy efficiency is the ratio of microservice throughput and cluster power draw. Power draw is determined by our choice of SmartNIC-acceleration, while E3 focuses on maximizing microservice throughput on this heterogeneous architecture. We describe how we support microservice offload to a SmartNIC and address the request routing, microservice placement, and scheduling challenges.

Figure 3: Average RTT (3 runs) of different communication mechanisms in a SmartNIC-accelerated server.

Figure 4: Hardware and software architecture of E3.

E3 is a distributed microservice execution platform. We follow the design philosophies of Azure Service Fabric [41] but add energy efficiency as a design requirement. Figure 4 shows the hardware and software architecture of E3. E3 runs in a typical datacenter, where servers are grouped into racks, with a ToR switch per rack. Each server is equipped with one or more SmartNICs, and each SmartNIC is connected to the ToR. This creates a new topology where host processors are reachable via any of the SmartNICs (Figure 4-a). SmartNICs within the same server also have multiple communication options—via the ToR or PCIe (§3.1).

Programming model. E3 uses a dataflow programming model. Programmers assemble microservices into a DAG of microservice nodes interconnected via channels in the direction of RPC flow (cf. Figure 1). A channel provides lossless data communication between nodes. A DAG in E3 describes all RPC and execution paths of a single microservice application, but multiple DAGs may coexist and execute concurrently. E3 is responsible for mapping DAGs to computational nodes.

Software stack. E3 employs a central, replicated cluster resource controller [41] and a microservice runtime on each host and SmartNIC. The resource controller includes four components: (1) traffic control, responsible for routing and load balancing requests between different microservices; (2) control-plane manager, placing microservice instances on cluster nodes; (3) data-plane orchestrator, dynamically migrates microservices across cluster nodes; (4) failover/replication manager, providing failover and node membership management using consistent hashing [76]. The microservice runtime includes an execution engine, an orchestrator agent, and a communication subsystem, described next.

Execution engine. E3 executes each microservice as a multi-threaded process, either on the SmartNIC or on the host. The host runs Linux. The SmartNIC runs a lightweight firmware. Microservices interact only via microservice APIs, allowing E3 to abstract from the OS. SmartNIC and host support hardware virtual memory for microservice confinement.
E3 is work-conserving and runs requests to completion. It leverages a round-robin policy for steering incoming requests to cores, context switching cores if needed.

**Orchestrator agent.** Each node runs an orchestrator agent to periodically monitor and report runtime execution characteristics to the resource controller. The information is used by (1) the failover manager to determine cluster health and (2) the data-plane orchestrator to monitor the execution performance of each microservice and make migration decisions. On the host, the agent runs as a separate process. On the SmartNIC, the agent runs on dedicated cores (blue in Figure 4-b) and a traffic manager hardware block exchanges packets between the NIC MAC ports and the agent. For each packet, the agent determines the destination (network, host, or SmartNIC core).

### 3.1 Communication Subsystem

E3 leverages various communication mechanisms, depending on where communicating microservices are located.

**Remote communication.** When communicating among host cores across different servers, E3 uses the Linux network stack. SmartNIC remote communication uses a user-level network stack [64].

**Local SmartNIC-host communication.** SmartNIC and host cores on the same server communicate via PCIe. Prior work has extensively explored communication channels via PCIe [48,49,61], and we adopt their design. High-throughput messaging for PCIe interconnects requires leveraging multiple DMA engines in parallel. E3 takes advantage of the eight DMA engines on the LiquidIO, which can concurrently issue scatter/gather requests.

**Local SmartNIC-SmartNIC communication.** SmartNICs in the same host can use three methods for communication. 1. Using the host to relay requests, involving two data transfers over PCIe and pointer manipulation on the host, increasing latency. 2. PCIe peer-to-peer [24], which is supported on most SmartNICs [16,52,60]. However, the bandwidth of peer-to-peer PCIe communication is capped in a NUMA system when the communication passes between sockets [58]. 3. ToR switch. We take the third approach and our experiments show that this approach incurs lower latency and achieves higher bandwidth than the first two.

### 3.2 Addressing and Routing

Since SmartNICs and their host servers share Ethernet MAC addresses, we have to use an addressing/routing scheme to distinguish between these entities and load balance across them. For illustration, assume we have a server with two SmartNICs; each NIC has one MAC port. If remote microservices communicate with this server, there will be two possible paths and each might be congested.

We use equal-cost multi-path (ECMP) [85] routing on the ToR switch to route and balance load among these ports. We assign each SmartNIC and the host its own IP. We then configure the ToR switch to route to SmartNICs directly via the attached ToR switch port and an ECMP route to the host IP via any of the ports. The E3 communication subsystem on each SmartNIC differentiates by destination IP address whether an incoming packet is for the SmartNIC or the host. On the host, we take advantage of NIC teaming [57] (also known as port trunking) to bond all related SmartNIC ports into a single logical interface, and then apply the dynamic link aggregation policy (supporting IEEE 802.3ad protocol). ECMP automatically balances connections to the host over all available ports. If a link or SmartNIC fails, ECMP will automatically rebalance new connections via the remaining links, improving host availability.

### 3.3 Control-plane Manager

The control-plane manager is responsible for energy-efficient microservice placement. This is a computing intensive operation due to the large search space with myriad constraints. Hence, it is done on the control plane. Service Fabric uses simulated annealing, a well-known approximate algorithm, to solve microservice placement. It considers three types of constraints: (1) currently available resources of each computing node (memory, disk, CPU, network bandwidth); (2) computing node runtime statistics (aggregate outstanding microservice requests); (3) individual microservice execution behavior (average request size, request execution time and frequency, diurnal variation, etc.). Service Fabric ignores network topology and favors spreading load over multiple nodes.

E3 extends this algorithm to support bump-in-the-wire SmartNICs, considering network topology. We categorize computing nodes (host or SmartNIC processors) into different levels of communication distance and perform a search from the closest to the furthest. We present the HCM algorithm (Algorithm 1). HCM takes as input the microservice DAG $G$ and source nodes $V_{src}$, as well as the cluster topology $T$, including runtime statistics for each computing node (as collected). HCM performs a breadth-first traversal of $G$ to map microservices to cluster computing nodes ($MS\_DAG\_TRAVERSE$).

If not already deployed ($get\_deployed\_node$), HCM (via $MS\_DAG\_TRAVERSE$) assigns a microservice $V$ to a computing node $N$ via the $find\_first\_fit$ function (lines 9-11) and deploys it via $set\_deployed\_node$. $find\_first\_fit$ is a greedy algorithm that returns the first computing node that satisfies the microservice constraints (via its resource and runtime statistics) without considering communication cost. If no such node is found, it returns a node closest to the constraints. Next, for the descendant microservices of a node $V$ (lines 12-15),
1. The same computing node as V with runtime execution. We believe that our proposed methods may be applied to detect load changes and migrating microservices in response to these changes among computational nodes at run-time. To do so, we piggyback several measurements onto the periodic node health reports made by orchestrator agents to the resource controller: This approach is lightweight and integrates well with runtime execution. We believe that our proposed methods can also be used in other microservice schedulers [34, 63, 67].

In this section, we introduce the additional techniques implemented in our data-plane orchestrator to mitigate issues of SmartNIC overload caused by compute-intensive microservices. These can interfere with the SmartNIC’s traffic manager, starving the host of network packets. They can also simply execute too slowly on the SmartNIC to be able to catch up with the incoming request rate.

### Host starvation

This issue is caused by head-of-line blocking of network traffic due to microservice interference with firmware on SmartNIC memory/cache. It is typically caused by a single compute-intensive microservice overloading the SmartNIC. To alleviate this problem, we monitor the incoming/outgoing network throughput and packet queue depth at the traffic manager. If network bandwidth is under-utilized, but there is a standing queue at the traffic manager, the SmartNIC is overloaded, and we need to migrate microservices.

### Microservice overload

This issue is caused by microservices in aggregate requiring more computational bandwidth than the SmartNIC can offer, typically because too many microservices are placed on the same SmartNIC. To detect this problem, we periodically monitor the execution time of each microservice and compare to its exponential moving average. When the difference is negative and larger than 20%, we assume a microservice overload and trigger microservice migration. The threshold was determined empirically.

### Microservice migration

For either issue, the orchestrator will migrate the microservice with the highest CPU utilization to the host. To do so, it uses a cold migration approach, similar to other microservice platforms. Specifically, when the orchestrator makes a migration decision, it will first push the microservice binary to the new destination, and then notify the runtime of the old node to (1) remove the microservice instance from the execution engine; (2) clean up and free any local resources; (3) migrate the working state, as represented by reliable collections [41], to the destination. After the orchestrator receives a confirmation from the original node, it will update connections and restart the microservice execution on the new node.

### 3.5 Failover/Replication Manager

Since SmartNICs share the same power supply as their host server, our failover manager treats all SmartNICs and the host to be in the same fault domain [41], avoiding replica placement within the same. Replication for fault tolerance is typically done across different racks of the same datacenter or across datacenters, and there is no impact from placing SmartNICs in the same failure domain as hosts.

### 4 Implementation

#### Host software stack

The E3 resource controller and host runtime are implemented in 1,287 and 3,617 lines of C (LOC),
When running on the SmartNIC, IPsec and API gateway can (1) use a hardware managed memory manager to store the state of vantage of a number of hardware accelerator libraries. We use support hashtables and arrays, preventing us from migrating are software emulated. E3 reliable collections currently only stateful and use reliable collections to maintain their state. Our compiler automatically uses the dedicated atomic fetch-and-add unit (65 LOC). For Count, EMA, KVS, and Flow monitor, NIDS can take advantage of the deterministic finite automata (Bayes), our compiler generates FPU code on the SmartNIC. For precision floating-point computations (EMA, KNN, Spike, and enterprise IT networks, by chaining NFs; (3) An IoT hub to build cloud-scale network middleboxes, software switches, and implements as a dataflow processing graph of workers that pass data tuples in real time to trigger computations; (2) Network function (NF) virtualization (NFV) [62], which is used to build cloud-scale network middleboxes, software switches, and enterprise IT networks, by chaining NFs; (3) An IoT hub (IOT) [54], which gathers sensor data from edge devices and generates events for further processing (e.g., spike detection, classifier) [13,78]. To maximize throughput, applications may shard and replicate microservices, resulting in a DAG node count larger than the involved microservice types. Table 3 presents the microservice types involved in each application, the deployed DAG node count, and references the workloads used for evaluation. The workloads are trace-based and synthetic benchmarks, validated against realistic scenarios. The average and maximum node fanouts among our applications are 6 and 12, respectively. Figure 1 shows IOT-TS as an example. IOT-TS is sharded into 6 microservices, and recommends each microservice has one backup replica.

Table 3: 8 microservice applications. N = # of DAG nodes.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>N Microservices</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFV-DN</td>
<td>Flow monitoring</td>
<td>72 Flow, IPsec, NIDS</td>
</tr>
<tr>
<td>NFV-IDN</td>
<td>Intrusion detection</td>
<td>60 DDoS, NATv4, NIDS</td>
</tr>
<tr>
<td>NFV-IFID</td>
<td>IPsec gateway</td>
<td>84 NATv4, Flow mon., IPsec, DDoS</td>
</tr>
<tr>
<td>RTA-PTC</td>
<td>Twitter analytics</td>
<td>60 Count, Top Ranner, KNN</td>
</tr>
<tr>
<td>RTA-SF</td>
<td>Spam filter</td>
<td>96 Spike, Count, KVS, Bayes</td>
</tr>
<tr>
<td>RTA-SHM</td>
<td>Server health mon.</td>
<td>84 Count, EMA, SQL, BM25</td>
</tr>
<tr>
<td>IOT-DH</td>
<td>IoT data hub</td>
<td>108 API gw, Count, KNN, KVS, SQL</td>
</tr>
<tr>
<td>IOT-TS</td>
<td>Thermostat</td>
<td>108 APLEMA, Spike, Recommend, SQL</td>
</tr>
</tbody>
</table>

Table 4: Evaluated systems and clusters. BC = Beefy cores, WC = Wimpy cores, Mem = Memory (GB), Idle and Peak power (W), Bw = Network bandwidth (Gb/s).

<table>
<thead>
<tr>
<th>System/Cluster</th>
<th>Cost [$]</th>
<th>BC</th>
<th>WC</th>
<th>Mem</th>
<th>Idle</th>
<th>Peak</th>
<th>Bw</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beefy</td>
<td>4,500</td>
<td>12</td>
<td>0</td>
<td>64</td>
<td>83</td>
<td>201</td>
<td>20</td>
</tr>
<tr>
<td>Wimpy</td>
<td>2,209</td>
<td>32</td>
<td>2</td>
<td>79</td>
<td>95</td>
<td>60</td>
<td>20</td>
</tr>
<tr>
<td>Type1-SmartNIC</td>
<td>4,650</td>
<td>12</td>
<td>12</td>
<td>68</td>
<td>98</td>
<td>222</td>
<td>20</td>
</tr>
<tr>
<td>Type2-SmartNIC</td>
<td>6,750</td>
<td>16</td>
<td>48</td>
<td>144</td>
<td>145</td>
<td>252</td>
<td>40</td>
</tr>
<tr>
<td>SuperBeefy</td>
<td>12,550</td>
<td>24</td>
<td>0</td>
<td>192</td>
<td>77</td>
<td>256</td>
<td>80</td>
</tr>
<tr>
<td>4×Beefy</td>
<td>18,000</td>
<td>48</td>
<td>0</td>
<td>256</td>
<td>332</td>
<td>804</td>
<td>80</td>
</tr>
<tr>
<td>4×Wimpy</td>
<td>8,836</td>
<td>128</td>
<td>8</td>
<td>316</td>
<td>380</td>
<td>880</td>
<td>80</td>
</tr>
<tr>
<td>2×B.+2×W.</td>
<td>13,018</td>
<td>24</td>
<td>64</td>
<td>132</td>
<td>324</td>
<td>592</td>
<td>80</td>
</tr>
<tr>
<td>2×Type2-SmartNIC</td>
<td>13,500</td>
<td>32</td>
<td>96</td>
<td>288</td>
<td>290</td>
<td>504</td>
<td>80</td>
</tr>
<tr>
<td>1×SuperBeefy</td>
<td>12,550</td>
<td>24</td>
<td>0</td>
<td>192</td>
<td>77</td>
<td>256</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 5: Evaluated microservices.

<table>
<thead>
<tr>
<th>Microservice</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPsec</td>
<td>Authenticates (SHA-1) &amp; encrypts (AES-CBC-128) NATv4 [43]</td>
</tr>
<tr>
<td>BM25</td>
<td>Search engine ranking function [86], e.g., ElasticSearch</td>
</tr>
<tr>
<td>NATv4</td>
<td>IPv4 network address translation using DIR-24-8-BASIC [33]</td>
</tr>
<tr>
<td>NIDS</td>
<td>Network intrusion detection w/aho-corasick parallel match [82]</td>
</tr>
<tr>
<td>Count</td>
<td>✓ Item frequency counting based on a bitmap [43]</td>
</tr>
<tr>
<td>EMA</td>
<td>✓ Exponential moving average (EMA) for data streams [84]</td>
</tr>
<tr>
<td>KVS</td>
<td>✓ Hashable-based in-memory key-value store [25]</td>
</tr>
<tr>
<td>Flow mon.</td>
<td>✓ Flow monitoring system using count-min sketch [43]</td>
</tr>
<tr>
<td>DDoS</td>
<td>✓ Entropy-based DDoS detection [50]</td>
</tr>
<tr>
<td>Recommend</td>
<td>✓ Recommendation system using collaborative filtering [83]</td>
</tr>
<tr>
<td>KNN</td>
<td>Classifier using the K-nearest neighbours algorithm [88]</td>
</tr>
<tr>
<td>Spike</td>
<td>✓ Spike detector from a data stream using Z-score [73]</td>
</tr>
<tr>
<td>Bayes</td>
<td>✓ Naive Bayes classifier based on maximum a posteriori [50]</td>
</tr>
<tr>
<td>API gw</td>
<td>✓ API rate limiter and authentication gateway [9]</td>
</tr>
<tr>
<td>Top Ranner</td>
<td>✓ Top-K ranner using quicksort [79]</td>
</tr>
<tr>
<td>SQL</td>
<td>✓ In-memory SQL database [53]</td>
</tr>
</tbody>
</table>

Microservices. We implemented 16 popular microservices on E3, as shown in Table 2, in an aggregate 6,966 LOC. Six of the services are stateless or use read-only state that is modified only via the cluster control plane. The remaining services are stateful and use reliable collections to maintain their state. When running on the SmartNIC, IPsec and API gateway can use the crypto coprocessor (105 LOC), while Recommend and NIDS can take advantage of the determinisitic finite automata unit (65 LOC). For Count, EMA, KVS, and Flow monitor, our compiler automatically uses the dedicated atomic fetch-and-add units on the SmartNIC. When performing single-precision floating-point computations (EMA, KNN, Spike, and enterprise IT networks, by chaining NFs; (3) An IoT hub (IOT) [54], which gathers sensor data from edge devices and generates events for further processing (e.g., spike detection, classifier) [13,78]. To maximize throughput, applications may shard and replicate microservices, resulting in a DAG node count larger than the involved microservice types. Table 3 presents the microservice types involved in each application, the deployed DAG node count, and references the workloads used for evaluation. The workloads are trace-based and synthetic benchmarks, validated against realistic scenarios. The average and maximum node fanouts among our applications are 6 and 12, respectively. Figure 1 shows IOT-TS as an example. IOT-TS is sharded into 6 microservices, and recommends each microservice has one backup replica.

5 Evaluation

Our evaluation aims to answer the following questions:

1. What is the energy efficiency benefit of microservice SmartNIC-offload? Is it proportional to client load? What is the latency cost? (§5.1)
2. Does E3 overcome the challenges of SmartNIC-offload? (§5.2, §5.3, §5.4)
3. Do SmartNIC-accelerated servers provide better total cost of ownership than other cluster architectures? (§5.5)

Experimental setup. Our experiments run on a set of clusters (Table 4 presents the server and cluster configurations), attached to an Arista DCS-7050S ToR switch. Beefy is a Supermicro 1U server, with a 12-core E5-2680 v3 processor.
at 2.5GHz, and a dual-port 10Gbps Intel X710 NIC. Wimpy is ThunderX-like, with a CN6880 processor (32 cnMIPS64 cores running at 1.2GHz), and a dual-port 10Gbps NIC. SuperBeefy is a Supermicro 2U machine, with a 24-core Xeon Platinum 8160 CPU at 2.1GHz, and a dual-port 40Gbps Intel XL710 NIC. Our SmartNIC is the Cavium LiquidIOII [16], with one OCTEON processor with 12 cnMIPS64 cores at 1.2GHz, 4GB memory, and two 10Gbps ports. Based on this, we build two SmartNIC servers: Type1 is Beefy, but swaps with one OCTEON processor with 12 cnMIPS64 cores at XL710 NIC. Our SmartNIC is the Cavium LiquidIOII [16], of which we provide as many as necessary. To do so, we deploy as many instances of each application and peak utilization. We evaluate the latency and energy efficiency of using SmartNICs for microservice applications, compared to homogeneous clusters. We compare 3×Beefy to 3×Type1-SmartNIC, to ensure that microservices also communicate remotely. We focus first on peak utilization, which is desirable for energy efficiency, as it amortizes idle power draw. To do so, we deploy as many instances of each application and apply as much client load as necessary to maximize request throughput without overloading the cluster, as determined by the knee of the latency-throughput curve. Figure 5 shows that Type1-SmartNIC achieves an average 2.5×, 1.3×, and 1.3× better energy efficiency across the NFV, RTA, and IOT application classes, respectively. This goes along with 43.3%, 92.3%, and 80.4% average latency savings and 35.5%, 90.4%, 88.6%, 99th percentile latency savings, respectively. NFV-FIN gains the most—3× better energy efficiency—because E3 is able to run all microservices on the SmartNICs. RTA-PTC benefits the least—12% energy efficiency improvement at 4% average and tail latency cost—as E3 only places the Count microservice on the SmartNIC and migrates the rest to the host.

Power proportionality. This experiment evaluates the power proportionality of E3 (energy efficiency at lower than peak utilization). Using 3×Type1-SmartNIC, we choose an application from each class (NFV-FIN, RTA-SHM, and IOT-TS) and vary the offered request load between idle and peak via a client side request limiter. Figure 8 shows that RTA-SHM and IOT-TS are power proportional. NFV-FIN is not power proportional but also draws negligible power. NFV-FIN runs all microservices on the SmartNICs, which have low active power, but the cnMIPS architecture has no per-core sleep states.

We conclude that applications can benefit from E3’s microservice offload to SmartNICs, in particular at peak cluster utilization. Peak cluster utilization is desirable for energy efficiency and microservices make it more common due to light-weight migration. However, transient periods of low load can occur and E3 draws power proportional to request load. We can apply insights from Prekas, et al. [66] to reduce polling overheads and improve power proportionality further.

5.2 Avoiding Host Starvation

We show that E3’s data-plane orchestrator prevents host starvation by identifying head-of-line blocking of network traffic. To do so, we use 3×Type1-SmartNIC and place as many microservices on the SmartNIC as fit in memory. E3 identifies the microservices that cause interference (Top Ranker in RTA-PTC, Spike in RTA-SF, API gateway in IOT-DH and IOT-TS) and migrates them to the host. As shown in Figure 7, our approach achieves up to 29× better energy efficiency and up to 89% latency reduction across RTA-PTC, RTS-SF, IOT-DH, and IOT-TS. For the other applications, our traffic engine has little effect because the initial microservice assignment already put the memory intensive microservices on the host.

5.3 Sharing SmartNIC and Host Bandwidth

This experiment evaluates the benefits of sharing SmartNIC network bandwidth with the host. We compare two Type2-SmartNIC configurations: 1. Sharing aggregate network bandwidth among host and SmartNICs, using ECMP to balance host traffic over SmartNIC ports; 2. Replacing one SmartNIC with an Intel X710 NIC used exclusively to route traffic to the host. To emphasize the load balancing benefits, we always place the client-facing microservices on the host server. Note that SmartNIC-offloaded microservices still exchange network traffic (when communicating remotely or among SmartNICs) and interfere with host traffic.

Figure 9 shows that load balancing improves application throughput up to 2.9× and cluster energy efficiency up to 2.7× (NFV-FIN). Available host network bandwidth when sharing SmartNICs can be up to 4× that of the dedicated NIC, which balances better with the host compute bandwidth. With a dedicated NIC, host processors can starve for network bandwidth. IOT-TS is compute-bound and thus benefits the
While SmartNICs benefit energy efficiency and thus potentially bring cost savings, can they compete with other forms of heterogeneous clusters, especially when factoring in the capital expense to acquire the hardware? In this experiment, we evaluate the cost efficiency, in terms of request throughput over total cost and time of ownership, of using SmartNICs for microservices, compared with four other clusters (see Table 4). Assuming that clusters are usually at peak utilization, we use the cost efficiency metric $\frac{\text{Throughput} \times T}{\text{CAPEX} + \left(\left(\text{Power} \times T\right) \times \text{Electricity}\right)}$, where Throughput is the measured average throughput at peak utilization for each application, as executed by E3 on each cluster, $T$ is elapsed time, CAPEX is the capital expense to purchase the cluster including all hardware components ($\), Power is the elapsed peak power draw of the cluster (Watts), and Electricity is the price of electricity ($/Watts). The cluster cost and power data is shown in Table 4 and we use the average U.S. electricity price [32] of $0.0733/kWh. Figure 10 reports results for three applications of very different points in the workload space, extrapolated over time of ownership by our cost efficiency metric.

We make three observations. First, in the long term (>1 year of ownership), cost efficiency is increasingly dominated by energy efficiency. This highlights the importance of energy efficiency for data center design, where servers are typically replaced after several years to balance CAPEX [12]. Second, when all microservices are able to run on a low power platform (NFV-FIN), both $4 \times$ Wimpy and $2 \times$ Type2-SmartNIC clusters are the most cost efficient. After 5 years, $4 \times$ Wimpy is 14.1% more cost efficient than $2 \times$ Type2-SmartNIC because of the lower power draw. Third, when a microservice application contains both compute and IO-intensive microservices (RTA-SHM, IOT-TS), the $2 \times$ Type2-SmartNIC cluster is up to 1.9 times more cost efficient after 5 years of ownership than the next best cluster configuration ($4 \times$ Beefy in both cases).

Table 5 presents the measured energy-efficiency, which shows cost efficiency in the limit (over very long time of ownership). We can see that $4 \times$ Wimpy is only 3% more energy efficient (but has lower CAPEX) than $2 \times$ Type2-SmartNIC for NFV-FIN. $2 \times$ Type2-SmartNIC is on average 2.37 more energy-efficient (but has higher CAPEX) than $1 \times$ SuperBeefy, which is the second-best cluster in terms of energy-efficiency.

### 5.5 Energy Efficiency = Cost Efficiency

While SmartNICs benefit energy efficiency and thus potentially bring cost savings, can they compete with other forms of heterogeneous clusters, especially when factoring in the capital expense to acquire the hardware? In this experiment, we evaluate the cost efficiency, in terms of request throughput over total cost and time of ownership, of using SmartNICs for microservices, compared with four other clusters (see Table 4). Assuming that clusters are usually at peak utilization,
FatTree [31] topology with 40 servers per rack. To avoid host starvation and microservice overload, E3’s data-plane orchestrator receives one heartbeat message (16B) every 50ms from each SmartNIC that reports the queue length of the traffic manager and the SmartNIC’s microservice execution times. The orchestrator parses the heartbeat message and makes a migration decision (§3.4). Figure 11 shows that the time taken to transmit the message and make a decision with a large number of servers stays well below the time taken to migrate the service (on the order of 10s-100s of ms) and is negligibly impacted by the number of deployed microservices. This is because the heartbeat message contributes only 1Kbps of traffic, even with 50K servers.

E3 uses HCM in the control-plane manager. We compare it to simulated annealing and ILP, deploying 10K microservices on an increasing number of servers. Table 6 shows that while HCM does not scale as well as simulated annealing, it can deploy new microservices in a reasonable time span (<<1s) at scale. ILP fails to deliver acceptable performance.

Tail latencies. At scale, tail latencies dominate [21]. While SmartNICs can introduce high tail latency for some microservices (§2.3), E3 places these microservices on the host to ensure that application-level tail-latency inflation is minimized (§5.1). The tail-latency impact of SmartNIC offload is reduced at scale, as baseline communication latency increases with increasing inter-rack distance.

Table 6: Per-microservice deployment time (ms) scalability.

<table>
<thead>
<tr>
<th>Servers</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>600</th>
<th>800</th>
<th>1,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCM</td>
<td>4.85</td>
<td>8.31</td>
<td>19.83</td>
<td>34.32</td>
<td>74.39</td>
<td>263.46</td>
</tr>
<tr>
<td>Annealing</td>
<td>3.15</td>
<td>4.73</td>
<td>7.43</td>
<td>15.64</td>
<td>23.50</td>
<td>61.42</td>
</tr>
<tr>
<td>ILP</td>
<td>7.64</td>
<td>19.43</td>
<td>84.83</td>
<td>361.85</td>
<td>2.57x</td>
<td>1s</td>
</tr>
</tbody>
</table>

Figure 6: Communication-aware microservice placement.

Figure 7: Avoiding host starvation (HS).

Figure 8: Power draw of 3 applications normalized to idle power of 3×Type1-SmartNIC, varying request load.

Figure 9: ECMP-based SmartNIC sharing (log y scale).

Figure 10: Cost efficiency of 3 applications across the cluster configurations from Table 4.

Figure 11: Orchestrator migration decision time scalability.
Energy-efficiency and power budgets. E3’s energy efficiency benefits are constant regardless of deployment size and power budgets. At scale, there is additional energy cost for core and spine switches, but these are negligible compared to racks (ToRs and servers). Within a rack, ToR switch energy consumption stays constant, as all compared systems use the same number of switch ports. Our results show that E3 achieves up to 1.9x more throughput under the same power budget. Conversely, operators can save 48% of power, offering the same bandwidth.

6 Related Work

Architecture studies for microservices. Prior work has explored the architectural implications of running microservices [22, 28, 80]. It shows that wimpy servers hold potential for microservices under low loads, as they have less cache utilization and network virtualization overhead compared with traditional monolithic cloud workloads. FAWN [8] explored using only low-power processor architecture for data-intensive workloads as an energy and cost-effective alternative to server multiprocessors. However, FAWN assumed that I/O speeds are much lower than CPU speeds and so CPUs would be left idle for data-intensive applications. With the advent of fast network technologies, server CPUs are still required. Motivated by these studies, we focus on using SmartNIC-accelerated servers for energy efficiency.

Heterogeneous scheduling. A set of schedulers for performance-asymmetric architectures have been proposed. For example, Kumar et al. [45, 46] use instructions per cycle to determine relative speedup of each thread on different types of cores. HASS [74] introduces the architectural signature concept as a scheduling indicator, which contains information about memory-boundness, available instruction-level parallelism, etc. CAMP [72] combines both efficiency and thread-level parallelism specialization and proposes a lightweight technique to discover which threads could use fast cores more efficiently. PTask [70] provides a data-flow programming model for programmers to manage computation for GPUs. It enables sharing GPUs among multiple processes, parallelizes multiple tasks, and eliminates unnecessary data movements. These approaches target long-running computations, mostly on cache coherent architectures, rather than microsecond-scale, request-based workloads over compute nodes that do not share memory and are hence not applicable.

Microservice scheduling. Wisp [77] enforces end-to-end performance objectives by globally adapting rate limiters and request schedulers based on operator policies under varying system conditions. This work is not concerned with SmartNIC heterogeneity. UNO [47] is an NFV framework that can systematically place NFs across SmartNIC and host with a resource-aware algorithm on the control plane. E3 is a microservice platform and thus goes several steps further: (1) E3 uses a data-plane orchestrator to detect node load and migrates microservices if necessary; (2) HCM considers communication distance during the placement. With the advent of SmartNICs and programmable switches, researchers have identified the potential performance benefits of applying request processing across the communication path [15]. E3 is such a system designed for the programmable cloud and explores the energy efficiency benefits of running microservices.

Power proportionality. Power proportional systems can vary energy use with the presented workload [51]. For example, Prekas, et al. propose an energy-proportional system management policy for memcached [66]. While E3 can provide energy proportionality, we are primarily interested in energy-efficiency. Geoffrey et al. [19] propose a heterogeneous power-proportional system. By carefully selecting component ensembles, it can provide an energy-efficient solution for a particular task. However, due to the high cost of ensemble transitions, we believe that this architecture is not fit for high bandwidth I/O systems. Rivoire, et al. propose a more balanced system design (for example, a low-power, mobile processor with numerous laptop disks connected via PCIe) and show that it achieves better energy efficiency for sorting large data volumes [68]. Pelican [11] presents a software storage stack on under-provisioned hardware targeted at cold storage workloads. Our proposal could be viewed as a balanced-energy approach for low-latency query-intensive server applications, rather than cold, throughput intensive ones.

7 Conclusion

We present E3, a microservice execution platform on SmartNIC-accelerated servers. E3 extends key system components (programming model, execution engine, communication subsystem, scheduling) of the Azure Service Fabric microservice platform to a SmartNIC. E3 demonstrates that SmartNIC offload can improve cluster energy-efficiency up to 3× and cost efficiency up to 1.9× at up to 4% latency cost for common microservices.

Acknowledgments

This work is supported in part by NSF grant 1751231 and the Texas Systems Research Consortium. We would like to thank the anonymous reviewers and our shepherd, Ada Gavrilovska, for their comments and feedback.

References


[45] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA heterogeneous multi-core architectures: The


