The Case For In-Network Computing On Demand

Yuta Tokusashi
Keio University

Huynh Tu Dang
Università della Svizzera italiana

Fernando Pedone
Università della Svizzera italiana

Robert Soulé
Università della Svizzera italiana
Barefoot Networks

Noa Zilberman
University of Cambridge

Abstract

Programmable network hardware can run services traditionally deployed on servers, resulting in orders-of-magnitude improvements in performance. Yet, despite these performance improvements, network operators remain skeptical of in-network computing. The conventional wisdom is that the operational costs from increased power consumption outweigh any performance benefits. Unless in-network computing can justify its costs, it will be disregarded as yet another academic exercise.

In this paper, we challenge that assumption, by providing a detailed power analysis of several in-network computing use cases. Our experiments show that in-network computing can be extremely power-efficient. In fact, for a single watt, a software system on commodity CPU can be improved by a factor of ×100 using an FPGA, and a factor of ×1000 utilizing ASIC implementations. However, this efficiency depends on the system load. To address changing workloads, we propose *in-network computing on demand*, where services can be dynamically moved between servers and the network. By shifting the placement of services on-demand, data centers can optimize for both performance and power efficiency.

CCS Concepts • **Networks In-network processing**; • **Hardware Power estimation and optimization**.

ACM Reference Format:
Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing On Demand. In *Proceedings of Fourteenth EuroSys Conference 2019 (EuroSys '19).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3302424.3303979

EuroSys '19, March 25–28, 2019, Dresden, Germany
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6281-8/19/03...$15.00
https://doi.org/10.1145/3302424.3303979

1 Introduction

Data center operators face a challenging task. On the one hand, they must satisfy the ever-increasing demand for greater data volumes and better performance. On the other hand, they must decrease operational costs and their environmental footprint by reducing power consumption.

One promising approach to increasing application performance is *in-network computing* [22, 33, 36, 38, 69]. In-network computing refers to a particular type of hardware acceleration where network traffic is intercepted by the accelerating network device before it reaches the host, and where computations traditionally performed in software are executed by a network device, such as a networked FPGA [27], smart network interface card (smartNIC), or programmable ASIC [11].

Researchers have used in-network computing to achieve eye-popping performance results. For example, Jin et al. [38] demonstrated that a key-value cache implemented in a programmable ASIC can process more than 2B queries/second, and Chung et al. [18] demonstrated support of neural networks at tens of tera-operations per second. And, Jepsen et al. [36] describe a stream processing benchmark that achieves 4B events/second.

But, while such orders of magnitude performance improvements certainly sound attractive, to date, there has been very little attention paid to the other side of the ledger. Power consumption is a tremendous concern for cloud service providers [29] and data center operators have expressed qualms over the impact of hardware acceleration [56]. The conventional wisdom is that FPGAs and programmable network devices are power hungry, and so it is natural to ask if the benefits are worth the cost. In this paper, we explore the question: *Can in-network computing justify its power consumption?*

Answering this question is not easy, as there are many challenges for characterizing the power-vs-performance trade-offs for in-network computing. First, there is a wide variety of potential hardware targets (e.g., FPGAs, ASICs, etc.) and many different vendors. It is widely known that platforms from different vendors have different power properties. Second, there is a diversity of applications [18, 36, 38], and each of these applications use in-network computing in different ways. Third, implementations of similar applications often make different design choices, such as using on-chip or off-chip memory. Fourth, different applications are written using different tools and frameworks (e.g., hand-written Verilog vs. high-level synthesis), which can impact their resource usage, performance, and power consumption.

To mitigate these challenges, we used the following methodology. (i) We selected three diverse applications, allowing us to sample from distinct use cases within the data center: a key value store, a consensus protocol, and a domain name system. (ii) Each of the applications was developed using a different language and tool chain: Verilog, Kiwi/Emu [73], and P4 [14]. (iii) We built upon the modularity of one of the designs (KVS), to benchmark the power contribution of different components. And (iv), we used a common acceleration platform (NetFPGA-SUME [86]), and a single server environment, allowing for an apples-to-apples comparison. But, in order to generalize our findings, we also studied the behavior of one application, consensus, on a switch ASIC, and extended the discussion to SmartNICs and systems on chip (SoC)

Our study reveals some subtle and surprising results:

- The increase in power consumption for in-network computing over standard network devices is small.
- Software-based solutions are more power efficient at idle. But, with a very low processing load, sometimes 10% of the CPU's processing capability, in-network computing becomes more power efficient.
- Increasing the processing load has little effect on the power consumption of in-network computing.

Given these observations, we present several "rules-of-thumb" for when and how to leverage in-network computing. Through a modular design of the in-network computing applications, we demonstrate the importance (or lack) of certain design choices.

We argue that programmable network hardware can and should be treated the same as other scheduled computing resources. Services can be assigned to network devices when the workload and operational conditions are favorable. In support of this thesis, we propose *in-network computing on demand* in data centers. This new approach to in-network computing enables seamlessly shifting workloads from servers to the network, in a manner that optimizes power efficiency and saves up to 50% of the power compared with software-based solutions.

In short, this paper makes the following contributions:

- It provides a detailed study of the power consumption of in-network computing applications, showing only modest additional power consumption overhead.
- It compares software-based and network-based applications, showing that in-network computing has better power efficiency and performance starting at low loads.
- It introduces *in-network computing on demand*, optimizing power efficiency by shifting applications between the software and the network.

2 Scope

Sections 4-7 describe a set of experiments that evaluate the trade-off between performance and power-consumption for in-network computing applications, as well as observations from different hardware targets. Before delving into the details, we first define the scope of this work.

Choice of Applications. We study three applications: a key value store, a consensus protocol, and a domain name system (DNS) server. We chose these particular applications for several important reasons: (i) they represent three distinct use cases within a data center, (ii) they are implemented using very different architectures, (iii) different design flows were used in their development, (iv) they are available under an open-source license, allowing to reproduce this work, and (v) they can all be run on a common hardware platform (NetFPGA SUME).

However, there has been significant work on accelerating applications in the network, and there are many different possible design choices. We did not necessarily choose applications that yielded the best performance characteristics. Indeed, other applications have achieved better performance through specialization (e.g., [26, 64]), running on different hardware targets (e.g., [38]), or through design choices such as protocol or memory type (e.g., [33, 46]). On a similar note, we did not choose the applications based on particular feature sets (e.g., Caribou [32] provides a wide range of functionality that would be impossible to provide with an ASIC). It was more important to our study to explore different architectures and workflows on a common hardware target.

In-Network Computing vs. Hardware Acceleration. This study focuses specifically on in-network computing, and not on the more general topic of hardware acceleration. By in-network computing, we mean that we study designs that serve as both network devices and accelerators. For example, we do not study GPUs, as they are terminating devices. Prior work has focused on hardware acceleration [60] and alternative deployments. For example, Catapult [67] places an FPGA in front of a NIC to accelerate applications such as neural networks [18]. These deployments are out of scope for this paper.

Performance Metrics. We study power consumption for both the low-end and high-end of utilization, not just at peak performance. We chose throughput as the main performance metric, as most in-network computing deployments will have lower latency simply by virtue of their deployment. We briefly discuss latency in Section 9.5.

Deployment. For our study, we assume that a single in-network computing application is deployed on a network device. Recent work has proposed virtualization techniques for deploying multiple data-plane programs concurrently [85]. It would be interesting in future work to study the impact of such a deployment.

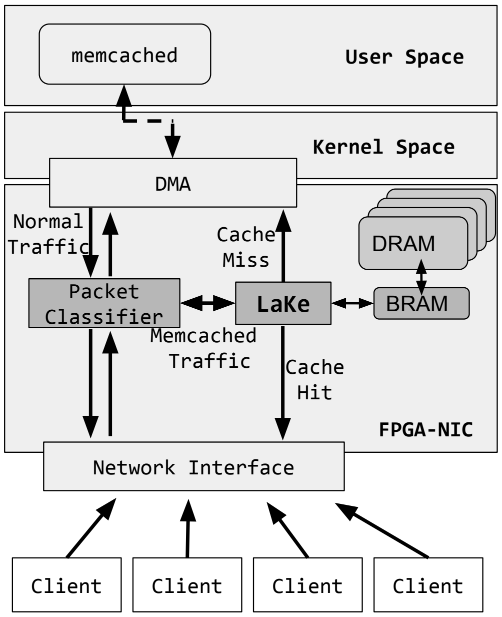


Figure 1. High level architecture of LaKe.

3 Case Studies

Below, we provide an overview of the three applications used in our case study: a key value store (KVS), a consensus protocol, and a domain name system (DNS) server. These applications are all good candidates for network acceleration, as opposed to hardware acceleration, because they are I/O bound rather than CPU bound on the host. KVS and DNS were also shown to be latency-sensitive on the microsecond level [65].

We stress that the architecture of these applications is not a contribution of this paper. We describe their designs here in order to provide the necessary background for the later sections. For more details, we refer the readers to the original papers [20, 21, 73, 77, 78]. All architectures either support, or are modified to support, both application-specific and standard network functionality.

3.1 LaKe: Key-Value Store

LaKe [77, 78], a Layered Key-value store, can be considered a hardware-based implementation of memcached [61]. It accelerates storage accesses by providing two layers of caching: an on-chip memory (BRAM) on an FPGA and DRAM memory located on the FPGA card, as shown in Figure 1. A query is only forwarded to software if there are misses at both layers.

LaKe was implemented in Verilog. This allows for fine-grain control of low-level resources and avoids potential overheads due to compiling from high-level languages.

LaKe uses multiple processing elements (PE) to conceal latency contributed by accesses to external memory. The number of PEs is scalable and configurable. Each PE takes less than 3% of the FPGA resources. 5 PEs are sufficient to achieve 10GE line rate (roughly 13M queries/sec). LaKe supports standard memcached functionality, unlike other solutions [46], and provides $\times 10$ latency and throughput improvement and $\times 24$ power efficiency improvement compared to software-based memcached.

LaKe has several important traits that make it ideal for this study. First, LaKe runs on a platform that also acts, at the same time, as a NIC or a switch, allowing us to enable or disable its KVS functionality. Second, it has a modular and scalable design. By controlling the number of PEs, power efficiency can be balanced against throughput. Third, LaKe enables studying power efficiency trade-offs in the use of different types of memories.

3.2 P4xos: Consensus

Several recent projects have used programmable network hardware to accelerate consensus protocols, including NetPaxos [22], Speculative Paxos [66], NoPaxos [48], Consensus in a Box [33], and NetChain [37]. In this paper, we focus on the P4 implementation of Lamport's Paxos algorithm [42] described in Paxos Made Switch-y [21]. We refer to the implementation as P4xos.

The Paxos protocol distinguishes the following roles that a process can play: *proposers*, *acceptors*, *learners*, and *leaders* [42]. P4xos provides P4 implementations of the leader and acceptors. It optimizes the protocol because it: (i) reduces end-to-end latency by executing consensus logic as messages pass through the network, and (ii) avoids network I/O bottlenecks in software implementations by executing Paxos logic in the hardware.

One aspect of P4xos relevant to this study is that the components are interchangeable with multiple software implementations, including the open-source libpaxos library [49], and a variation of libpaxos ported to use the kernel-bypass DPDK [25]. Moreover, because P4xos is written in P4, one can use P4-to-FPGA compilers [79, 84] and P4-to-ASIC compilers [16] to target both hardware devices. Thus, overall, we can make direct comparisons between four different variations: traditional software library, software library using DPDK, FPGA-based, and ASIC-based.

We evaluated P4xos on several hardware targets, including a CPU, an FPGA, and a programmable ASIC. The libpaxos software implementation of an acceptor could achieve a throughput of 178K messages/second. A deployment on NetFPGA SUME could achieve 10M messages/second. And, the ASIC-based deployment could process over 2.5 billion consensus messages per second. Latency in the FPGA was less than on the CPU. Latency on the ASIC was less than the FPGA.

3.3 EMU DNS: Network Function

Several projects have explored data-plane acceleration for DNS servers, leveraging FPGAs [73] or kernel-bypass [51, 70]. In this paper, we focus on Emu DNS [73].

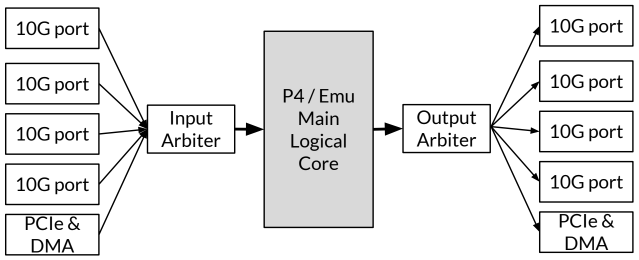


Figure 2. High level architecture of P4xos and Emu DNS, as implemented on NetFPGA. The main logical core (shaded grey) is the output of a program compiled from P4/C#.

Emu DNS implements a subset of DNS functionality, supporting non-recursive queries. The design supports resolution queries from names to IPv4 addresses. If the queried name is absent from the resolution table, Emu DNS informs the client that it cannot resolve the name.

Emu DNS was developed using Emu [73], a framework for developing network functions on FPGAs using C#. Emu builds on the Kiwi compiler [72], which allows developers to program FPGAs with .NET code. Emu provides Kiwi with a library for network functionality.

Both P4xos and Emu DNS share a similar high-level device architecture, as shown in Figure 2. In both cases, interfaces, queueing, and arbitration are done in shell modules provided by NetFPGA. Both the P4xos and Emu DNS programs are compiled to a main logical block that uses only on-chip memory. The micro-architecture of each project's logical block is, obviously, different.

Prior work [73] performed a benchmark comparison between Emu DNS and NSD [62], an open source, authoritative only, name server running on a host. The experiments showed that Emu DNS provides $\times 5$ throughput improvement and approximately $\times 70$ average and 99th percentile latency improvement.

The original Emu DNS acts only as a DNS, and not as a NIC or a switch. To support dynamic shifting between hardware and software, we amended the original design with a packet classifier, similar to the one used in LaKe, allowing Emu DNS to serve both as a NIC (for non-DNS traffic) and as a DNS server.

3.4 Applications: Similarities and Differences

All three applications share a common property: they were implemented on the NetFPGA SUME platform [86]. This property is essential for our study, as it allows us to benchmark the application performance and power consumption given the same underlying hardware capabilities. One of the known problems in power benchmarking is that platforms from different vendors have different power characteristics; this is not the case in our study.

Beyond sharing the same platform, all three implementations are UDP based, a common case for DNS and Paxos. While offloading TCP to hardware is possible [59, 71], existing solutions did not match the needs set in §2. All three chosen applications use the same 10GE interfaces presented on the NetFPGA SUME front panel.

The three applications differ in several important aspects: their role, their development flow, and the way they are used. In term of usage, Emu DNS represents a common network function provided in data centers. P4xos is used to achieve consensus in distributed systems. LaKe represents a common data center application. As we will discuss in § 9, the usage of the applications reflects on the ability to dynamically shift them in a working data center, and on the limitations to doing so.

The applications also differ in the way they are implemented, using different pipeline architectures. Moreover, LaKe uses external memories (SRAM and DRAM), whereas P4xos and Emu DNS use only on-chip memory.

Finally, three different design flows were used in the development of the applications: Verilog for LaKe, P4 (using P4-NetFPGA) for P4xos, and C# (using Emu) for Emu DNS. This results in differences in performance, resource usage, and potentially power consumption. We show in §4 and §5 that the effect of those is minimal, while other design decisions (e.g., external memory) have a significant effect on power consumption. The complexity of the designs is not comparable: Emu DNS is by far the simplest design. The scalability and modularity of LaKe makes it hard to compare to P4xos, yet both designs tend to many intricacies.

4 Power/Performance Evaluation

One of the main criticisms of in-network computing is that it is power hungry [56]. In this section we examine this claim, by evaluating the power consumption of the described applications under different loads. The power consumption of each application is evaluated for both software- and hardware-based implementations, including overheads, e.g., power supply unit. Our evaluation focuses on the following questions:

- What is the trade-off between power consumption and throughput of different applications?
- Is in-network computing less power efficient than host-based solutions?
- Does an in-network computing solution require high network utilization to justify its power consumption?

The results reported in this section do not report an absolute truth for in-network computing. Different applications will have different power consumption profiles. Different servers will implement different power efficiency optimizations, have a different number of cores and will achieve different peak throughput. Similarly, different smart NICs, FPGA cards, and programmable network devices will result in different performance and power consumption results. Yet, we are not trying to unravel the performance and power efficiency of specific designs. Rather, we try to gain understanding for *different* applications running on *similar* platforms.

4.1 Experimental Setup

The goal of our experiments was to measure the power consumption under different loads. We did not evaluate functionality or performance, which were part of the contributions of previous works.

Note that the setup for this evaluation differs from those in §9. An Intel Core i7-6700K 4-cores server, running at 4GHz, equipped with 64GB RAM, Intel X520 NIC, and Ubuntu 16.04 LTS (Linux kernel 4.13.0) was used for software-based evaluation. For hardware-based evaluation, the NIC was replaced by NetFPGA-SUME [86] card. For KVS evaluation, the Intel NIC turned out to be a performance bottleneck, and was therefore replaced by 10GE Mellanox NIC (MCX311A-XCCT).

We used OSNT [2] to send traffic, which enabled us to control data rates at very fine granularities and reproduce results. Average throughput was measured at the granularity of a second. We used an Endace DAG card 10X2-S to measure latency, measuring the isolated latency of the application under test, traffic source excluded. Power measurements were taken using a SHW 3A power meter [6].

4.2 Key-Value Store - Power/Performance

With LaKe, in contrast to other in-network computing use cases, the role of the server software is not eliminated by shifting functionality to hardware. LaKe serves as a first and second level cache. In the event of cache misses at both levels, the software services the request. We used Memcached (v1.5.1) as both the host-side software replying to queries missed in LaKe's cache, and as the software implementation we benchmark against. The power consumption evaluation of LaKe, therefore, includes the combined power consumption of the NetFPGA board and the server. Note that the NIC is taken out of the server for LaKe's evaluation, as LaKe replaces it.

We measure the power consumption of the KVS, starting with an idle system, and then gradually increasing the query rate until reaching peak performance. Peak performance is full line rate for LaKe and approximately 1Mpps for memcached. We verify that the CPU reaches full utilization on all 4-cores.

Figure 3(a) presents the power-to-throughput trade-off for the KVS. The x-axis shows the number of queries sent to the server every second, while the y-axis presents the power consumption of the server under such load. We show the power consumption for memcached (software only), LaKe within a server, and LaKe as a standalone platform, i.e., working outside a server and without the power consumption contributed by the hosting server. As the figure shows, the power consumption of the server while idle or under low utilization is just 39W, while LaKe draws 59W even when idle. However, the picture changes quickly as query rate increases. At less than 100Kpps, LaKe is already more power efficient than the software-based KVS, with the crossing point occurring around 80Kpps. Interestingly, we found that after replacing the Mellanox NIC with an Intel X520 NIC, the host became more power efficient; the crossing point moved to over 300Kpps. However, the maximum throughput the server achieves using the Intel NIC is lower.

LaKe has a high base power consumption, but the consumption does not increase significantly under load. Figure 3(a) shows the throughput up to 2Mpps. But, we note that LaKe reached full line rate performance, supporting over 13Mpps for the same power consumption.

4.3 Paxos - Power/Performance

We evaluated the power consumption of the Paxos leader and acceptor roles for three different use cases: the basic software implementation of libpaxos, the software implementation using DPDK, and P4xos on NetFPGA.

We start with an idle system, and gradually increase the message rate. The libpaxos software uses one core, and we verify that the core reached 100% utilization.

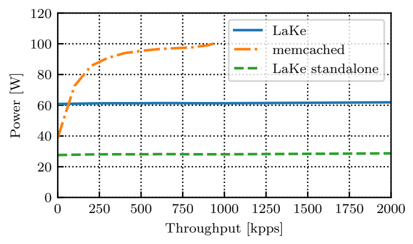
Figure 3(b) presents the power-to-throughput trade-off for Paxos. As with the KVS, the idle power consumption of the server is lower than the card, but as the query rate increases, P4xos (hardware) becomes more power efficient. As P4xos doesn't use the external memories on NetFPGA, its base power consumption is 10W lower than LaKe. The crossing point between software and hardware power efficiency is at 150K messages/sec.

Note that the power consumption for the DPDK implementation is high even under low load, and remains almost constant under an increasing load. This is as expected, since DPDK constantly polls. This illustrates that software design choices have a strong impact on power consumption, independent of the hardware platform.

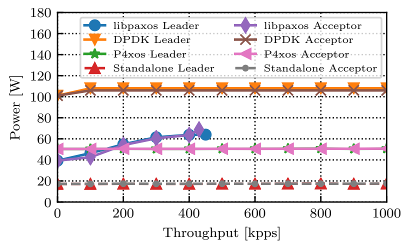
The power consumption results of P4xos in hardware include the power consumption of the server hosting the board. The power consumption of P4xos outside the server is 18.2W when idle, with the additional dynamic power consumption (under maximum load) being no more than 1.2W. Yet, it is not expected to have stand alone FPGA boards in a data center environment: the platforms require power supply, management and programming interfaces (e.g., for updates). Encasing such boards within a standard server enclosure is therefore an expected practice. Typically, multiple acceleration boards will share a single enclosure [8], reducing the per-board power consumption contribution to the system.

4.4 DNS - Power/Performance

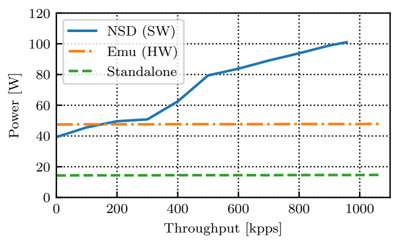
The peak performance of Emu DNS is roughly 1M requests served every second. This is comparable to the 956K requests we measure served by the software, and a result of Emu's non-pipelined nature. This case demonstrates aspects of power



(a) KVS



(b) Paxos



(c) DNS

Figure 3. Power vs throughput comparison of KVS (a), Paxos (b), and DNS (c). in-network computing becomes power efficient once query rate exceeds 80Kppsm 150Kpps and 150Kpps, respectively

efficiency where in-network computing does not provide significant performance benefits.

We measure the power consumption of Emu DNS, starting with an idle system, and gradually increasing the query rate until peak performance is reached.

The power consumption as a function of performance, shown in Figure 3(c), is almost identical to P4xos. The power consumption of Emu DNS is about 48W, starting at 47.5W and reaching less than 48W under full load. The idle server takes less than 40W, but less than 200Kpps are enough for the power consumption to exceed the hardware implementation. At peak throughput, the server draws twice the power of Emu DNS.

5 Lessons from an FPGA

In-network computing designs often offer significant performance improvements, but at the cost of bespoke functionality [46], small memory [73], or of a limited feature set [37]. In this section we build upon the modularity of LaKe (KVS) to explore the performance and power efficiency effects of such design decisions.

Beyond illustrating the effects of such design decisions, this section also highlights the challenge in comparing state-of-the-art in-network computing solutions. For example, the difference between a design that uses just a small on-chip memory, and one that mitigates a miss in the cache, can be an order of magnitude in performance and 66% in power consumption. We assert that future research should take greater care when catering for in-network computing design.

5.1 Clock Gating, Power Gating and Deactivating Modules

The power consumption of a hardware device depends on many aspects, from properties of the manufacturing process (static power, leakage) to aspects depending on activity (such as the effect of clock frequency).

For the purpose of our discussion, we focus on the case where the in-network computing platform is given (in our case, NetFPGA). The ASIC/FPGA device on the platform is set as well (in our case, Xilinx Virtex-7 690T FPGA). The operator can only change performance and power efficiency within these limitations.

We focus on three types of power-saving techniques: clock gating, power gating, and deactivating modules. Clock gating refers to the case where the clock to certain parts of a design is active only when activity is required. Power gating refers to a similar case where the power to specific parts of the design is disabled. As Virtex-7 does not support power gating, we compare to the case where the modules in question are eliminated from the design, but note that many more recent ASICs and FPGAs do support power gating. The last case, deactivating modules, refers to the case where modules are only used when needed (e.g., using one processing core instead of five), and are either idle or held in reset.

Figure 4 summarizes the effect of the different power-saving techniques for LaKe. As shown, the power consumption of an idle server (without a NetFPGA card) was roughly equivalent to the power consumption of a stand alone (host-less) NetFPGA card programmed with LaKe but also idle. This means that the basic power consumption of a stand-alone accelerator (including its power supply) can be roughly the same as a server. In §4 we refer to the power consumption of LaKe as the combined power of the NetFPGA platform and the server, as both build the complete multi-layered cache platform.

Clock gating to the LaKe module and the PEs earns less than 1W of power saving. The power contribution of each PE is also small, about 0.25W (power gating). The biggest contributor to power consumption is the external memories—no less than 10W. Reset to the external memory interfaces can save 40% of their power. Clock and power gating to the same interfaces is not supported.

5.2 Processing Cores

On FPGAs, and in particular for the case of LaKe (and Emu DNS), the cost of more logic is low. The power overhead of Lake's logic over the NetFPGA reference NIC is 2.2W, including five processing cores, interconnects and a packet classification module. In terms of FPGA resources, this translates to less than 3% of logical elements and on-chip memory resources. In return, each processing core can support up to

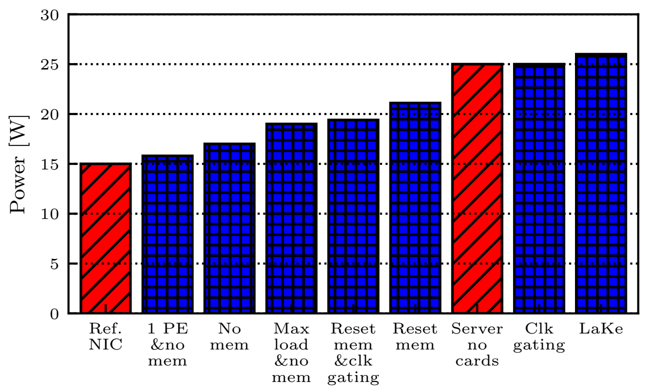


Figure 4. The effects of LaKe's design trade-offs on power consumption. Blue indicates LaKe's power consumption. Red refers to NetFPGA's NIC and an i7 Server.

3.3Mqps. There is a limit on the number of cores used, which is not the FPGA resources or power consumption, rather the interconnect between them and the memories, as well as the interconnect between them and the NIC data path.

5.3 Memories

Using off-chip memory is expensive: 4GB of DRAM memory costs 4.8W and 18MB of SRAM costs 6W[1]. There is obviously a gain here as well. 4GB of DRAM is enough to hold 33M entries of 64B value chunks and 268M entries of hash table entries. This is $\times 65k$ the number of entries using only on chip memory. The SRAM holds a list of up to 4.7M free memory chunks, $\times 32k$ the number of entries using on chip memory. Using external memories also affects the hit ratio in the LaKe L2 cache, and consequently on the latency. A hit in the on-chip cache takes no more than $1.4\mu s$, while a miss in the hardware will be $\times 10$ longer ($13.5\mu s$ median, $14.3\mu s$ 99thpercentile). Off chip memory adds a bit of latency compared to the on-chip cache, but saves significantly in comparison to software ($1.67\mu s$ median, $1.9\mu s$ 99thpercentile at 100Kqps, and up to 99thpercentile of $3\mu s$ at 10Mqps).

The trade-off here is clear, and depends on the number of expected keys: if you care about low latency, you should opt for the LaKe option using external memories, whereas if power is your concern, on-chip memory is the right way. Given that past work [7] had shown that in KVS the number of unique keys requested every hour is in the order of $10^9 - 10^{11}$, with the percentage of unique keys requested ranging from 3% to 35%, KVS applications would benefit from the use of external memories. On the other hand, use cases such as NetChain [37] will do better with on-chip memory.

5.4 Infrastructure

The cost of using a programmable card is absolute, yet the relative power within a host strongly depends on the system in which the card is installed. So far, we have focused on a light-weight platform using an i7 Intel CPU. In this system, the initial power-cost of an unused FPGA is higher than that of the server. For comparison, we consider a single 3.5GHz Intel Xeon E5-2637 v4 on a SuperMicro X10-DRG-Q motherboard. In this setup, the idle power consumption of the server, without a NIC, is 83W, meaning 20W more than the power consumption of LaKe running at full load in our base setup described in §5. The power difference of installing a NetFPGA card on this machine and running LaKe, P4xos, or Emu DNS is the same as with the base setup, because the power consumption of the board is constant. The peak power consumption of LaKe running on the Xeon server is, obviously, higher, as it combines the power consumption of LaKe, and the (higher) power consumption of the Xeon server replying to queries that are a miss in LaKe. Data centers, however, also deploy low-power instances, e.g. ARM based, and on such low-power platforms the relative power cost of the FPGA is higher.

The fact that power consumption is platform dependent applies also to the FPGA devices: FPGA from different vendors or from different generations will lead to a different power consumption. For example, Xilinx UltraScale+ achieves ×2.4 performance/Watt compared with Xilinx Virtex 7 [83].

6 Lessons from an ASIC

FPGA devices are very different from ASICs, both in terms of technology and the availability of power saving mechanisms. As a point of comparison, we also report experimental results on the power efficiency of in-network computing on an ASIC using Barefoot's Tofino chip [11].

The power consumption of programmable switches is the same or better than fixed-function devices. In other words, if a programmable switch is used strictly for networking, it does not incur additional power costs. However, the question remains: if we use a programmable switch to also support in-network computing applications, will there be additional power consumption costs? We explore this question below. Due to the large variance in power between different ASICs and ASIC vendors [52], we only report normalized power consumption.

For the evaluation, we ran the P4xos leader and acceptor roles on a Tofino, which required some architecture-specific changes to the code for memory accesses. We used a 1.28Tbps configuration of $32 \times 40Gbps$ for the Tofino, with a "snake" connectivity[2], which exercises all ports and enables testing the device at full capacity. The Paxos program is combined with a layer 2 forwarding program. Hence, the switch executes both standard switching and the consensus algorithm at the same time. We compare the power consumption of Tofino running only layer 2 forwarding to Tofino running the combined forwarding and P4xos. The power consumption of transceivers is excluded.

[1]These results are indicative of the NetFPGA SUME platform.

[2]Each output port is connected to the next input port, see [20].

The power consumption when idle is the same for both the ASIC with forwarding alone, and the ASIC with forwarding plus P4xos. As the packet rate increases, there is only a minor difference in power consumption between the two cases; running P4xos adds no more than 2% to the overall power consumption. While 2% may sound like a significant number in a data center, the diagnostic program supplied with Tofino (diag.p4) takes 4.8% more power than the layer 2 forwarding program under full load, more than twice that of P4xos.

While the power consumption of Tofino increases under load, the relative increase in power using P4xos is almost constant with the rate. Furthermore, in contrast to a server, where momentary power consumption can more than double itself (§4), the difference between the minimum and maximum consumption is less than 20%.

It is true that the power consumption of a server is less than that of the switch. Yet, as our experiment shows, adding in-network computing to networking equipment *already installed in a data center* has a negligible effect on the power consumption, while providing orders of magnitude improvement in throughput. Even at a relatively low utilization rate (10%), our implementation of P4xos on Tofino achieves $\times 1000$ higher Paxos throughput than a server, while its absolute dynamic power consumption[3] is 1/3 of the absolute dynamic power consumption of the server (at 180Kpps).

A common measure of power efficiency is operations per watt. While software base consensus achieves 10K's of message per watt, and FPGA based designs achieve 100K's of messages per watt, the ASIC implementation easily achieves 10M's of messages per watt.

7 Lessons from a Server

The evaluation in sections 4 and 5 was using an Intel Core i7-6700K 4-cores machine. We perform a limited evaluation to study the power consumption of a Xeon class server, more suitable to data center environments. The server that we use in the evaluation is ASUS ESC4000-G3S using two sockets of Intel Xeon E5-2660 v4, each CPU with 14 cores and base frequency of 2GHz.

We evaluate the power consumption of the CPU cores on the server under different loads, using a synthetic workload, without I/O, and monitor using running average power limit (RAPL). The power consumption of the server is 56W in idle, evenly divided between the sockets, and 134W under full load of all cores. The power consumption of the server jumps when even a single core is used, up to 91W. Not only the power consumption of the socket with the running core increases, but also of the second socket, almost equally. However, once the core is running, the overhead of an additional core running is small, in the order of 1W-2W. We provide further breakdown in our released dataset.

[3]The difference between idle power consumption and power consumption under a given load.

Interestingly, even at a low CPU core load, e.g., 10%, the power consumption of the server reaches 86W. Given the smaller overhead of running an application in the network, it becomes desirable even when workloads under-utilize a server's computer resources.

8 When to Use In-Network Computing

Niccolini et al. [60] define the energy consumption as:

$$E = P_d(f) \times T_d(W, f) + P_s \times T_s + P_i \times T_i \tag{1}$$

Where E is the energy consumption, P is power consumption, f is device frequency, W the number of processed packets, and T is the time at given state. $P_d(f) \times T_d(W, f)$ accounts for the energy used when actively processing packets. $P_s \times T_s$ is the energy required to transition from sleep state. Finally, $P_i \times T_i$ is the energy consumption at idle. Packet rate R is defined as $R = W/T_d$.

In-network computing should be used when E^S, the energy of a system running an application in software, exceeds E^N, the energy of a system running the same application within the network.

Below, we try to answer two questions:

- If you use standard network devices, should you start using programmable ones?
- If you use programmable network devices, when should you offload to the network?

For the first question, the dominant components will be P_i^S and P_i^N. Assuming the programmable network device is not used for in-network computing, the energy penalty of including it as part of normal network operation is the one to worry about.

For the second question, $P_i^N = P_i^S$, as the programmable device is the same. As in-network computing devices are part of the network, forwarding packets and providing networking functionality, their idle and sleep mode power is not changing regardless of the location of a workload. Here P_d^S and P_d^N become dominant components. At low data rates $P_d^N(R) > P_d^S(R)$ due to the additional power consumed by now active in-network computing logic in the device, but as R grows, $P_d^N(R) < P_d^S(R)$, as in-network computing is more power efficient at high utilization (§4,§5). The tipping point from the software to the network occurs when R reaches $P_d^N(R) = P_d^S(R)$.

9 In-Network Computing on Demand

In this paper, we argue that programmable network devices should be treated as one would treat other scheduled computing resources. Workloads can be assigned to network devices, and one should be able to reallocate the workloads to other computing resources. §8 provided an analysis describing when in-network computing can be optimally used, and next we discuss the *how*.

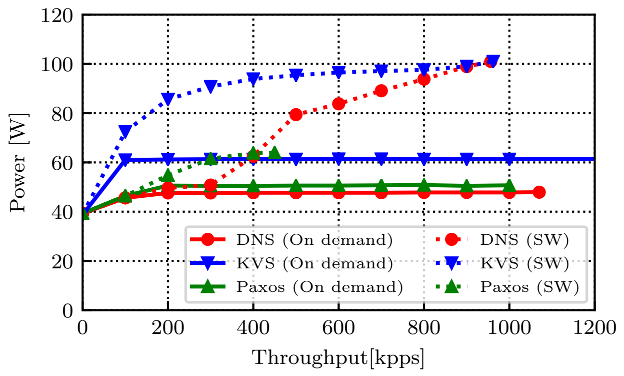


Figure 5. Power consumption of KVS, Paxos and DNS using in-network computing on demand. Solid lines indicate in-network computing on demand, and dashed lines indicate software-based solutions.

As there is no doubt that in-network computing offers significant performance benefits (§4), the question becomes *how can we benefit from the performance of in-network computing, without losing power efficiency?*

We propose *in-network computing on demand*, a scheme to dynamically shift computing between servers and the network, according to load and power consumption. This scheme is useful where identical applications run on the server and in the network, as in our examples. It can be applied to a wide range of applications, though possibly not all (as discussed later). It is also not applicable to bespoke in-network computing applications, which have no server-side equivalent.

The power consumption using in-network computing on demand is illustrated in Figure 5. As the figure shows, at low utilization power consumption is derived from the properties of the software-based system. As utilization increases, processing is shifted to the network, and the power consumption changes little with utilization.

We consider the communication cost associated with in-network computing on demand. Stateless applications will require no additional communication cost to run, whereas stateful application will have a communication cost that is bounded by the communication cost of shifting the application from one server to another. We discuss this further in §9.2. The networking device providing in-network computing services is expected to be en-route to a server running the application (otherwise it is not in-network computing, but standard offloading), meaning that no additional latency is introduced.

Two components are required to support in-network computing on demand. The first is a controller, deciding where the processing should be done and when the processing should be shifted between a server and the network. The second is an application-specific task, which may be null, in charge of the actual transition of an application. In §9.1 we discuss two approaches to implementing the controller, while §9.2 reflects on the application-specific requirements and evaluation.

9.1 In-Network Computing on Demand Controller

We propose two types of in-network computing on demand controllers: host-controlled and network-controlled. We present proofs-of-concept for both approaches, evaluate them and discuss trade-offs between the approaches. The network-controlled approach typically reacts faster, but must make its choices based on fewer parameters.

Network-Controlled In-Network Computing. The first controller design makes offloading decisions in the network hardware, based on the traffic load. The goal is to reduce load on the host as early as possible, to mitigate bottlenecks, and provide another layer of offloading (rather than encumbering the host with an additional controller). The control is not entirely automatic: all of its parameters are configurable.

The controller uses a pair of parameters to shift a workload from the host to the network. The first parameter is the average message rate that would trigger the transition, and the second is the averaging period (implemented as a sliding window). If the average message rate *of the accelerated application* exceeds the message rate threshold over the averaging period, the device transitions the workload to the network. A mirror pair of parameters is used to shift workloads from the network back to the host. Using two sets of parameters provides hysteresis, and attends to concerns of rapidly shifting workloads back-and-forth between the host and the network.

A disadvantage of this approach is that it does not take into account the actual power consumption of the host. It only has access to the packet rate. Different applications have very different power profiles [63], and there is no suitable heuristic that can be applied to the shifting thresholds. Our controller is implemented in 40 lines of code within the FPGA's classifier module, and consumes negligible resources (order of 0.1%).

Host-Controlled In-Network Computing. The second controller design makes offloading decisions at the host, using information such as the CPU usage and power consumption. A shift occurs when there is a clear power consumption benefit, and the offloading leads to a performance gain. A shift may also happen when computing demands exceed available resources, and the network provides extra compute capacity.

Like the network-based controller, the host-based controller maintains two sets of parameters: one for shifting the workload to the network, and one for shifting the workload back. As long as the application is running, the controller monitors its CPU usage. We also monitor the end-host's power consumption using running average power limit (RAPL). If the application exceeds a (programmable) power threshold set for offloading, and CPU usage is high, the controller shifts the workload to the network. Monitoring the power consumption alone is not sufficient, as a high power consumption can be triggered by multiple applications running on the same host.

As before, the information is inspected over time, avoiding harsh decisions based on spikes and outliers. In order to shift back to the host from the network, the controller needs information from the network (e.g., packet rate processed using in-network computing). Otherwise, the shift may be inefficient, or cause a workload to bounce back and forth. Our controller is implemented in 204 lines of code, and consumes only 0.3% CPU usage, mainly for performing RAPL reads.

The host-controlled approach provides better control and flexibility to the user. Yet, care needs to be taken when benchmarking a workload [57]. The algorithms used in this paper are naive, providing a proof of concept. They can be enhanced by more sophisticated algorithms. In energy proportional servers, energy efficiency is not linear, though power consumption still grows linearly with utilization [12], and algorithms such as those based on PEAS [81] may improve the energy consumption. These algorithms are beyond the scope of this paper and remain part of future work.

9.2 On Demand Applications

Key Value Store. LaKe shifts from the software to the network, as query rate demands. An application using LaKe remains oblivious to the shift. As LaKe natively acts as a NIC to all non-KVS traffic, at the start of the day all traffic can be sent and processed by the software. Both network-based and host-based controllers support the transitioning of KVS.

To support in-network computing on demand, and provide optimal power efficiency, LaKe's memories need to be held in reset and clock-gating to the logical modules should be enabled. Here, the triggering of a shift means that at first all memory accesses will be a miss, and queries will continue to be forwarded to the software, until the cache, both on and off chip, warms. Consequently, latency would start to drop, but query rate will be maintained. Enabling LaKe will not necessarily increase the throughput. As shown in Section 5, LaKe becomes power efficient at a low query rate that is also sustained by the software. Therefore, unless the query rate is externally increased, the throughput will not change.

Figure 6 demonstrates the transition from running in software to running on hardware, using host controlled in-network computing. The red line on each graph indicates the moment of transition. Clock gating and memories reset are not enabled in this experiment.

We maintain the same server as described in Section 4, however we replace OSNT with a similar server running a mutilate based [45] memcached client, using the Facebook "ETC" [7] arrival distribution. ChainerMN [1] (Chainer v4.4.0), a deep learning framework, is running as a second workload on the host, passing traffic through the same LaKe card. CPU power consumption is read from RAPL, and is increased due to ChainerMN. Transition is triggered after three seconds of sustained high load, and then again as ChainerMN stops. Throughput is reported by the hardware counter on the LaKe card. As Figure 6 shows, the transition from software to hardware had no effect on KVS throughput, not even momentarily. The latency of query-hit improves ten-fold within tens of microseconds.

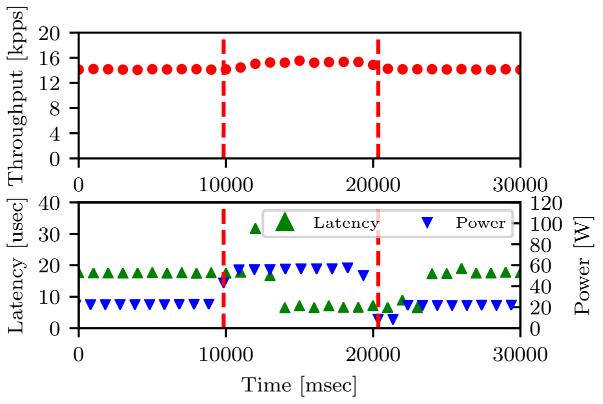


Figure 6. Transitioning KVS from the software to the network and back. The transitions are indicated by a red dashed line.

The approach described above has a minimal cost; there is about 5W gap between the power consumption of a NIC and that of LaKe with memories in reset and LaKe module's clock gated. We expect that on production designs and ASICs, this power consumption can be further minimized. Other approaches, such as partial reconfiguration of FPGA or keeping LaKe's cache warm all the time, are possible, but may result in a momentary traffic halt or reduced power saving, correspondingly. We therefore choose the approach that keeps LaKe programmed but inactive, in order to get the best of both performance and power efficiency worlds.

The eventual outcome of the on demand swap of KVS is shown in Figure 5. At low query rate, power consumption is low. When the traffic rate grows, in-network computing is enabled and the power consumption of LaKe becomes the dominant figure. We note that this graph is indicative of a case where all queries are (after warm up) hit in LaKe. In a case where many queries are a miss in the hardware, more power would be consumed by server attending to these queries. The worst case power consumption strongly depends on the workload [7].

Paxos. Modifying the deployment of Paxos is significantly more challenging than shifting a KVS. In fact, changing the members of Paxos—regardless of whether the roles are implemented in software or hardware—requires addressing two well-studied problems in distributed systems: *leader election* (i.e., shifting the role of the leader from one member to another) and *reconfiguration* (i.e., replacing one or more acceptors) [42]. In this paper, we focus on leader election, and describe one possible implementation. We focus on leader election because even at low message rates, a leader can become a bottleneck for end-to-end system performance. For reconfiguration, we point readers to protocols from prior work [35, 43, 44] which could be adapted for this setting.

In the Paxos protocol, the leader assigns monotonically increasing sequence numbers to client requests. Thus, there are two challenges that must be addressed for leader election. First, there must be a mechanism to identify a non-faulty leader from a set of candidates [42]. Second, the newly elected leader must learn the most recent sequence number.

For the purposes of shifting on demand, the mechanism for identifying the new leader is somewhat simplified when compared to the general case of coping with failures. We use a centralized controller to initiate the shift, depending on the workload. To actually implement the shift, the controller modifies switch forwarding rules to send messages to the new leader.

The new leader starts with an initial sequence number of 1 and must learn the next sequence number that it can use (i.e., a consensus instance that has not been used by the previous leader). We extended the acceptor logic to include the last-voted-upon sequence number whenever the acceptor responds to a message. Using this information, the new leader can determine the most recent not-yet-used sequence number.

However, there are a few subtleties that must be addressed. In the process of switching leaders, some consensus instances may have no decision (e.g., if not enough acceptors have voted in the consensus instance). Therefore, there may be gaps in the sequence numbers, which would prevent the protocol from making progress. To cope with that possibility, we use two mechanisms: a time out at the client and a time out at the learner.

The clients resend requests after a time-out period if the learner has not acknowledged. When a client re-tries, the newly elected leader will increment the sequence number. After a sufficient number of re-tries, the leader will eventually learn the new sequence number.

The learner will look for gaps in instance numbers after a time-out period. If it discovers a gap, then it will send a message to the newly elected leader, asking it to re-initiate that instance of Paxos. If that instance has previously been voted on, then the learners will receive a new value. Otherwise, they learn a no-op value.

Figure 7 shows the throughput and latency for consensus messages over time as we shift the leader from software to hardware and back. The red vertical lines indicate when a shift occurs. We see that the throughput increases and the latency is halved when the leader is implemented in hardware. The shift is triggered as the in-network computing controller replaces a forwarding rule to send client requests to the new leader. After both shifts, the new leader fails to propose until it learns the latest Paxos instance from the acceptors. Note that the throughput drops to zero for about 100 msec. This corresponds to the value of the client timeout. This timeout value was chosen arbitrarily, and could be reduced by tuning to the particular deployment.

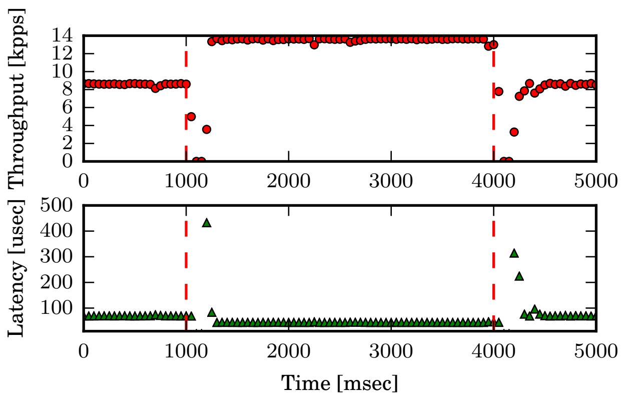


Figure 7. Transitioning Paxos leader from the software to the network and back. The transitions are indicated by a red dashed line.

DNS Server. Dynamically shifting DNS operation from software to the network is much the same as shifting KVS. The network-based controller is similar for both cases, due to the similarity between the DNS and KVS packet classifiers in the hardware. The host-based controller for DNS is simpler than that of the KVS, if the host is a dedicated DNS server that does not run other tasks in parallel. We omit the graphs for brevity.

Shifting a DNS server to a programmable ASIC, like Barefoot's Tofino, should also be possible. Prior work, such as NetChain [38] and NetCache[37], have already demonstrated the possibility of implementing a cache on Tofino, and DNS responses fit comfortably within the storage limits for values identified in their evaluation [38]. The biggest challenge would be supporting DNS queries that require parsing deeper than the maximum supported depth. However, in the worst case scenario, those queries could be treated as iterative requests.

9.3 Real Workloads

We investigate the applicability of in-network computing on demand by examining two real-world workloads, from Facebook's Dynamo [82] and the Google cluster data [68, 80]. The two workloads present two different use cases. In Dynamo, every cluster runs a unique workload. In Google, the workloads are heterogeneous.

Dynamo provides several important insights to this work. First, the power consumption of the webservers used by Facebook was significantly higher than that of the $i7$-based servers we used in Section 4, and doubled between generations of CPUs. Even at low loads of 10%, the dynamic power exceeded 30W (Westmere L5639) and 75W (Haswell E5-2678v3), more than the power consumption of a fully utilized smartNIC, let alone the power contributed by in-network computing. Furthermore, the power as a capping function for the workload was a driving force of Dynamo.

Second, Dynamo had shown that on a rack level, the power variation over three seconds was 12.8% at the 99^{th} percentile, and 26.6% over thirty seconds, with the median power variation being less than 5%. Caching—one of our case study applications—had a median power variation of 9.2% over sixty seconds, with a 99^{th} percentile of 26.2%. Other applications, such as a web server, had a median power variation of 37.2% and a 99^{th} percentile of 62.2%. The appropriateness of in-network computing depends on the power variance. If there is low power variance over the scheduling period, it will be safe to use in-network computing. If there is large variance, in-network computing on demand may be incorrect or inefficient, due to the variability of the power consumption over time. Dynamo does not provide CPU utilization information. Therefore, we cannot say if in-network computing would be beneficial at all times or only on demand.

We explore the Google trace to understand transient effects. The lack of power consumption information and the normalization of CPU core utilization limit our insights. In the Google trace, 90% of resource utilization is by jobs longer than two hours, though these jobs represent only 5% of the total number of jobs [68]. The long run times make these resource-hungry jobs candidates for offloading to the network. Moreover, the time scale for scheduling is long, fitting in-network computing on demand. Based on our observation that even a low utilization of a CPU core may draw more power than in-network computing (§7), we identify more than 1.39 million unique tasks in the trace that utilize for at least five minutes 10% or more of a CPU core, making them candidates for offloading. However, on average, every node within the cluster has 7.7 (normalized) CPU cores running such tasks within every five minutes sample period, diminishing the power saving benefit of in-network computing (assuming a limited number of workloads can be offloaded at a time).

The Google trace leads us to consider a different usage model: in-network computing on demand as load diminishes, rather than when load increases. When a multitude of jobs run on the same server, offloading to the network saves little power. However, as jobs end or are migrated from the server, moving the last (or first) job to the network will save power. The benefit of in-network computing remains applicable when latency or throughput, rather than power efficiency, are the targets, regardless of the load on the CPU.

9.4 Switch On-Demand?

We analyze the benefit in in-network computing on demand using switch ASIC. Unlike NIC, a switch serves multiple nodes, and in the next example we focus on a Top of Rack (ToR) switch, serving a rack of n nodes. We maintain the notations used in 8, where we asserted that the tipping point from the software to the network occurs when R reaches $P^N_d(R) = P^S_d(R)$.

Discussions with several silicon switch vendors confirmed that $P^N_i = P^S_i$ holds for these specific devices, and that switches' dynamic power consumption mostly increases linearly with data rate. The difference in power consumption between fixed-function and programmable switches is small, and favors the programmable switch (comparing Arista switches [3–5]). With switches designed to serve billions of packets per second, taking less than 5W per 100G port [5], a million queries will draw less than 1W[4]. This is unparalleled with the CPU power consumption under similar conditions (§4). Consequently $P^N_d(R)$ will equal $P^S_d(R)$ when R is almost zero, a result that is further strengthened when we remember that both $P^N_d(R)$ and $P^S_d(R)$ contain a shared power consumption element, the dynamic power consumption of the switch forwarding packets, which based on our discussions dominates $P^N_d(R)$.

This is slightly different to the case discussed for FPGA, where the power difference between a forwarding-only and workload-processing in not negligible, relative to the maximum power consumption of an FPGA NIC.

A different case consider the switch handling just some of the requests, and the rest are handled by the host (e.g., caching). Here the host may still consume significant power, possibly close to the saturation point. Under this scenario, the greatest benefit will come from performance, and it is a function of hit:miss ratio to define the efficiency of offloading on-demand.

9.5 Further Discussion

Latency. In-network computing reduces latency by design. By terminating a transaction within the network, instead of reaching the host, time can be saved. P4xos, Emu DNS and LaKe have all demonstrated significant latency improvement at the 99^{th} percentile. The tail latency of an in-network computing application depends on its implementation: a fully pipelined design that does not access external memories will have an almost-constant latency (±100ns on NetFPGA SUME) and additional pipeline stages required to implement an application will often have nano-second scale overhead. In these architectures, power consumption and latency will be independent. Latency variance due to congestion will be the result of switch-forwarding, and thus be experienced in a software-based environment as well. Access to external memories can lead to latency increase of hundreds of nanoseconds ([78], depending on hit and miss ratio) and additional power consumption. Still, it will be faster than going through PCIe to the host [88], processing there and accessing similar (power consuming) memories on the host. To conclude, where latency is the target, there is no need for in-network computing on demand, as in-network computing will provide lower latency.

Generality of In-Network Computing on Demand. In-network computing is not the magic cure-all for data centers' problems. Not all applications are suitable to be shifted to the

[4]assuming packets of 1500B or smaller

network, and the gain won't be the same for all. In-network computing is best suited for applications that are network-intensive, i.e., where the communication between hosts has a high toll on the CPU. Latency sensitive applications are also well suited for in-network computing. It is no coincidence that the most popular in-network computing applications to date are caching related [37, 38, 46]. Caching provides a large benefit in the common case, and a way to handle tail events. Other applications may find in-network computing on demand to be hard. For example, using Paxos in the network is hard, and doing it on demand is even harder. The effort of implementing an in-network computing solution may just be too high for some applications. Furthermore, each application may have a different power consumption gain, as shown in Figure 5.

In-Network Computing Alternatives. Readers may wonder if there are no simpler solutions to increasing application performance, rather than in-network computing. One solution, for example, is using multiple standard NICs in a server to achieve higher bandwidth [24, 30]. However, this approach comes at the cost of more NICs, increasing power and price. Alternatively, one may use multiple servers, or opt for a multi-socket or multi-node architecture [47]. These may be cost and power equivalent to an FPGA, a smartNIC, or an ASIC based design. But, their performance per watt is unlikely to match the ASIC-based solution. GPUs are efficient for offloading computation-heavy applications, but as they are not directly connected to the network, they are less suitable for network-intensive applications.

10 FPGA, SmartNIC or Switch?

"Where should I place my in-network computing application?" one may wonder. The answer is not conclusive. Today, a switch ASIC can provide both the highest performance and the highest performance per Watt. Running in a switch also cuts in half the number of (application-specific) packets through the switch: instead of both request and reply packets going through the switch, only one packet goes through: entering as the request, and coming out as the reply. A switch may not be, however, the cheapest solution, with a price tag of $\times 10$ or more compared to other solutions[5]. Using a switch as the place to implement in-network computing leads to other questions. What is the topology of the network? Can and will all messages travel through a specific (non addressed) switch? What are the implications of a switch failure (as opposed to a smartNIC/FPGA next to the end-host)? The answers are all application and data center dependent. Switches also have limited flexibility compared to other programmable devices: they have limited resources (per Gbps) and a vendor-provided target architecture, that may not fit all applications.

SmartNICs maintain the same power consumption as NICs, typically limiting their power consumption to 25W supplied through the PCI express slot, while achieving millions of operations per Watt, including external memories access [23, 54]. There are currently four architectural approaches to SmartNICs: FPGA based [27, 58, 74], ASIC based [59], combining ASIC and FPGA [55], and SoC based [53]. The FPGA-based design is closest to the NetFPGA-based design we discussed, while the ASIC-based smartNICs are closest to the switch-ASIC approach. SoC-based smartNICs are likely to provide the easiest trajectory for implementing in-network computing, but their resource and performance scalability is limited compared to other solutions, as they balance both programmable resources and processing cores, leading the networking requirement to face earlier the resource wall [87]. The power efficiency of SoC based solution depends on the type of integration between the data plane and the processing cores. Still, the introduction of SoC FPGA by manufacturers such as Intel is likely to increase the use of hybrid in-network computing solutions.

Between FPGA, smartNIC and ASIC, FPGA (and FPGA-based smartNICs) is likely to provide the poorest performance and performance per Watt, due to its general purpose nature. Yet, FPGA performance per Watt in real data centers is not significantly below ASIC. Azure's FPGA-based AccelNet SmartNIC [27] consumes 17W-19W (standalone) on a board supporting 40GE, providing close to 4Mpps/W for some use cases. This is slightly better, but on a par with, the FPGA-based power consumption reported in this work. The big advantage of FPGA, and FPGA-based platforms, is their flexibility—the ability to implement almost every application and to use (on a bespoke board) any interface, memory or storage device. ASIC-based smartNICs may not be suitable for **every** in-network function, but for many applications, they will provide a good trade-off of programmability, cost, maturity and power consumption.

11 Related and Future Work

Because green computing and power efficiency are extremely important to cloud computing [29], there has been a considerable amount of prior work (e.g., [9, 31]). Much of this work has been dedicated to power efficient computing and the assignment of workloads (e.g., [13, 41, 50]), including dynamic offloading to GPUs ([34]).

Although there has recently been an uptick in interest on in-network computing [22, 33, 36, 38, 69], the concept is not new. Previous systems have leveraged middleboxes, hardware accelerators, and offered network-as-a-service [17, 19, 28]. The main trend that distinguishes recent work is expressiveness, i.e., the ability to (relatively) easily shift applications from software to network. The introduction of programmable data planes [15] and domain specific languages for networking [14] has increased the potential impact of previous solutions, and their potential throughput by orders of magnitude (e.g., [37]).

[5]List prices, obtained from https://colfaxdirect.com

There are several notable examples of recent work on in-network computing, including caching applications [38, 46, 77], distributed systems services [21, 22, 33], monitoring and telemetry [40], and more. An advantage of these solutions is their integration within, or replacement of, commodity network devices, whether NICs or switches. This leads to a reduction in power overheads, as we have demonstrated, as well as reductions in cost, space, and equipment (e.g., cables).

In-network computing contrasts with acceleration solutions offered by cloud providers today, such as Amazon's F1 [10] and Google's TPUs [39]. While the power consumption of such platforms is not divorced from our results [39], the main difference is that these solutions are *additions* to the data center environment, whereas in-network computing takes advantage of equipment that is already part of the data center. In some cases, acceleration platforms are not networked-attached, but rather all transactions go through the CPU. This approach is ideal for applications that are computation intensive, but not suited to network intensive application

This paper has demonstrated that in-network computing can be power efficient, and we hope that our work will inspire future research. There are several research challenges that stem from this work. First, exploring the extent of in-network computing on demand and the ability to shift *any* application to the network. Second, power-aware data center scheduling that supports in-network computing on demand on a large scale. And last, but not least, reliability of in-network computing. With in-network computing providing ×1000 power saving, the future of green computing has just become better.

12 Conclusion

In-network computing is an emerging trend, but has been criticized as being impractical under real operating conditions. Through extensive evaluations on diverse case studies, this work explores that assumption. Our experiments show that although in-network computing is inefficient at low message rates, it quickly becomes more power efficient than host-based solutions under increased load. Inspired by this observation, we introduced in-network computing on demand, a scheme allowing to dynamically shift loads between software- and hardware- based solutions, always benefiting from the best power efficiency.

We make the results and a reproducible environment available on our website [76] and on github [75].

Acknowledgments

We thank the anonymous EuroSys reviewers and our shepherd, Gustavo Alonso, for their valuable feedback. This work was partially funded by JSPS Research Fellowship, JSPS KAKENHI Grant Number JP17J02958, Swiss National Science Foundation (200021_166132), Leverhulme Trust (ECF-2016-289), Isaac Newton Trust and a research award from Western Digital.

References

[1] Takuya Akiba, Keisuke Fukuda, and Shuji Suzuki. 2017. ChainerMN: Scalable Distributed Deep Learning Framework. In *Proceedings of Workshop on ML Systems in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*.

[2] Gianni Antichi, Muhammad Shahbaz, Yilong Geng, Noa Zilberman, Adam Covington, Marc Bruyere, Nick McKeown, Nick Feamster, Bob Felderman, Michaela Blott, Andrew W. Moore, and Philippe Owezarski. 2014. OSNT: Open source network tester. *IEEE Network* 28, 5 (2014), 6–12.

[3] Arista. 2018. 7050X3 Series 10/25/40/50/100G Data Center Switches. https://www.arista.com/assets/data/pdf/Datasheets/7050X3-Datasheet.pdf.

[4] Arista. 2018. 7060X and 7260X Series 10/25/40/50/100G Data Center Switches. https://www.arista.com/assets/data/pdf/Datasheets/7060X_7260X_DS.pdf.

[5] Arista. 2018. 7170 Series Programmable Data Center Switches. https://www.arista.com/assets/data/pdf/Datasheets/7170-Datasheet.pdf.

[6] System Artware. [n. d.]. *SHW 3A Watt hour meter*. http://www.system-artware.co.jp/shw3a.html.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.

[8] Amazon AWS. [n. d.]. *Amazon EC2 F1 Instances*. https://aws.amazon.com/ec2/instance-types/f1/[Online, accessed February 2019].

[9] J. Baliga, R. W. A. Ayre, K. Hinton, and R. S. Tucker. 2011. Green Cloud Computing: Balancing Energy in Processing, Storage, and Transport. *Proc. IEEE* 99, 1 (Jan 2011), 149–167.

[10] Jeff Bar. 2017. EC2 F1 Instances with FPGAs âĂŞ Now Generally Available. https://aws.amazon.com/blogs/aws/ec2-f1-instances-with-fpgas-now/-generally-available/.

[11] Barefoot Tofino 2018. Barefoot Tofino. https://www.barefootnetworks.com/products/brief-tofino/.

[12] Luiz André Barroso and Urs Hölzle. 2007. The case for energy-proportional computing. *Computer* 12 (2007), 33–37.

[13] R. Bianchini and R. Rajamony. 2004. Power and energy management for server systems. *Computer* 37, 11 (Nov 2004), 68–76.

[14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.

[15] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. *SIGCOMM Comput. Commun. Rev.* 43, 4 (Aug. 2013), 99–110.

[16] Mihai Budiu and Chris Dodd. 2016. The architecture of the P416 compiler. https://p4.org/assets/p4-ws-2017-p4-compiler.pdf.

[17] Brian Carpenter and Scott Brim. 2002. *Middleboxes: Taxonomy and issues*. Technical Report RFC 3234. IETF.

[18] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengil, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2017. Accelerating persistent neural networks at datacenter scale. In *HOTCHIPS*.

[19] Paolo Costa, Matteo Migliavacca, Peter R Pietzuch, and Alexander L Wolf. 2012. NaaS: Network-as-a-Service in the Cloud.. In *Hot-ICE*.

[20] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Hakim Weatherspoon, Marco Canini, Noa Zilberman, Fernando Pedone, and Robert Soulé. 2018. *P4xos: Consensus as a Network Service*. Research Report 2018-01. USI. http://www.inf.usi.ch/research_publication.htm?id=105

[21] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. 2016. Paxos Made Switch-y. *SIGCOMM Comput. Commun. Rev.* 46, 2 (May 2016), 18–24.

[22] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. Netpaxos: Consensus at network speed. In *SOSR*. ACM, 5.

[23] Sujal Das. 2016. The Arrival of SDN 2.0: SmartNIC Performance, COTS Server Efficiency and Open Networking. https://www.netronome.com/blog/the-arrival-of-sdn-20-smartnic-performance-cots-server-efficiency-and-open-networking/. [Online; accessed May 2018].

[24] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: exploiting parallelism to scale software routers. In *SOSP*. ACM, 15–28.

[25] DPDK 2015. DPDK. http://dpdk.org/.

[26] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *NSDI*. 401–414.

[27] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, and Mike Andrewartha et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI18*. 51–66.

[28] Ben Gelernter. 1998. Help design challenges in network computing. In *Proceedings of the 16th annual international conference on Computer documentation*. ACM, 184–193.

[29] Google. 2017. Environmental report: 2017 progress update.

[30] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM Computer Communication Review*, Vol. 40. ACM, 195–206.

[31] Andy Hopper and Andrew Rice. 2008. Computing for the future of the planet. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 366, 1881 (2008), 3685–3697.

[32] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: intelligent distributed storage. *VLDB* 10, 11 (2017), 1202–1213.

[33] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware.. In *NSDI*. 425–438.

[34] Muhammad Asim Jamshed, Jihyung Lee, Sangwoo Moon, Insu Yun, Deokjin Kim, Sungryoul Lee, Yung Yi, and KyoungSoo Park. [n. d.]. Kargus: a highly-scalable software-based intrusion detection system. In *ACM CCS*.

[35] Leander Jehl and Hein Meling. 2014. Asynchronous Reconfiguration for Paxos State Machines. In *ICDCN*. 119–133.

[36] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. 2018. Life in the fast lane: A line-rate linear road. In *SOSR*. ACM, 10.

[37] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *NSDI*. 35–49.

[38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *SOSP*. ACM, 121–136.

[39] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, and Gaurav et al. Agrawal. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *ISCA*. 1–12.

[40] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. 2015. In-band network telemetry via programmable dataplanes. In *SOSR*.

[41] Myungsun Kim, Kibeom Kim, James R. Geraci, and Seongsoo Hong. 2014. Utilization-aware Load Balancing for the Energy Efficient Operation of the Big.LITTLE Processor. In *DATE*. 223:1–223:4.

[42] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169.

[43] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2009. Vertical Paxos and Primary-backup Replication. In *PODC*. 312–313.

[44] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2010. Reconfiguring a State Machine. *SIGACT News* 41, 1 (March 2010), 63–73.

[45] Jacob Leverich. 2014. Mutilate: high-performance memcached load generator. https://github.com/leverich/mutilate

[46] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *SOSP*. 137–152.

[47] Hu Li. 2015. Introducing "Yosemite": the first open source modular chassis for high-powered microservers. https://code.facebook.com/posts/1616052405274961/introducing-yosemite-the-first-open-source-modular-chassis-for-\high-powered-microservers-/. [Online; accessed May 2018].

[48] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *USENIX OSDI*.

[49] libpaxos 2013. libpaxos. https://bitbucket.org/sciascid/libpaxos.

[50] Dumitrel Loghin, Bogdan Marius Tudor, Hao Zhang, Beng Chin Ooi, and Yong Meng Teo. 2015. A Performance Study of Big Data on Small Nodes. *VLDB* 8, 7 (Feb. 2015), 762–773.

[51] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In *ACM SIGCOMM*. 175–186. http://doi.acm.org/10.1145/2619239.2626311

[52] Mellanox. [n. d.]. *Mellanox Spectrum vs Broadcom and Cavium*. http://www.mellanox.com/img/products/switches/Mellanox-Spectrum-vs-Broadcom-and-Cavium.png[Online, accessed May 2018].

[53] Mellanox. 2018. BlueField SmartNIC Ethernet. http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic. [Online; accessed February 2019].

[54] Mellanox. 2018. ConnectX-6 EN Single/Dual-Port Adapter Supporting 200Gb/s Ethernet. http://www.mellanox.com/page/products_dyn?product_family=266&mtag=connectx_6_en_card. [Online; accessed May 2018].

[55] Mellanox. 2018. Mellanox Innova-2 Flex Open Programmable SmartNIC. http://www.mellanox.com/page/products_dyn?product_family=276&mtag=programmable_adapter_cards_innova2flex. [Online; accessed February 2019].

[56] Jeff Mogul and Jitu Padhye. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come HotNets-XVI Dialogue. https://conferences.sigcomm.org/hotnets/2017/dialogues/dialogue140.pdf.

[57] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. 2009. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices* 44, 3 (2009), 265–276.

[58] Napatech. [n. d.]. *Napatech SmartNIC for Virtualization Solutions*. https://www.napatech.com/products/napatech-smartnic-virtualization/[Online, accessed September 2018.

[59] Netronome. [n. d.]. *About Agilio SmartNICs*. [Online; accessed February 2019].

[60] Luca Niccolini, Gianluca Iannaccone, Sylvia Ratnasamy, Jaideep Chandrashekar, and Luigi Rizzo. 2012. Building a Power-Proportional Software Router.. In *ATC12*. 89–100.

[61] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *NSDI*.

[62] NLnet Labs Name Server Daemon 2018. NLnet Labs Name Server Daemon. https://www.nlnetlabs.nl/projects/nsd/.

[63] Eva Papadogiannaki, Lazaros Koromilas, Giorgos Vasiliadis, and Sotiris Ioannidis. 2017. Efficient Software Packet Processing on Heterogeneous and Asymmetric Hardware Architectures. *TON* 25, 3 (2017),

1593–1606.

[64] Marius Poke and Torsten Hoefler. 2015. Dare: High-performance state machine replication on rdma networks. In *HPDC*. ACM, 107–118.

[65] Diana Andreea Popescu, Noa Zilberman, and Andrew W Moore. 2017. *Characterizing the impact of network latency on cloud-based applicationsâĂŹ performance*. Technical Report UCAM-CL-TR-914. University of Cambridge. https://doi.org/10.17863/CAM.17588

[66] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *USENIX NSDI*.

[67] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *ISCA*. 13–24. http://dl.acm.org/citation.cfm?id=2665671.2665678

[68] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. 2011. *Google cluster-usage traces: format + schema*. Technical Report. Google Inc., Mountain View, CA, USA. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data.

[69] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-Network Computation is a Dumb Idea Whose Time Has Come. In *HOTNETS*. ACM, 150–156.

[70] Florian Schmidt, Oliver Hohlfeld, René Glebke, and Klaus Wehrle. 2015. Santa: Faster Packet Delivery for Commonly Wished Replies. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 597–598. http://doi.acm.org/10.1145/2829988.2790014

[71] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. 2015. Scalable 10Gbps TCP/IP stack architecture for reconfigurable hardware. In *FCCM*. IEEE, 36–43.

[72] Satnam Singh and David J. Greaves. 2008. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *Field-Programmable Custom Computing Machines*. IEEE, 3–12.

[73] Nik Sultana, Salvator Galea, David Greaves, Marcin Wojcik, Jonny Shipton, Richard Clegg, Luo Mai, Pietro Bressana, Robert Soulé, Richard Mortier, Paolo Costa, Peter Pietzuch, Jon Crowcroft, Andrew W. Moore, and Noa Zilberman. 2017. Emu: Rapid Prototyping of Networking Services. In *USENIX ATC*.

[74] Netcope Technologies. [n. d.]. *Netcope unveils Netcope P4 - a breakthrough in smart NIC performance and programmability*. https://www.netcope.com/en/company/press-center/press-releases/netcope-unveils-np4-a-breakthrough-in-smartnic[Online, accessed September 2018.

[75] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing on Demand, Repository. https://github.com/cucl-srg/INC-ondemand/.

[76] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. 2019. The Case For In-Network Computing on Demand, Website. https://www.cl.cam.ac.uk/research/srg/netos/projects/inc-ondemand/.

[77] Yuta Tokusashi and Hiroki Matsutani. 2017. Multilevel NoSQL Cache Combining In-NIC and In-Kernel Approaches. *IEEE Micro* 37, 5 (2017), 44–51.

[78] Yuta Tokusashi, Hiroki Matsutani, and Noa Zilberman. 2018. LaKe: The Power of In-Network Computing. In *ReConFig18*.

[79] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. 2017. P4FPGA: A Rapid Prototyping Framework for P4. In *ACM SOSR*.

[80] John Wilkes. 2011. More Google cluster data. Google research blog. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[81] Daniel Wong. 2016. Peak efficiency aware scheduling for highly energy proportional servers. In *ISCA*. IEEE, 481–492.

[82] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. 2016. Dynamo: Facebook's data center-wide power management system. In *ISCA*. IEEE, 469–480.

[83] Xilinx. [n. d.]. *Power Efficiency*. https://www.xilinx.com/products/technology/power.html[Online, accessed May 2018].

[84] Xilinx SDNet Development Environment 2014. Xilinx SDNet Development Environment. www.xilinx.com/sdnet.

[85] Peng Zheng, Theophilus Benson, and Chengchen Hu. 2018. P4Visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs. In *ACM CoNext*. 98–111. http://doi.acm.org/10.1145/3281411.3281436

[86] Noa Zilberman, Yuri Audzevich, G. Adam Covington, and Andrew W. Moore. 2014. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE MICRO* 34, 5 (Sept. 2014), 32–41.

[87] Noa Zilberman, Gabi Bracha, and Golan Schzukin. 2019. Stardust: Divide and conquer in the data center network. In *NSDI*. USENIX.

[88] Noa Zilberman, Matthew Grosvenor, Diana Andreea Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wójcik, and Andrew W Moore. 2017. Where has my time gone?. In *PAM*. Springer, 201–214.