Near-Data Processing

陈辉

Contents

I 文献 **2**

1 *Cloud-Scale-Acceleration-Architecture* **2**
1.1 Introduction 2
1.2 Hardware Architecture 2

2 *Practical-Near-Data-Processing-for-In-memory-Analytics-Frameworks* **3**

Part I
文献

1 *Cloud-Scale-Acceleration-Architecture*

(P1)

1.1 Introduction

Balance of Homogeneity and Specialization

Specialized accelerator should be programmable to adjust the diversity of cloud servers and rapid change. That's why we need FPGAs or GPUs.

This paper describes a cloud-scale FPGA-based acceleration architecture called *Configurable Cloud* which allows datapath of cloud communication to be accelerated with FPGAs. This datapath can include networking flows, **storage flows**, security operations and distributed applications.

We use a protocol called **LTL(Lightweight Transport Layer)** to make the remote FPGA resources apper closer than either a single local SSD access or the time to get through the host's networking stack.

1.2 Hardware Architecture

Designing Requirements

- Homogeneity.
- Fungibility.
- Flexibility.
  - *Local compute acceleration(through PCIe).* Local acceleration handles **hight-value scenarios** such as search rankingon each FPGA's host.
  - *Network acceleration.* e.g. intrusion detection, deep packet inspection and network encryption.
  - *Global application acceleration(through FPGA pools).* Through this, idle FPGAs can be used for distributed large-scale applications such as machine learning.
- **Physical restrictions**. e.g. physical space, power consumption, and temperature. That's **why we choose FPGAs instead of GPUs**.

2 *Practical-Near-Data-Processing-for-In-memory-Analytics-Frameworks*

See paper(P1) See presentation(P1)

A Cloud-Scale Acceleration Architecture

Adrian M. Caulfield Eric S. Chung Andrew Putnam
Hari Angepat Jeremy Fowers Michael Haselman Stephen Heil Matt Humphrey
Puneet Kaur Joo-Young Kim Daniel Lo Todd Massengill Kalin Ovtcharov
Michael Papamichael Lisa Woods Sitaram Lanka Derek Chiou Doug Burger

Microsoft Corporation

***Abstract*—Hyperscale datacenter providers have struggled to balance the growing need for specialized hardware (efficiency) with the economic benefits of homogeneity (manageability). In this paper we propose a new cloud architecture that uses reconfigurable logic to accelerate both network plane functions and applications. This Configurable Cloud architecture places a layer of reconfigurable logic (FPGAs) between the network switches and the servers, enabling network flows to be programmably transformed at line rate, enabling acceleration of local applications running on the server, and enabling the FPGAs to communicate directly, at datacenter scale, to harvest remote FPGAs unused by their local servers. We deployed this design over a production server bed, and show how it can be used for both service acceleration (Web search ranking) and network acceleration (encryption of data in transit at high-speeds). This architecture is much more scalable than prior work which used secondary rack-scale networks for inter-FPGA communication. By coupling to the network plane, direct FPGA-to-FPGA messages can be achieved at comparable latency to previous work, without the secondary network. Additionally, the scale of direct inter-FPGA messaging is much larger. The average round-trip latencies observed in our measurements among 24, 1000, and 250,000 machines are under 3, 9, and 20 microseconds, respectively. The Configurable Cloud architecture has been deployed at hyperscale in Microsoft's production datacenters worldwide.**

I. Introduction

Modern hyperscale datacenters have made huge strides with improvements in networking, virtualization, energy efficiency, and infrastructure management, but still have the same basic structure as they have for years: individual servers with multicore CPUs, DRAM, and local storage, connected by the NIC through Ethernet switches to other servers. At hyperscale (hundreds of thousands to millions of servers), there are significant benefits to maximizing homogeneity; workloads can be migrated fungibly across the infrastructure, and management is simplified, reducing costs and configuration errors.

Both the slowdown in CPU scaling and the ending of Moore's Law have resulted in a growing need for hardware specialization to increase performance and efficiency. However, placing specialized accelerators in a subset of a hyperscale infrastructure's servers reduces the highly desirable homogeneity. The question is mostly one of economics: whether it is cost-effective to deploy an accelerator in every new server, whether it is better to specialize a subset of an infrastructure's new servers and maintain an ever-growing number of configurations, or whether it is most cost-effective to do neither. Any specialized accelerator must be compatible with the target workloads through its deployment lifetime (e.g. six years: two years to design and deploy the accelerator and four years of server deployment lifetime). This requirement is a challenge given both the diversity of cloud workloads and the rapid rate at which they change (weekly or monthly). It is thus highly desirable that accelerators incorporated into hyperscale servers be programmable, the two most common examples being FPGAs and GPUs.

978-1-5090-3508-3/16/$31.00 © 2016 IEEE

Both GPUs and FPGAs have been deployed in datacenter infrastructure at reasonable scale without direct connectivity between accelerators [1], [2], [3]. Our recent publication described a medium-scale FPGA deployment in a production datacenter to accelerate Bing web search ranking using multiple directly-connected accelerators [4]. That design consisted of a rack-scale fabric of 48 FPGAs connected by a secondary network. While effective at accelerating search ranking, our first architecture had several significant limitations:

- The secondary network (a 6x8 torus) required expensive and complex cabling, and required awareness of the physical location of machines.
- Failure handling of the torus required complex re-routing of traffic to neighboring nodes, causing both performance loss and isolation of nodes under certain failure patterns.
- The number of FPGAs that could communicate directly, without going through software, was limited to a single rack (i.e. 48 nodes).
- The fabric was a limited-scale "bolt on" accelerator, which could accelerate applications but offered little for enhancing the datacenter infrastructure, such as networking and storage flows.

In this paper, we describe a new cloud-scale, FPGA-based acceleration architecture, which we call the *Configurable Cloud*, which eliminates all of the limitations listed above with a single design. This architecture has been — and is being — deployed in the majority of new servers in Microsoft's production datacenters across more than 15 countries and 5 continents. A Configurable Cloud allows the datapath of cloud communication to be accelerated with programmable hardware. This datapath can include networking flows, storage flows, security operations, and distributed (multi-FPGA) applications.

The key difference over previous work is that the accelera-

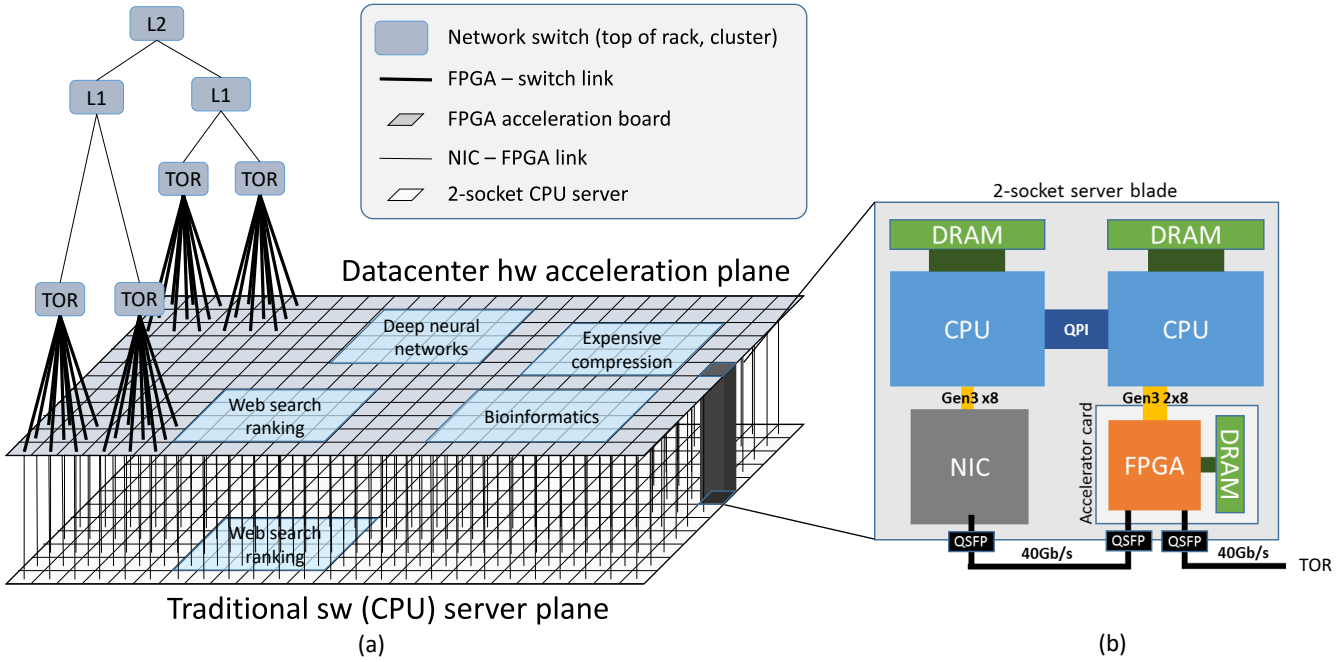

Fig. 1. (a) Decoupled Programmable Hardware Plane, (b) Server + FPGA schematic.

tion hardware is tightly coupled with the datacenter network—placing a layer of FPGAs between the servers' NICs and the Ethernet network switches. Figure 1b shows how the accelerator fits into a host server. All network traffic is routed through the FPGA, allowing it to accelerate high-bandwidth network flows. An independent PCIe connection to the host CPUs is also provided, allowing the FPGA to be used as a local compute accelerator. The standard network switch and topology removes the impact of failures on neighboring servers, removes the need for non-standard cabling, and eliminates the need to track the physical location of machines in each rack.

While placing FPGAs as a network-side "bump-in-the-wire" solves many of the shortcomings of the torus topology, much more is possible. By enabling the FPGAs to generate and consume their own networking packets independent of the hosts, each and every FPGA in the datacenter can reach every other one (at a scale of hundreds of thousands) in a small number of microseconds, without any intervening software. This capability allows hosts to use remote FPGAs for acceleration with low latency, improving the economics of the accelerator deployment, as hosts running services that do not use their local FPGAs can donate them to a global pool and extract value which would otherwise be stranded. Moreover, this design choice essentially turns the distributed FPGA resources into an independent computer in the datacenter, at the same scale as the servers, that physically shares the network wires with software. Figure 1a shows a logical view of this plane of computation.

This model offers significant flexibility. From the local perspective, the FPGA is used as a compute or a network accelerator. From the global perspective, the FPGAs can be managed as a large-scale pool of resources, with acceleration services mapped to remote FPGA resources. Ideally, servers not using all of their local FPGA resources can donate those resources to the global pool, while servers that need additional resources can request the available resources on remote servers. Failing nodes are removed from the pool with replacements quickly added. As demand for a service grows or shrinks, a global manager grows or shrinks the pools correspondingly. Services are thus freed from having a fixed ratio of CPU cores per FPGAs, and can instead allocate (or purchase, in the case of IaaS) only the resources of each type needed.

Space limitations prevent a complete description of the management policies and mechanisms for the global resource manager. Instead, this paper focuses first on the hardware architecture necessary to treat remote FPGAs as available resources for global acceleration pools. We describe the communication protocols and mechanisms that allow nodes in a remote acceleration service to connect, including a protocol called LTL (Lightweight Transport Layer) that supports lightweight connections between pairs of FPGAs, with mostly lossless transport and extremely low latency (small numbers of microseconds). This protocol makes the datacenter-scale remote FPGA resources appear closer than either a single local SSD access or the time to get through the host's networking stack. Then, we describe an evaluation system of 5,760 servers which we built and deployed as a precursor to hyperscale production deployment. We measure the performance characteristics of the system, using web search and network flow encryption as examples. We show that significant gains in efficiency are possible, and that this new architecture enables a much broader and more robust architecture for the acceleration

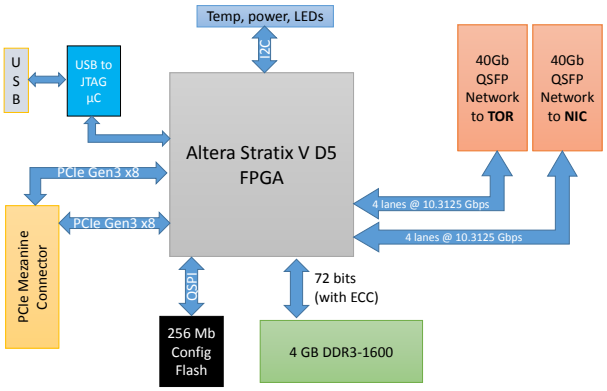

Fig. 2. Block diagram of the major components of the accelerator board.

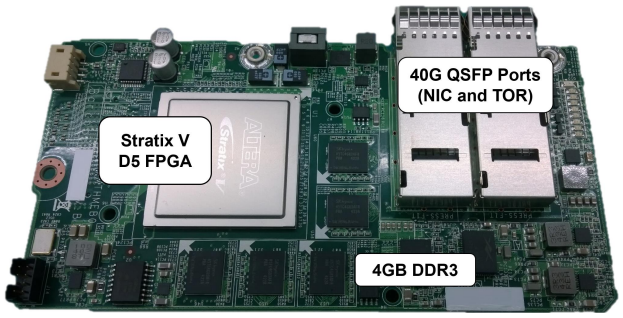

Fig. 3. Photograph of the manufactured board. The DDR channel is implemented using discrete components. PCIe connectivity goes through a mezzanine connector on the bottom side of the board (not shown).

of hyperscale datacenter services.

II. Hardware Architecture

There are many constraints on the design of hardware accelerators for the datacenter. Datacenter accelerators must be highly manageable, which means having few variations or versions. The environments must be largely homogeneous, which means that the accelerators must provide value across a plurality of the servers using it. Given services' rate of change and diversity in the datacenter, this requirement means that a single design must provide positive value across an extremely large, homogeneous deployment.

The solution to addressing the competing demands of homogeneity and specialization is to develop accelerator architectures which are programmable, such as FPGAs and GPUs. These programmable architectures allow for hardware homogeneity while allowing fungibility via software for different services. They must be highly *flexible* at the system level, in addition to being programmable, to justify deployment across a hyperscale infrastructure. The acceleration system we describe is sufficiently flexible to cover three scenarios: local compute acceleration (through PCIe), network acceleration, and global application acceleration, through configuration as pools of remotely accessible FPGAs. Local acceleration handles high-value scenarios such as search ranking acceleration where every server can benefit from having its own FPGA. Network acceleration can support services such as intrusion detection, deep packet inspection and network encryption which are critical to IaaS (e.g. "rental" of cloud servers), and which have such a huge diversity of customers that it makes it difficult to justify local compute acceleration alone economically. Global acceleration permits accelerators unused by their host servers to be made available for large-scale applications, such as machine learning. This decoupling of a 1:1 ratio of servers to FPGAs is essential for breaking the "chicken and egg" problem where accelerators cannot be added until enough applications need them, but applications will not rely upon the accelerators until they are present in the infrastructure. By decoupling the servers and FPGAs, software services that demand more FPGA capacity can harness spare FPGAs from other services that are slower to adopt (or do not require) the accelerator fabric.

In addition to architectural requirements that provide sufficient flexibility to justify scale production deployment, there are also physical restrictions in current infrastructures that must be overcome. These restrictions include strict power limits, a small physical space in which to fit, resilience to hardware failures, and tolerance to high temperatures. For example, the accelerator architecture we describe is the widely-used OpenCompute server that constrained power to 35W, the physical size to roughly a half-height half-length PCIe expansion card (80mm x 140 mm), and tolerance to an inlet air temperature of 70°C at 160 lfm airflow. These constraints make deployment of current GPUs impractical except in special HPC SKUs, so we selected FPGAs as the accelerator.

We designed the accelerator board as a standalone FPGA board that is added to the PCIe expansion slot in a production server SKU. Figure 2 shows a schematic of the board, and Figure 3 shows a photograph of the board with major components labeled. The FPGA is an Altera Stratix V D5, with 172.6K ALMs of programmable logic. The FPGA has one 4 GB DDR3-1600 DRAM channel, two independent PCIe Gen 3 x8 connections for an aggregate total of 16 GB/s in each direction between the CPU and FPGA, and two independent 40 Gb Ethernet interfaces with standard QSFP+ connectors. A 256 Mb Flash chip holds the known-good *golden image* for the FPGA that is loaded on power on, as well as one application image.

To measure the power consumption limits of the entire FPGA card (including DRAM, I/O channels, and PCIe), we developed a power virus that exercises nearly all of the FPGA's interfaces, logic, and DSP blocks—while running the card in a thermal chamber operating in worst-case conditions (peak ambient temperature, high CPU load, and minimum airflow due to a failed fan). Under these conditions, the card consumes 29.2W of power, which is well within the 32W TDP limits for a card running in a single server in our datacenter, and below the max electrical power draw limit of 35W.

The dual 40 Gb Ethernet interfaces on the board could allow for a private FPGA network as was done in our previous

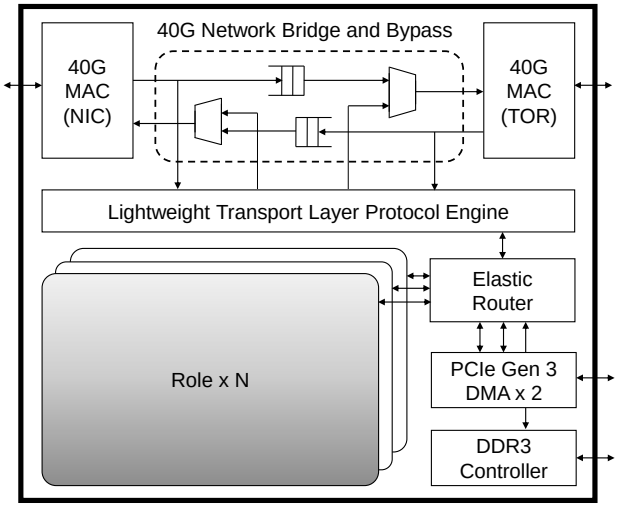

Fig. 4. The Shell Architecture in a Single FPGA.

	ALMs	MHz
Role	55340 (32%)	175
40G MAC/PHY (TOR)	9785 (6%)	313
40G MAC/PHY (NIC)	13122 (8%)	313
Network Bridge / Bypass	4685 (3%)	313
DDR3 Memory Controller	13225 (8%)	200
Elastic Router	3449 (2%)	156
LTL Protocol Engine	11839 (7%)	156
LTL Packet Switch	4815 (3%)	-
PCIe DMA Engine	6817 (4%)	250
Other	8273 (5%)	-
Total Area Used	**131350 (76%)**	-
Total Area Available	172600	-

Fig. 5. Area and frequency breakdown of production-deployed image with remote acceleration support.

work [4], but this configuration also allows the FPGA to be wired as a "bump-in-the-wire", sitting between the network interface card (NIC) and the top-of-rack switch (ToR). Rather than cabling the standard NIC directly to the ToR, the NIC is cabled to one port of the FPGA, and the other FPGA port is cabled to the ToR, as we previously showed in Figure 1b.

Maintaining the discrete NIC in the system enables us to leverage all of the existing network offload and packet transport functionality hardened into the NIC. This simplifies the minimum FPGA code required to deploy the FPGAs to simple bypass logic. In addition, both FPGA resources and PCIe bandwidth are preserved for acceleration functionality, rather than being spent on implementing the NIC in soft logic.

Unlike [5], both the FPGA and NIC have separate connections to the host CPU via PCIe. This allows each to operate independently at maximum bandwidth when the FPGA is being used strictly as a local compute accelerator (with the FPGA simply doing network bypass). This path also makes it possible to use custom networking protocols that bypass the NIC entirely when desired.

One potential drawback to the bump-in-the-wire architecture is that an FPGA failure, such as loading a buggy application, could cut off network traffic to the server, rendering the server unreachable. However, unlike a torus or mesh network, failures in the bump-in-the-wire architecture do not degrade any neighboring FPGAs, making the overall system more resilient to failures. In addition, most datacenter servers (including ours) have a side-channel management path that exists to power servers on and off. By policy, the known-good golden image that loads on power up is rarely (if ever) overwritten, so power cycling the server through the management port will bring the FPGA back into a good configuration, making the server reachable via the network once again.

A. Shell architecture

Within each FPGA, we use the partitioning and terminology we defined in prior work [4] to separate the application logic (Role) from the common I/O and board-specific logic (Shell) used by accelerated services. Figure 4 gives an overview of this architecture's major shell components, focusing on the network. In addition to the Ethernet MACs and PHYs, there is an intra-FPGA message router called the Elastic Router (ER) with virtual channel support for allowing multiple Roles access to the network, and a Lightweight Transport Layer (LTL) engine used for enabling inter-FPGA communication. Both are described in detail in Section V.

The FPGA's location as a bump-in-the-wire between the network switch and host means that it must always pass packets between the two network interfaces that it controls. The shell implements a bridge to enable this functionality, shown at the top of Figure 4. The shell provides a tap for FPGA roles to inject, inspect, and alter the network traffic as needed, such as when encrypting network flows, which we describe in Section III.

Full FPGA reconfiguration briefly brings down this network link, but in most cases applications are robust to brief network outages. When network traffic cannot be paused even briefly, partial reconfiguration permits packets to be passed through even during reconfiguration of the role.

Figure 5 shows the area and clock frequency of the shell IP components used in the production-deployed image. In total, the design uses 44% of the FPGA to support all shell functions and the necessary IP blocks to enable access to remote pools of FPGAs (i.e., LTL and the Elastic Router). While a significant fraction of the FPGA is consumed by a few major shell components (especially the 40G PHY/MACs at 14% and the DDR3 memory controller at 8%), enough space is left for the role(s) to provide large speedups for key services, as we show in Section III. Large shell components that are stable for the long term are excellent candidates for hardening in future generations of datacenter-optimized FPGAs.

B. Datacenter Deployment

To evaluate the system architecture and performance at scale, we manufactured and deployed 5,760 servers containing this accelerator architecture and placed it into a production datacenter. All machines were configured with the shell described above. The servers and FPGAs were stress tested using the power virus workload on the FPGA and a standard burn-in test for the server under real datacenter environmental conditions. The servers all passed, and were approved for production use in the datacenter.

We brought up a production Bing web search ranking service on the servers, with 3,081 of these machines using the FPGA for local compute acceleration, and the rest used for other functions associated with web search. We mirrored live traffic to the bed for one month, and monitored the health and stability of the systems as well as the correctness of the ranking service. After one month, two FPGAs had hard failures, one with a persistently high rate of single event upset (SEU) errors in the configuration logic, and the other with an unstable 40 Gb network link to the NIC. A third failure of the 40 Gb link to the TOR was found not to be an FPGA failure, and was resolved by replacing a network cable. Given aggregate datacenter failure rates, we deemed the FPGA-related hardware failures to be acceptably low for production.

We also measured a low number of soft errors, which were all correctable. Five machines failed to train to the full Gen3 x8 speeds on the secondary PCIe link. There were eight total DRAM calibration failures which were repaired by reconfiguring the FPGA. The errors have since been traced to a logical error in the DRAM interface rather than a hard failure. Our shell scrubs the configuration state for soft errors and reports any flipped bits. We measured an average rate of one bit-flip in the configuration logic every 1025 machine days. While the scrubbing logic often catches the flips before functional failures occur, at least in one case there was a role hang that was likely attributable to an SEU event. Since the scrubbing logic completes roughly every 30 seconds, our system recovers from hung roles automatically, and we use ECC and other data integrity checks on critical interfaces, the exposure of the ranking service to SEU events is low. Overall, the hardware and interface stability of the system was deemed suitable for scale production.

In the next sections, we show how this board/shell combination can support local application acceleration while simultaneously routing all of the server's incoming and outgoing network traffic. Following that, we show network acceleration, and then acceleration of remote services.

III. Local Acceleration

As we described earlier, it is important for an at-scale datacenter accelerator to enhance local applications *and* infrastructure functions for different domains (e.g. web search and IaaS). In this section we measure the performance of our system on a large datacenter workload.

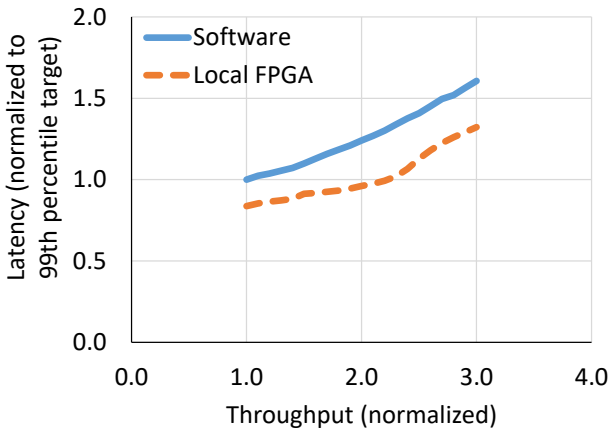

Fig. 6. 99% Latency versus Throughput of ranking service queries running on a single server, with and without FPGAs enabled.

A. Bing Search Page Ranking Acceleration

We describe Bing web search ranking acceleration as an example of using the local FPGA to accelerate a large-scale service. This example is useful both because Bing search is a large datacenter workload, and since we had described its acceleration in depth on the Catapult v1 platform [4]. At a high level, most web search ranking algorithms behave similarly; query-specific features are generated from documents, processed, and then passed to a machine learned model to determine how relevant the document is to the query.

Unlike in [4], we implement only a subset of the feature calculations (typically the most expensive ones), and neither compute post-processed synthetic features nor run the machine-learning portion of search ranking on the FPGAs. We do implement two classes of features on the FPGA. The first is the traditional finite state machines used in many search engines (e.g. "count the number of occurrences of query term two"). The second is a proprietary set of features generated by a complex dynamic programming engine.

We implemented the selected features in a *Feature Functional Unit* (FFU), and the *Dynamic Programming Features* in a separate DPF unit. Both the FFU and DPF units were built into a shell that also had support for execution using remote accelerators, namely the ER and LTL blocks as described in Section V. This FPGA image also, of course, includes the network bridge for NIC-TOR communication, so all the server's network traffic is passing through the FPGA while it is simultaneously accelerating document ranking. The pass-through traffic and the search ranking acceleration have no performance interaction.

We present results in a format similar to the Catapult results to make direct comparisons simpler. We are running this image on a full production bed consisting of thousands of servers. In a production environment, it is infeasible to simulate many different points of query load as there is substantial infrastructure upstream that only produces requests at the rate of arrivals. To produce a smooth distribution with repeatable results, we used a single-box test with a stream

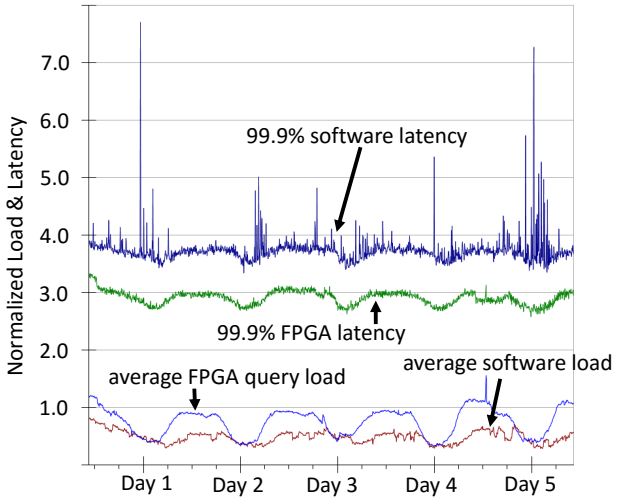

Fig. 7. Five day query throughput and latency of ranking service queries running in production, with and without FPGAs enabled.

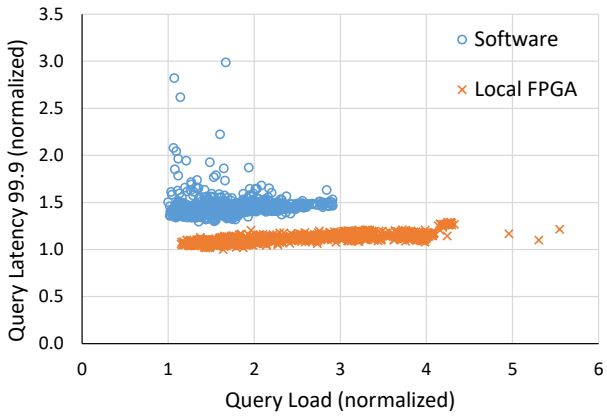

Fig. 8. Query 99.9% Latency vs. Offered Load.

of 200,000 queries, and varied the arrival rate of requests to measure query latency versus throughput. Figure 6 shows the results; the curves shown are the measured latencies of the 99% slowest queries at each given level of throughput. Both axes show normalized results.

We have normalized both the target production latency and typical average throughput in software-only mode to 1.0. The software is well tuned; it can achieve production targets for throughput at the required 99th percentile latency. With the single local FPGA, at the target 99th percentile latency, t he throughput can be safely increased by 2.25x, which means that fewer than half as many servers would be needed to sustain the target throughput at the required latency. Even at these higher loads, the FPGA remains underutilized, as the software portion of ranking saturates the host server before the FPGA is saturated. Having multiple servers drive fewer FPGAs addresses the underutilization of the FPGAs, which is the goal of our remote acceleration model.

Production Measurements: We have deployed the FPGA accelerated ranking service into production datacenters at scale and report end-to-end measurements below.

Figure 7 shows the performance of ranking service running in two production datacenters over a five day period, one with FPGAs enabled and one without (both datacenters are of identical scale and configuration, with the exception of the FPGAs). These results are from live production traffic, not on a synthetic workload or mirrored traffic. The top two bars show the normalized tail query latencies at the 99.9th percentile (aggregated across all servers over a rolling time window), while the bottom two bars show the corresponding query loads received at each datacenter. As load varies throughout the day, the queries executed in the software-only datacenter experience a high rate of latency spikes, while the FPGA-accelerated queries have much lower, tighter-bound latencies, despite seeing much higher peak query loads.

Figure 8 plots the load versus latency over the same 5-day period for the two datacenters. Given that these measurements were performed on production datacenters and traffic, we were unable to observe higher loads on the software datacenter (vs. the FPGA-enabled datacenter) due to a dynamic load balancing mechanism that caps the incoming traffic when tail latencies begin exceeding acceptable thresholds. Because the FPGA is able to process requests while keeping latencies low, it is able to absorb more than twice the offered load, while executing queries at a latency that never exceeds the software datacenter **at any load**.

IV. Network Acceleration

The bump-in-the-wire architecture was developed to enable datacenter networking applications, the first of which is host-to-host line rate encryption/decryption on a per flow basis. As each packet passes from the NIC through the FPGA to the ToR, its header is examined to determine if it is part of an encrypted flow that was previously set up by software. If it is, the software-provided encryption key is read from internal FPGA SRAM or the FPGA-attached DRAM and is used to encrypt or decrypt the packet. Thus, once the encrypted flows are set up, there is no load on the CPUs to encrypt or decrypt the packets; encryption occurs transparently from software's perspective, which sees all packets as unencrypted at the end points.

The ability to offload encryption/decryption at network line rates to the FPGA yields significant CPU savings. According to Intel [6], its AES GCM-128 performance on Haswell is 1.26 cycles per byte for encrypt and decrypt each. Thus, at a 2.4 GHz clock frequency, 40 Gb/s encryption/decryption consumes roughly five cores. Different standards, such as 256b or CBC are, however, significantly slower. In addition, there is sometimes a need for crypto hash functions, further reducing performance. For example, AES-CBC-128-SHA1 is needed for backward compatibility for some software stacks and consumes at least fifteen cores to achieve 40 Gb/s full duplex. Of course, consuming fifteen cores just for crypto is impractical on a typical multi-core server, as there would be that many fewer cores generating traffic. Even five cores of

savings is significant when every core can otherwise generate revenue.

Our FPGA implementation supports full 40 Gb/s encryption and decryption. The worst case half-duplex FPGA crypto latency for AES-CBC-128-SHA1 is 11 μs for a 1500B packet, from first flit to first flit. In software, based on the Intel numbers, it is approximately 4 μs. AES-CBC-SHA1 is, however, especially difficult for hardware due to tight dependencies. For example, AES-CBC requires processing 33 packets at a time in our implementation, taking only 128b from a single packet once every 33 cycles. GCM latency numbers are significantly better for FPGA since a single packet can be processed with no dependencies and thus can be perfectly pipelined.

The software performance numbers are Intel's very best numbers that do not account for the disturbance to the core, such as effects on the cache if encryption/decryption is blocked, which is often the case. Thus, the real-world benefits of offloading crypto to the FPGA are greater than the numbers presented above.

V. Remote Acceleration

In the previous section, we showed that the Configurable Cloud acceleration architecture could support local functions, both for Bing web search ranking acceleration and accelerating networking/infrastructure functions, such as encryption of network flows. However, to treat the acceleration hardware as a global resource and to deploy services that consume more than one FPGA (e.g. more aggressive web search ranking, large-scale machine learning, and bioinformatics), communication among FPGAs is crucial. Other systems provide explicit connectivity through secondary networks or PCIe switches to allow multi-FPGA solutions. Either solution, however, limits the scale of connectivity. By having FPGAs communicate directly through the datacenter Ethernet infrastructure, large scale and low latency are both achieved. However, these communication channels have several requirements:

- They cannot go through software protocol stacks on the CPU due to their long latencies.
- They must be resilient to failures and dropped packets, but should drop packets rarely.
- They should not consume significant FPGA resources.

The rest of this section describes the FPGA implementation that supports cross-datacenter, inter-FPGA communication and meets the above requirements. There are two major functions that must be implemented on the FPGA: the inter-FPGA communication engine and the intra-FPGA router that coordinates the various flows of traffic on the FPGA among the network, PCIe, DRAM, and the application roles. We describe each below.

A. Lightweight Transport Layer

We call the inter-FPGA network protocol LTL, for Lightweight Transport Layer. This protocol, like previous work [7], uses UDP for frame encapsulation and IP for routing packets across the datacenter network. Low-latency communication demands infrequent packet drops and infrequent packet reorders. By using "lossless" traffic classes provided in datacenter switches and provisioned for traffic like RDMA and FCoE, we avoid most packet drops and reorders. Separating out such traffic to their own classes also protects the datacenter's baseline TCP traffic. Since the FPGAs are so tightly coupled to the network, they can react quickly and efficiently to congestion notification and back off when needed to reduce packets dropped from incast patterns.

Figure 9 gives a block diagram of the LTL protocol engine used to support the inter-FPGA network protocol. At the endpoints, the LTL protocol engine uses an ordered, reliable connection-based interface with statically allocated, persistent connections, realized using send and receive connection tables. The static allocation and persistence (until they are deallocated, of course) reduces latency for inter-FPGA and inter-service messaging, since once established they can communicate with low latency. Reliable messaging also reduces protocol latency. Although datacenter networks are already fairly reliable, LTL provides a strong reliability guarantee via an ACK/NACK based retransmission scheme. Outgoing packets are buffered and tracked in an unacknowledged frame queue until their receipt is acknowledged by the receiver (see Ack Generation and Ack Receiver in Figure 9). Timeouts trigger retransmission of unACKed packets. In some cases, such as when packet reordering is detected, NACKs are used to request timely retransmission of particular packets without waiting for a timeout. Timeouts can also be used to identify failing nodes quickly, if ultra-fast reprovisioning of a replacement is critical to the higher-level service. The exact timeout value is configurable, and is currently set to 50 μsec.

Datacenter networks handle multiple traffic classes and protocols, some of which expect near-lossless behavior. FPGAs routing traffic between the server's NIC and TOR, as a bump-in-the-wire, must not interfere with the expected behavior of these various traffic classes. To that end, the LTL Protocol Engine shown in Figure 4 allows roles to send and receive packets from the network without affecting–and while supporting–existing datacenter protocols.

To achieve both requirements, the tap supports per-flow congestion management, traffic class based flow control, and bandwidth limiting via random early drops. It also performs basic packet classification and buffering to map packets to classes. Our LTL implementation is capable of generating and responding to Priority Flow Control [8] frames to pause traffic on lossless traffic classes. LTL also implements the DC-QCN[9] end-to-end congestion control scheme. In combination, these features allow the FPGA to safely insert and remove packets from the network without disrupting existing flows and without host-side support.

We measured the end-to-end round-trip latency to go from one FPGA to another, where both FPGAs are attached to the same TOR to be 2.88 μs. This delay is comparable to the average latencies in our Catapult v1 system [4], where nearest neighbor (1-hop) communication had a round-trip latency of approximately 1 μs. However, worst-case round-trip

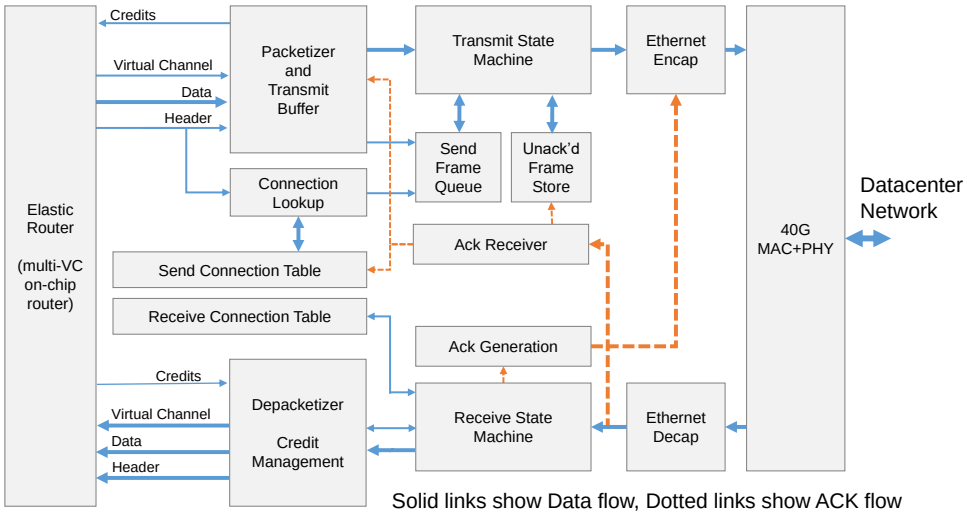

Fig. 9. Block diagram of the LTL Protocol Engine.

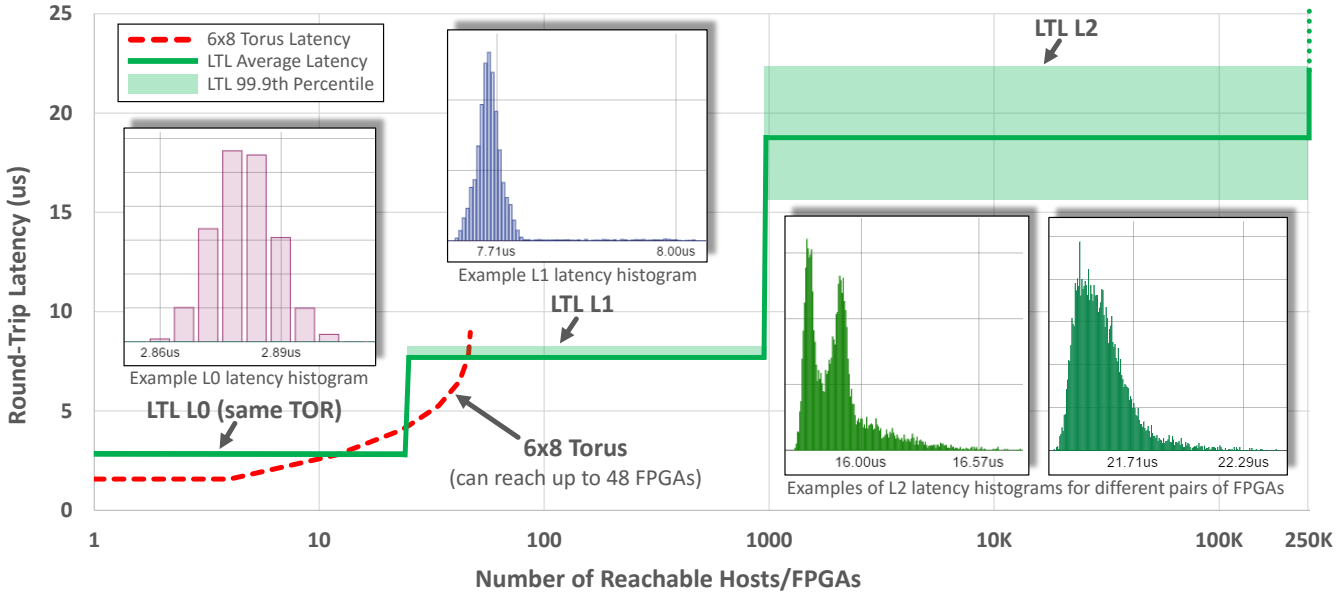

Fig. 10. Round-trip latency of accesses to remote machines with LTL compared to the 6x8 torus from [4]. Shaded areas represent range of latencies up to the 99.9th percentile. LTL enables round-trip access to 100,000+ machines in under 23.5 μs.

communication in the torus requires 7 μsec, slower than LTL. These latencies are also comparable to RDMA read latencies that we measured on the same types of systems. A complete evaluation follows in section V-C.

B. Elastic Router

The Elastic Router is an on-chip, input-buffered crossbar switch designed to support efficient communication between multiple endpoints on an FPGA across multiple virtual channels (VCs). This section describes the architectural and microarchitectural specifications of ER, its design rationales, and how it integrates with other components.

The Elastic Router was developed to support intra-FPGA communication between Roles on the same FPGA and inter-FPGA communication between Roles running on other FPGAs through the Lightweight Transport Layer. In an example single-role deployment, the ER is instantiated with 4 ports: (1) PCIe DMA, (2) Role, (3) DRAM, and (4) Remote (to LTL). The design can be fully parameterized in the number of ports, virtual channels, flit and phit sizes, and buffer capacities.

Any endpoint can send a message through the ER to any other port including itself as U-turns are supported. In addition, multiple ERs can be composed to form a larger on-chip network topology, e.g., a ring or a 2-D mesh.

The ER supports multiple virtual channels, virtualizing the physical links between input and output ports. The ER employs credit-based flow control, one credit per flit, and is an input-buffered switch. Unlike a conventional router that allocates a static number of flits per VC, the ER supports an elastic policy

that allows a pool of credits to be shared among multiple VCs, which is effective in reducing the aggregate flit buffering requirements.

The LTL and ER blocks are crucial for allowing FPGAs to (1) be organized into multi-FPGA services, and (2) to be remotely managed for use as a remote accelerator when not in use by their host. The area consumed is 7% for LTL and 2% for ER (Figure 5.) While not insubstantial, services using only their single local FPGA can choose to deploy a shell version without the LTL block. Services needing a multi-FPGA accelerator or services not using their local FPGA (to make it available for global use) should deploy shell versions with the LTL block.

With this communication protocol, it is now possible to manage the datacenter's accelerator resources as a global pool. The next sections provide an overview of LTL performance, describe an example service running remotely, and give a brief overview of how hardware services are managed.

C. LTL Communication Evaluation

Our datacenter network is organized into three tiers. At the bottom tier (L0), each top-of-rack (TOR) switch connects directly to 24 hosts. The next tier of switches (L1) form pods of 960 machines. The final layer (L2) connects multiple pods together that can connect more than a quarter million of machines. Each layer of the hierarchy introduces more over-subscription such that the node-to-node bandwidth is greatest between nodes that share a L0 switch and least between pairs connected via L2.

Figure 10 shows LTL round-trip latency results for FPGAs connected through the different datacenter network tiers described above, namely L0, L1, and L2. For each tier, lines represent average latency while the shaded areas capture the range of latencies observed up to the 99.9th percentile. Note that the x-axis is logarithmic. Results were obtained through cycle-level measurements across multiple sender-receiver pairs and capture idle LTL round-trip latency from the moment the header of a packet is generated in LTL until the corresponding ACK for that packet is received in LTL. For each tier we also include sample latency distributions from individual runs. As a point of comparison, we also show results from our previously published 6x8 torus network [4] that is, however, limited to 48 FPGAs. Note that even though we generated LTL traffic at a very low rate to obtain representative idle latencies, L1 and L2 results are inevitably affected by other datacenter traffic that is potentially flowing through the same switches.

Average round-trip latency for FPGAs on the same TOR (L0) is 2.88 μs. L0 distributions were consistently very tight across all runs with the 99.9th percentile at 2.9 μs. L1 average latency is 7.72 μs with a 99.9th percentile latency of 8.24 μs indicating a tight distribution for the majority of L1 traffic. However, in this case there is also a small tail of outliers—possibly packets that got stuck behind other traffic going through the same L1 switch—that encounter up to roughly half a microsecond of additional latency. L2 average latency is 18.71 μs with a 99.9th percentile of 22.38 μs. Even though

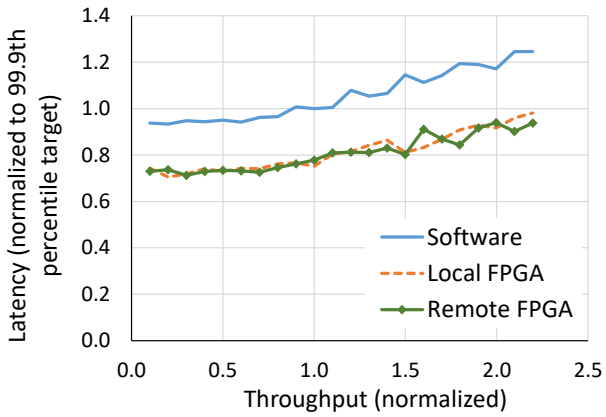

Fig. 11. Latencies of software ranking, locally accelerated ranking, and remotely accelerated ranking. All data are normalized to software 99.9th percentile latency target.

L2 latencies and distributions can vary significantly, as is highlighted by the two example L2 histograms, it is worth noting that L2 latency never exceeded 23.5 μs in any of our experiments. The higher L2 latency variability and noise does not come as a surprise as L2 switches can connect up to hundreds of thousands of hosts. In addition to oversubscription effects, which can become much more pronounced at this level, L2 latencies can be affected by several other factors that range from physical distance and cabling to transient background traffic from other workloads and even L2 switch internal implementation details, such as multi-pathing and ASIC organization.

Compared to LTL, our previous 6x8 torus, which employs a separate physical inter-FPGA network, offers comparable round-trip latencies at low FPGA counts. However, communication is strictly limited to groups of 48 FPGAs and the separate dedicated inter-FPGA network can be expensive and complex to cable and maintain. Similarly, failure handling in the torus can be quite challenging and impact latency as packets need to be dynamically rerouted around a faulty FPGA at the cost of extra network hops and latency. LTL on the other hand shares the existing datacenter networking infrastructure allowing access to hundreds of thousands of hosts/FPGAs in a fixed number of hops. Failure handling also becomes much simpler in this case as there is an abundance of spare accessible nodes/FPGAs.

D. Remote Acceleration Evaluation

To study the end-to-end impact of accelerating production-level applications using remote FPGAs, we evaluate the search ranking accelerator from Section III running remotely over the network via LTL. Figure 11 shows the throughput of a single accelerator when accessed remotely compared to the software and locally attached accelerator. The data are all normalized to the 99.9th percentile latency target of ranking running in software mode. The data show that over a range of throughput targets, the latency overhead of remote accesses is minimal.

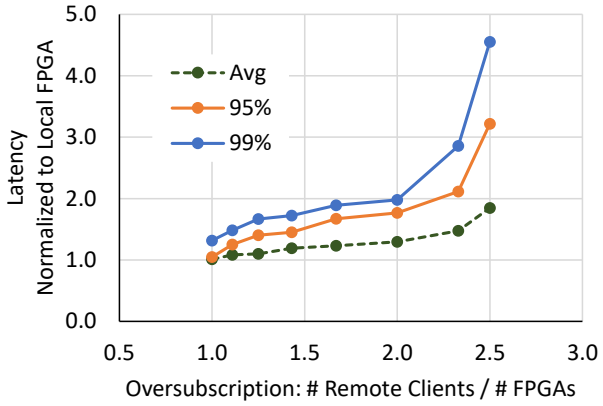

Fig. 12. Average, 95th, and 99th percentile latencies to a remote DNN accelerator (normalized to locally-attached performance in each latency category).

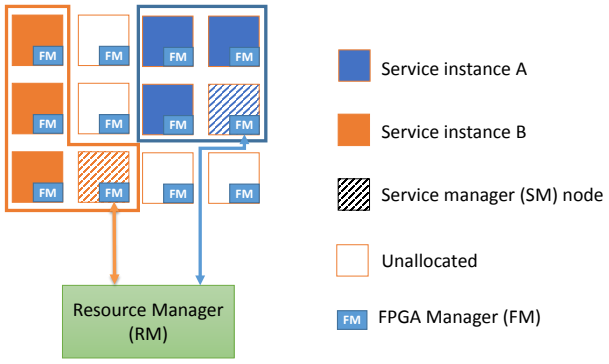

Fig. 13. Two Hardware-as-a-Service (HaaS) enabled hardware accelerators are shown running under HaaS. FPGAs are allocated to each service from Resource Manager's resource pool. Each service has a Service Manager node to administer the service on the allocated resources. Each FPGA has a lightweight FPGA Manager for managing calls from the Service Manager.

The impact on the host server while serving remote requests is minimal, because the FPGA directly handles the network and processing load. The host sees no increase in CPU or memory utilization, and only a small increase in overall power draw. Thus, FPGA-enabled servers can safely donate their FPGA to the global pool of hardware resources with minimal impact on software performance. One concern when sharing the FPGA with remote services is that network bandwidth can be reduced by the remote service. To prevent issues, LTL implements bandwidth limiting to prevent the FPGA from exceeding a configurable bandwidth limit.

E. Oversubscription Evaluation

One key motivation behind enabling remote hardware services is that the resource requirements for the hardware fabric rarely map 1:1 with the resource needs for the software fabric. Some services will need more FPGAs than the number of servers. Other services will have unused FPGA resources which can be made available to other services. Because the accelerators communicate directly rather than through the CPU, a service borrowing an FPGA can do so with minimal impact to the performance of the host server.

To evaluate the impact of remote service oversubscription, we deployed a small pool of latency-sensitive Deep Neural Network (DNN) accelerators shared by multiple software clients in a production datacenter. To stress the system, each software client sends synthetic traffic to the DNN pool at a rate several times higher than the expected throughput per client in deployment. We increased the ratio of software clients to accelerators (by removing FPGAs from the pool) to measure the impact on latency due to oversubscription.

Figure 12 shows the average, 95th and 99th percentile request latencies as the ratio of clients to FPGAs (oversubscription) increases. These figures plot end-to-end request latencies, measuring the time between when a request is enqueued to the work queue and when its response is received from the accelerator. To expose the latencies more clearly, these results do not include end-to-end service and software latencies as the ranking data in Figure 11 do. In the no oversubscription (1 to 1) case, remotely accessing the service adds 1% additional latency to each request on average, 4.7% additional latency at the 95th percentile, and 32% at the 99th percentile. As expected, contention and queuing delay increases as oversubscription increases. Eventually, the FPGA reaches its peak throughput and saturates, causing latencies to spike due to rapidly increasing queue depths. In this particular case study, each individual FPGA has sufficient throughput to sustain 2-2.5 software clients (operating at very high rates several times the expected throughput in production) before latencies begin to spike prohibitively. This shows that conservatively half to two-thirds of the FPGAs can be freed up for other functions.

F. Hardware-as-a-Service Model

While a complete overview of the management of our hardware fabric is beyond the scope of this paper, we provide a short overview of the Hardware-as-a-Service (HaaS) platform here. HaaS manages FPGAs in a manner similar to Yarn [10] and other job schedulers. Figure 13 shows a few services running under the HaaS model. A logically centralized Resource Manager (RM) tracks FPGA resources throughout the datacenter. The RM provides simple APIs for higher-level Service Managers (SM) to easily manage FPGA-based hardware Components through a lease-based model. Each Component is an instance of a hardware service made up of one or more FPGAs and a set of constraints (locality, bandwidth, etc.). SMs manage service-level tasks such as load balancing, inter-component connectivity, and failure handling by requesting and releasing Component leases through RM. A SM provides pointers to the hardware service to one or more end users to take advantage of the hardware acceleration. An FPGA Manager (FM) runs on each node to provide configuration and status monitoring for the system.

VI. Related Work

There are many possible options for incorporating accelerators into large-scale systems, including the type of accelerator, how it interfaces with the CPUs, and how the accelerators can

communicate with one another. Below we describe a taxonomy that uses those three categories.

1) CPU-Accelerator memory integration. The closer the integration with the CPU, the finer-grain the problem that can be beneficially offloaded to the accelerator. Possibilities include:
   - (C) - Coherent accelerators, where data movement is handled by a memory coherence protocol.
   - (I) - I/O level, where data movement is done via DMA transfers, and
   - (N) - Network level, where data movement is done via Ethernet (or other) packets.
2) Accelerator connectivity scale. The scale at which accelerators can directly communicate without CPU intervention. Possibilities include:
   - (S) Single server / single appliance (i.e. CPU management is required to communicate with accelerators on other servers)
   - (R) Rack level
   - (D) Datacenter scale
3) Accelerator type. Possibilities include:
   - (F) FPGAs
   - (G) GPUs
   - (A) ASICs

In this section we describe a subset of previously proposed datacenter or server accelerators, organized by this taxonomy, with an emphasis on FPGA-based accelerators. While most designs fit into one category, our proposed design could fit into two. Our accelerator could fit as an ISF design when doing local acceleration since there are no additional FPGAs (and hence no inter-FPGA connectivity) within the same server. However, the design is an NDF design when doing network acceleration since connected FPGAs (including the local one) can all be accessed via Ethernet packets. LTL ties together local acceleration and network acceleration scenarios, so altogether we view it as an NDF architecture. All of the systems that we survey below fall into other categories.

NSF: Network/Single/FPGA: The announced Mellanox hybrid NIC [5] enables one to build a bump-in-the-wire architecture where the FPGA is in-line with the NIC, although there has not been a published large-scale deployment of the Mellanox hardware. Other FPGA boards, such as NetFPGA [11], high-speed trading appliances, and network appliances are specifically designed to augment or replace standard network interface cards for specific applications. While these can each be considered bump-in-the wire architectures, there is little to no communication between or aggregation of accelerators. As such, they are limited to problem sizes that fit on within a single accelerator or appliance.

In addition to commercial network acceleration designs, there has been considerable research into FPGA-based bump-in-the-wire architectures. For example, FPGA-accelerated Memcached designs lend themselves naturally to directly attaching FPGA to the network [12], [13], [14], [15] and programming the FPGA to handle requests some directly from the network without software interference.

IRF: IO/Rack/FPGA: Catapult v1 is an IRF system, where FPGAs are connected by a dedicated network at rack scale [4] accessible through PCIe.

The scale of that communication is limited to a single rack, which similarly limits the scope of acceleration services that can be supported. In addition, these 2D torus network topologies suffer from resiliency challenges since the failure of one node affects neighboring nodes.

The newer version of Novo-G, called Novo-G# [16] also falls into this category with a three-dimensional 4x4x4 torus. Another such system is the Cray XD-1 [17], which places up to six FPGAs in a single chassis. These FPGAs attach directly to the dedicated rack-scale RapidArray network, which is distinct from the Ethernet network that connects multiple racks of up to 12 chassis. Another example is Maxwell [18], which also provides rack-scale direct FPGA-to-FPGA communication.

ISF: IO/Single/FPGA: The Baidu SDA [3], Novo-G [19], Maxeler MPC [20], Convey HC-2 [21], BeeCube BEE4 [22], and SRC MAPStation [23] are all large or multi-FPGA appliances, many with high connectivity between FPGAs within the appliance, but the CPU manages communication beyond a single box.

ISFG: IO/Zero/FPGA+GPU: The QP [24] system is designed to handle larger HPC-style problems with a large number of FPGAs and GPUs. However, all communication between accelerators has to be managed by the CPU, which increases the latency of inter-FPGA communication and limits the scale of problems that can be profitably accelerated by multiple FPGAs.

CSF: Coherent/Single/FPGA: Pulling the FPGA into the coherence domain of the CPU improves the granularity of communication between the accelerator and CPU, and can increase the scope of applications that can be profitably offloaded to the accelerator. One of the most prominent examples is IBM, who is shipping Power8 [25] systems with a coherent accelerator interface called CAPI [26]. Researchers have demonstrated several applications on CAPI such as bioinformatics [27] and large matrix processing [28].

Intel's hybrid CPU/FPGA [29] is another example. FPGA boards that included Intel FSB [30], QPI [31], or coherent HyperTransport [32], [33] are also included. While coherence helps within a limited scope, it does not scale across servers, and hence does not significantly differ on a datacenter scale from I/O integrated accelerators, as the CPU manages all FPGA-to-FPGA communication.

ISG: IO/Single/GPU: GPUs are to-date the most successful architecture for accelerating CPU-based systems. Multiple GPUs are commonly used for large problems, and the addition of technologies like NVLink [34] have enabled multiple GPUs to talk directly. However, the scale of NVLink is still limited to a single box, and there is no integration of GPUs with the network, so the scale of GPU acceleration achievable without CPU intervention is still relatively small. Some compelling recent work looked at using GPUs to offer DNNs as a service in the datacenter [35], with a subset of the datacenter's servers containing GPUs, and in a disaggregated design, servers with low-end CPUs feeding many GPUs in a single box. In all configurations, the GPU was accessed by the

CPU over PCIe, and all inter-GPU communication occurred within a single server.

ISA: IO/Single/ASIC: ASICs, whether in the form of stand-alone accelerators or as custom blocks integrated into conventional CPUs, are another approach that has worked well in client systems and for some server infrastructure functions. However, the general-purpose nature and rapid pace of change of applications makes single-function ASICs challenging to deploy at scale. Counterexamples may be crypto or compression blocks; however, the economics at scale of doing crypto and compression in soft vs. hard logic are not yet clear. In the long term, we expect that some acceleration functions–particularly for system offload functions–will first be implemented on FPGAs, and then after long-term stability of that function has been demonstrated, the function could be moved into hardened logic. Machine learning is one application that is sufficiently important to justify as a domain-specific ASIC at scale. One such example is the DianNao family of deep learning accelerators [36].

Other taxonomies: Fahmy and Vipin create a service taxonomy based on how FPGAs are exposed to external customers from a datacenter environment [37]. The remote acceleration model that we propose in this paper focuses not on whom the accelerators are exposed to, but the architecture, deployment, and performance of services on a global pool of accelerators. The remote acceleration model can support all three models proposed by Fahmy and Vipin, including Vendor Acceleration, Accelerators as a Service, and Fabric as a Service.

Finally, while other promising programmable accelerator architectures exist, such as MPPAs and CGRAs, none have reached the level of commercial viability and availability to be ready for production datacenters.

VII. Conclusions

The slowing of Moore's Law, coupled with the massive and growing scale of datacenter infrastructure, makes specialized accelerators extremely important. The most important problem to solve is in the design of *scalable accelerators*, which are economically viable across a large infrastructure, as opposed to accelerators that enhance a specialized small or medium-scale deployment of machines. What has made research in this space challenging is the large number of possible design options, spanning the type of accelerator (FPGAs, GPUs, ASICs), their location in the infrastructure (coherent, PCIe space, or network path), and their scale (how many can communicate directly with one another).

This paper described Configurable Clouds, a datacenter-scale acceleration architecture, based on FPGAs, that is both scalable and flexible. By putting in FPGA cards both in I/O space as well as between a server's NIC and the local switch, the FPGA can serve as both a network accelerator and local compute accelerator. By enabling the FPGA to talk directly to the network switch, each FPGA can communicate directly with every other FPGA in the datacenter, over the network, without any CPU software. This flexibility enables ganging together groups of FPGAs into service pools, a concept we call Hardware as a Service (HaaS). We demonstrate a reliable communication protocol for inter-FPGA communication that achieves comparable latency to prior state of the art, while scaling to hundreds of thousands of nodes.

The architecture was demonstrated successfully across multiple datacenter scenarios: use as a local offload engine (for accelerating Bing web search), a local network acceleration engine (for network crypto), and as a remote acceleration service for web search. With the Configurable Clouds design, reconfigurable logic becomes a first-class resource in the datacenter, and over time may even be running more computational work than the datacenter's CPUs. There will still, of course, be a role for GPUs and ASICs, which can augment a subset of servers with capabilities to accelerate specific workloads. This reconfigurable acceleration plane, however, will be pervasive, offering large-scale gains in capabilities and enabling rapid evolution of datacenter protocols. In addition to being a compute accelerator, we anticipate that it will drive evolution of the datacenter architecture in the near future, including network protocols, storage stacks, and physical organization of components.

The Catapult v2 architecture has already been deployed at hyperscale and is how most new Microsoft data center servers are configured. The FPGAs accelerate both compute workloads, such as Bing web search ranking, and Azure infrastructure workloads, such as software-defined networks and network crypto.

Acknowledgements

The authors would like to acknowledge the many individuals who contributed to this project: Mike Andrewartha, Jack Lavier, Mark Shaw, Kushagra Vaid, and the rest of the CSI team; Shlomi Alkalay, Kyle Holohan, Raja Seera, Phillip Xiao, and the rest of the Bing IndexServe team; Daniel Firestone, Albert Greenberg, Dave Maltz, and the rest of the Azure CloudNet team; Stuart Byma, Alvin Lebeck, and Jason Thong.

References

[1] M. Staveley, "Applications that scale using GPU Compute," in *AzureCon 2015*, August 2015.

[2] J. Barr, "Build 3D Streaming Applications with EC2's New G2 Instance Type," Nov 2013.

[3] J. Ouyang, S. Lin, W. Qi, Y. Wang, B. Yu, and S. Jiang, "SDA: Software-Defined Accelerator for Large-Scale DNN Systems," in *HotChips 2014*, August 2014.

[4] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. R. Larus, E. Peterson, G. Prashanth, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," in *International Symposium on Computer Architecture (ISCA)*, 2014.

[5] Mellanox, "ConnectX-4 Lx EN Programmable Adapter Card. Rev. 1.1," 2015.

[6] S. Gulley and V. Gopal, "Haswell Cryptographic Performance," July 2013. Available at http://www.intel.com/content/www/us/en/communications/haswell-cryptographic-performance-paper.html.

[7] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An fpga memcached appliance," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, (New York, NY, USA), pp. 245–254, ACM, 2013.

[8] IEEE, *IEEE 802.1Qbb - Priority-based Flow Control*, June 2011 ed., 2011.

[9] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale rdma deployments," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, (New York, NY, USA), pp. 523–536, ACM, 2015.

[10] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, (New York, NY, USA), pp. 5:1–5:16, ACM, 2013.

[11] G. Gibb, J. Lockwood, J. Naous, P. Hartke, and N. McKeown, "NetFPGA–An Open Platform for Teaching How to Build Gigabit-Rate Network Switches and Routers," in *IEEE Transactions on Education*, 2008.

[12] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached," *SIGARCH Comput. Archit. News*, vol. 41, pp. 36–47, June 2013.

[13] M. lavasani, H. Angepat, and D. Chiou, "An fpga-based in-line accelerator for memcached," *Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2013.

[14] M. Blott and K. Vissers, "Dataflow architectures for 10gbps line-rate key-value-stores," in *HotChips 2013*, August 2013.

[15] E. S. Fukuda, H. Inoue, T. Takenaka, D. Kim, T. Sadashisa, T. Asai, and M. Motomura, "Caching mecached at reconfigurable network interface," in *Proceedings of the 24th International Conference on Field Programmable Logic and Applications*, 2014.

[16] A. G. Lawande, A. D. George, and H. Lam, "Novo-g: a multidimensional torus-based reconfigurable cluster for molecular dynamics," *Concurrency and Computation: Practice and Experience*, pp. n/a–n/a, 2015. cpe.3565.

[17] Cray, *Cray XD1 Datasheet*, 1.3 ed., 2005.

[18] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. Mccormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest, "Maxwell - a 64 FPGA Supercomputer," *Engineering Letters*, vol. 16, pp. 426–433, 2008.

[19] A. George, H. Lam, and G. Stitt, "Novo-g: At the forefront of scalable reconfigurable supercomputing," *Computing in Science Engineering*, vol. 13, no. 1, pp. 82–86, 2011.

[20] O. Pell and O. Mencer, "Surviving the end of frequency scaling with reconfigurable dataflow computing," *SIGARCH Comput. Archit. News*, vol. 39, pp. 60–65, Dec. 2011.

[21] Convey, *The Convey HC-2 Computer*, conv-12-030.2 ed., 2012.

[22] BEECube, *BEE4 Hardware Platform*, 1.0 ed., 2011.

[23] SRC, *MAPstation Systems*, 70000 AH ed., 2014.

[24] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, and W. Hwu, "Qp: A heterogeneous multi-accelerator cluster," 2009.

[25] J. Stuecheli, "Next Generation POWER microprocessor," in *HotChips 2013*, August 2013.

[26] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, "Capi: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7–1, 2015.

[27] M. J. Jaspers, *Acceleration of read alignment with coherent attached FPGA coprocessors*. PhD thesis, TU Delft, Delft University of Technology, 2015.

[28] C.-C. Chung, C.-K. Liu, and D.-H. Lee, "Fpga-based accelerator platform for big data matrix processing," in *Electron Devices and Solid-State Circuits (EDSSC), 2015 IEEE International Conference on*, pp. 221–224, IEEE, 2015.

[29] P. Gupta, "Xeon+fpga platform for the data center," 2015.

[30] L. Ling, N. Oliver, C. Bhushan, W. Qigang, A. Chen, S. Wenbo, Y. Zhihong, A. Sheiman, I. McCallum, J. Grecco, H. Mitchel, L. Dong, and P. Gupta, "High-performance, Energy-efficient Platforms Using In-Socket FPGA Accelerators," in *FPGA'09: Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, (New York, NY, USA), pp. 261–264, ACM, 2009.

[31] Intel, "An Introduction to the Intel Quickpath Interconnect," 2009.

[32] D. Slogsnat, A. Giese, M. Nüssle, and U. Brüning, "An open-source hypertransport core," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, pp. 14:1–14:21, Sept. 2008.

[33] DRC, *DRC Accelium Coprocessors Datasheet*, ds ac 7-08 ed., 2014.

[34] "NVIDIA NVLink High-Speed Interconnect: Application Performance," Nov. 2014.

[35] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 27–40, ACM, 2015.

[36] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM SIGPLAN Notices*, vol. 49, pp. 269–284, ACM, 2014.

[37] S. A. Fahmy and K. Vipin, "A case for fpga accelerators in the cloud," Poster at SoCC 2014.

2015 International Conference on Parallel Architecture and Compilation

Practical Near-Data Processing for In-memory Analytics Frameworks

Mingyu Gao, Grant Ayers, Christos Kozyrakis
Stanford University
{*mgao12,geayers,kozyraki*}*@stanford.edu*

***Abstract*—The end of Dennard scaling has made all systems energy-constrained. For data-intensive applications with limited temporal locality, the major energy bottleneck is data movement between processor chips and main memory modules. For such workloads, the best way to optimize energy is to place processing near the data in main memory. Advances in 3D integration provide an opportunity to implement near-data processing (NDP) without the technology problems that similar efforts had in the past.**

This paper develops the hardware and software of an NDP architecture for in-memory analytics frameworks, including MapReduce, graph processing, and deep neural networks. We develop simple but scalable hardware support for coherence, communication, and synchronization, and a runtime system that is sufficient to support analytics frameworks with complex data patterns while hiding all the details of the NDP hardware. Our NDP architecture provides up to 16x performance and energy advantage over conventional approaches, and 2.5x over recently-proposed NDP systems. We also investigate the balance between processing and memory throughput, as well as the scalability and physical and logical organization of the memory system. Finally, we show that it is critical to optimize software frameworks for spatial locality as it leads to 2.9x efficiency improvements for NDP.

***Keywords*-Near-data processing; Processing in memory; Energy efficiency; In-memory analytics;**

I. Introduction

The end of Dennard scaling has made all systems energy-limited [1], [2]. To continue scaling performance at exponential rates, we must minimize energy overhead for every operation [3]. The era of "big data" is introducing new workloads which operate on massive datasets with limited temporal locality [4]. For such workloads, cache hierarchies do not work well and most accesses are served by main memory. Thus, it is particularly important to improve the memory system since the energy overhead of moving data across board-level and chip-level interconnects dwarfs the cost of instruction processing [2].

The best way to reduce the energy overhead of data movement is to avoid it altogether. There have been several efforts to integrate processing with main memory [5]–[10]. A major reason for their limited success has been the cost and performance overheads of integrating processing and DRAM on the same chip. However, advances in 3D integration technology allow us to place computation near memory through TSV-based stacking of logic and memory chips [11], [12]. As a result, there is again significant interest in integrating processing and memory [13].

With a practical implementation technology available, we now need to address the system challenges of near-data processing (NDP). The biggest issues are in the hardware/software interface. NDP architectures are by nature highly-distributed systems that deviate from the cache-coherent, shared-memory models of conventional systems. Without careful co-design of hardware and software runtime features, it can be difficult to efficiently execute analytics applications with non-trivial communication and synchronization patterns. We must also ensure that NDP systems are energy-optimized, balanced in terms of processing and memory capabilities, and able to scale with technology.

The goal of this paper is twofold. First, we want to design an efficient and practical-to-use NDP architecture for popular analytics frameworks including MapReduce, graph processing, and deep neural networks. In addition to using simple cores in the logic layers of 3D memory stacks in a multi-channel memory system, we add a few simple but key hardware features to support coherence, communication, and synchronization between thousands of NDP threads. On top of these features, we develop an NDP runtime that provides services such as task launch, thread communication, and data partitioning, but hides the NDP hardware details. The NDP runtime greatly simplifies porting analytics frameworks to this architecture. The end-user application code is unmodified: It is the same as if these analytics frameworks were running on a conventional system.

Second, we want to explore balance and scalability for NDP systems. Specifically, we want to quantify trade-offs on the following issues: what is the right balance of compute-to-memory throughput for NDP systems; what is the efficient communication model for the NDP threads; how do NDP systems scale; what software optimizations matter most for efficiency; what are the performance implications for the host processors in the system.

Our study produces the following insights: First, simple hardware support for coherence and synchronization and a runtime that hides their implementation from higher-level software make NDP systems efficient and practical to use with popular analytics frameworks. Specifically, using a pull-based communication model for NDP threads that utilizes the hardware support for communication provides a 2.5x efficiency improvement over previous NDP systems. Second, NDP systems can provide up to 16x overall advantage for both performance and energy efficiency over

1089-795X/15 $31.00 © 2015 IEEE
DOI 10.1109/PACT.2015.22

conventional systems. The performance of the NDP system scales well to multiple memory stacks and hundreds of memory-side cores. Third, a few (4-8) in-order, multi-threaded cores with simple caches per vertical memory channel provide a balanced system in terms of compute and memory throughput. While specialized engines can produce some additional energy savings, most of the improvement is due to the elimination of data movement. Fourth, to achieve maximum efficiency in an NDP system, it is important to optimize software frameworks for spatial locality. For instance, an edge-centric version of the graph framework improves performance and energy by more than 2.9x over the typical vertex-centric approach. Finally, we also identify additional hardware and software issues and opportunities for further research on this topic.

II. Background and Motivation

Processing-in-Memory (PIM): Several efforts in the 1990s and early 2000s examined single-chip logic and DRAM integration. EXECUBE, the first PIM device, integrated 8 16-bit SIMD/MIMD cores and 4 Mbits of DRAM [5], [6]. IRAM combined a vector processor with 13 Mbytes of DRAM for multimedia workloads [7]. DIVA [8], Active Pages [9], and FlexRAM [10] were drop-in PIM devices that augmented a host processor, but also served as traditional DRAM. DIVA and FlexRAM used programmable cores, while Active Pages used reconfigurable logic. Several custom architectures brought computation closer to data on memory controllers. The Impulse project added application-specific scatter and gather logic which coalesced irregularly-placed data into contiguous cachelines [14], and Active Memory Operations moved selected operations to the memory controller [15].

3D integration: Vertical integration with through-silicon vias (TSV) allows multiple active silicon devices to be stacked with dense interconnections [16], [17]. 3D stacking promises significant improvements in power and performance over traditional 2D planar devices. Despite thermal and yield challenges, recent advances have made this technology commercially viable [11], [12]. Two of the most prominent 3D-stacked memory technologies today are Micron's Hybrid Memory Cube (HMC) [18] and JEDEC's High Bandwidth Memory (HBM) specification [19], both of which consist of a logic die stacked with several DRAM devices. Several studies have explored the use of 3D-stacked memory for caching [20]–[24]. We focus on applications with no significant temporal locality and thus will not benefit from larger caches.

From PIM to NDP: 3D-stacking with TSVs addresses one of the primary reasons for the limited success on past PIM projects: the additional cost as well as the performance or density shortcomings of planar chips that combined processing and DRAM. Hence, there is now renewed interest in systems that use 3D integration for near-data processing [13]. Pugsley et al. evaluated a daisy chain of modified HMC devices with simple cores in the logic layers for MapReduce workloads [25]. NDA stacked Coarse-Grained Reconfigurable Arrays on commodity DRAM modules [26]. Both designs showed significant performance and energy improvements, but they also relied on host processor to coordinate data layout and necessary communication between NDP threads. Tesseract was a near-data accelerator for large-scale graph processing that provided efficient communication using message passing between memory partitions [27]. The Active Memory Cube focused on scientific workloads and used specialized vectorized processing elements with no caches [28]. PicoServer [29] and 3D-stacked server [30] focused on individual stacks for server integration, and targeted web applications and key-value store which are not as memory-intensive. Other work has studied 3D integration with GPGPUs [31], non-volatile memory [32], and other configurations [33]–[36].

However, implementation technology is not the only challenge. PIM and NDP systems are highly parallel but most do not support coherent, shared memory. Programming often requires specialized models or complex, low-level approaches. Interactions with features such as virtual memory, host processor caches, and system-wide synchronization have also been challenging. Our work attempts to address these issues and design efficient yet practical NDP systems.

The biggest opportunity for NDP systems is with emerging "big data" applications. These data-intensive workloads scan through massive datasets in order to extract compact knowledge. The lack of temporal locality and abundant parallelism suggests that NDP should provide significant improvements over conventional systems that waste energy on power-hungry processor-to-memory links. Moreover, analytics applications are typically developed using domain-specific languages for domains such as MapReduce, graphs, or deep neural networks. A software framework manages the low-level communication and synchronization needed to support the domain abstractions at high performance. Such frameworks are already quite popular and perform very well in cluster (scale-out) environments, where there is no cluster-scale cache coherence. By optimizing NDP hardware and low-level software for such analytics frameworks, we can achieve significant gains without exposing end programmers to any details of the NDP system.

III. NDP Hardware Architecture

NDP systems can be implemented with several technology options, including processing on buffer-on-board (BoB) devices [37], edge-bonding small processor dies on DRAM chips, and 3D-stacking with TSVs [11], [12]. We use 3D-stacking with TSVs because of its large bandwidth and energy advantages. Nevertheless, most insights we draw on NDP systems, such as the hardware/software interface

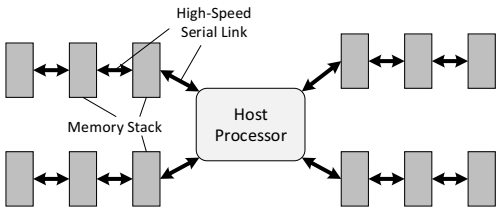

Figure 1. The Near Data Processing (NDP) architecture.

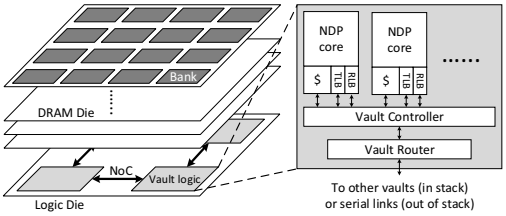

Figure 2. 3D memory stack and NDP components.

and the runtime optimizations for in-memory analytics, are applicable to NDP systems that use alternative options.

Figure 1 provides an overview of the NDP architecture we study. We start with a system based on a high-end host processor chip with out-of-order (OoO) cores, connected to multiple memory stacks. This is similar to a conventional system where the host processor uses multiple DDR3 memory channels to connect to multiple memory modules, but high-speed serial links are used instead of DDR interface. The memory stacks integrate NDP cores and memory using 3D stacking, as shown in Figure 2. The portions of applications with limited temporal locality execute on the NDP cores in order to minimize the energy overhead of data movement. The portions of applications with sufficient temporal locality execute on the host processor as usual. The NDP cores and the host processor cores see the same physical address space (shared memory).

Recently-proposed NDP hardware architectures use a similar base design. To achieve significant performance and energy improvements for complex analytics workloads, we need to further and carefully design the communication and memory models of the NDP system. Hence, after a brief overview of the base design (section III-A), we introduce the new hardware features (section III-B) that enable the software runtime discussed in section IV.

A. Base NDP Hardware

The most prominent 3D-stacked memory technologies today are Micron's Hybrid Memory Cube (HMC) [18] and JEDEC's High Bandwidth Memory (HBM) [19], [38]. Although the physical structures differ, both HMC and HBM integrate a logic die with multiple DRAM chips in a single stack, which is divided into multiple independent channels (typically 8 to 16). HBM exposes each channel as raw DDR-like interface; HMC implements DRAM controller in the logic layer as well as SerDes links for off-stack communication. We use the term *vault* to describe the vertical channel in HMC, including the memory banks and its separate DRAM controller. 3D-stacked memory provides high bandwidth through low-power TSV-based channels, while latency is close to normal DDR3 chips due to their similar DRAM core structures [38]. In this study, we use an HMC-like 4 Gbyte stack[1] with 8 DRAM dies by default, and investigate different vault organization in section VI.

We use general-purpose, programmable cores for near-data processing to enable a large and diverse set of analytics frameworks and applications. While vector processors [7], reconfigurable logic [9], [26], or specialized engines [39], [40] may allow additional improvements, our results show that most of the energy benefits are due to the elimination of data movement (see section VI). Moreover, since most of the NDP challenges are related to software, starting with programmable cores is an advantage.

Specifically, we use simple, in-order cores similar to the ARM Cortex-A7 [41]. Wide-issue or OoO cores are not necessary due to the limited locality and instruction-level parallelism in code that executes near memory, nor are they practical given the stringent power and area constraints. However, as in-memory analytics relies heavily on floating-point operations, we include one FPU per core. Each NDP core also has private L1 caches for instructions and data, 32 Kbytes each. The latter is used primarily for temporary results. There is not sufficient locality in the workloads to justify the area and power overheads of private or even shared L2 caches.

For several workloads—particularly for graph processing—the simple cores are underutilized as they are often stalled waiting for data. We use fine-grained multithreading (cycle-by-cycle) as a cost-effective way to increase the utilization of simple cores given the large amount of memory bandwidth available within each stack [42]. Only a few threads (2 to 4) are needed to match the short latency to nearby memory. The number of threads per core is also limited by the L1 cache size.

B. NDP Communication & Memory Model

The base NDP system is similar to prior NDP designs that target simple workloads, such as the embarrassingly-parallel map phase in MapReduce [25]. Many real-world applications, such as graph processing and deep learning, require more complex communication between hundreds to thousands of threads [43], [44]. Relying on the host processor to manage all the communication will not only turn it into the performance bottleneck, but will also waste

[1]Higher capacity stacks will become available as higher capacity DRAM chips are used in the future.

energy moving data between the host processor and memory stacks (see section VI). Moreover, the number of NDP cores will grow over time along with memory capacity. To fully utilize the memory and execution parallelism, we need an NDP architecture with support for efficient communication that scales to thousands of threads.

Direct communication for NDP cores: Unlike previous work [25], [26], we support direct communication between NDP cores within and across stacks because it greatly simplifies the implementation of communication patterns for in-memory analytics workloads (see section IV). The physical interconnect within each stack includes a 2D mesh network-on-chip (NoC) on the logic die that allows the cores associated with each vault to directly communicate with other vaults within the same stack. Sharing a single router per vault is area-effective and sufficient in terms of throughput. The 2D mesh also provides access to the external serial links that connect stacks to each other and to the host processor.

This interconnect allows all cores in the system, NDP and host, to access all memory stacks through a unified physical address space. An NDP core sends read/write accesses directly to its local vault controller. Remote accesses reach other vaults or stacks by routing based on physical addresses (see Figure 2). Data coherence is guaranteed with the help of virtual memory (discussed later). Remote accesses are inherently more expensive in terms of latency and energy. However, analytics workloads operate mostly on local data and communicate at well-understood points. By carefully optimizing the data partitioning and work assignment, NDP cores mostly access memory locations in their own local vaults (see section IV).

Virtual memory: NDP threads access the virtual address space of their process through OS-managed paging. Each NDP core contains a 16-entry TLB to accelerate translation. Similar to an IOMMU in conventional systems, TLB misses from NDP cores are served by the OS on the host processor. The runtime system on the NDP core communicates with the OS on a host core to retrieve the proper translation or to terminate the program in the case of an error. We use large pages (2 Mbyte) for the entire system to minimize TLB misses [45]. Thus, a small number of TLB entries is sufficient to serve large datasets for in-memory analytics, given that most accesses stay within the local vault.

We use virtual memory protection to prevent concurrent (non-coherent) accesses from the host processor and NDP cores to the same page. For example, while NDP threads are working on their datasets, the host processor has no access or read-only access for those pages. We also leverage virtual memory to implement coherence in a coarse-grained per-page manner (see below).

Software-assisted coherence: We use a simple and coarse-grained coherence model that is sufficient to support the communication patterns for in-memory analytics, namely limited sharing with moderate communication at well-specified synchronization points. Individual pages can be cached in only one cache in the system (the host processor's cache hierarchy or an NDP core's cache), which is called the *owner cache*. When a memory request is issued, the TLB identifies the owner cache, which may be local or remote to the issuing core, and the request is forwarded there. If there is a cache miss, the request is sent to the proper memory vault based on the physical address. The data is placed in the cache of the requesting core only if this is the owner cache. This model scales well as it allows each NDP core to hold its own working set in its cache without any communication. Compared to conventional directory-based coherence at cacheline granularity using models like MOESI [46], [47], our model eliminates the storage overhead and the lookup latency of directories.

To achieve high performance with our coherence model, it is critical to carefully assign the owner cache. A naive static assignment which evenly partitions the vault physical address space to each cache will not work well because two threads may have different working set sizes. We provide full flexibility to software by using an additional field (using available bits in PTEs) in each TLB entry to encode and track this page's owner cache. The NDP runtime provides an API to let the analytics framework configure this field when the host thread partitions the dataset or NDP threads allocate their local data (see section IV). In this way, even if an NDP core's dataset spills to non-local vault(s), the data can still be cached.

By treating the cache hierarchy in the host processor as a special owner cache, the same coherence model can also manage interactions between host and NDP cores. Note that the page-level access permission bits (R/W/X) can play an important role as well. For example, host cores may be given read-only access to input data for NDP threads but no access to data that are being actively modified by NDP cores. When an NDP or host core attempts to access a page for which it does not have permission, it is up to the runtime and the OS to handle it properly: signal an error, force it to wait, or transfer permissions.

Remote load buffers: While the coherence model allows NDP cores to cache their own working sets, NDP cores will suffer from latency penalties while accessing remote data during communication phases. As communication between NDP cores often involves streaming read-only accesses (see section IV), we provide a per-core *remote load buffer* (RLB), that allows sequential prefetching and buffering of a few cachelines of *read-only* data. Specifically, 4 to 8 blocks with 64 bytes per block are sufficient to accommodate a few threads. Remote stores will bypass RLBs and go to owner caches directly. RLBs are not kept coherent with remote memories or caches, thus they require explicit flushing through software. This is manageable because flushes are only necessary at synchronization points such as at barriers,

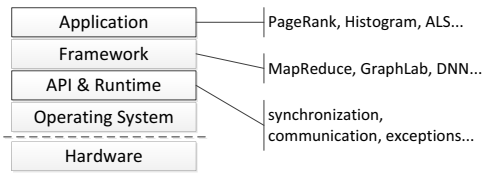

Figure 3. The NDP software stack.

and no writeback is needed. Note that caches do not have to be flushed unless there is a change in the assigned owner cache. Hence, synchronization overhead is low.

Remote atomic operations and synchronization: We support several remote atomic operations, including remote fetch-and-add and remote compare-and-swap, implemented at the vault controllers similarly to [48]. All NDP communication support is the same whether communication occurs within or across stacks. These remote atomic operations can be used to build higher-level synchronization primitives, specifically user-level locks and barriers. We use a hierarchical, tree-style barrier implementation: all threads running inside a vault will first synchronize and only one core signals an update to the rest of the stack. After all of the vaults in the stack have synchronized, one will send a message for cross-stack synchronization.

IV. Practical Software for NDP Systems

The NDP architecture in section III supports flexible inter-thread communication and scalable coherence. We exploit these features in a software infrastructure that supports a variety of popular analytics frameworks. As shown in Figure 3, a lightweight runtime interfaces between the OS and user-level software. It hides the low-level NDP hardware details and provides system services through a simple API. We ported three popular frameworks for in-memory analytics to the NDP runtime: MapReduce, graph processing, and deep neural networks. *The application developer for these frameworks is unaware that NDP hardware is used and would write exactly the same program as if all execution happened in a conventional system.* Due to the similarity between our low-level API and distributed (cluster) environments, we expect that other scale-out processing frameworks would also achieve high performance and energy efficiency on our NDP system.

A. NDP Runtime

The NDP runtime exposes a set of API functions that initialize and execute software on the NDP hardware. It also monitors the execution of NDP threads and provides runtime services such as synchronization and communication. Finally, it coordinates with the OS running on host cores for file I/O and exception handling.

Data partitioning and program launch: The NDP runtime informs applications running on the host cores about the availability of NDP resources, including the number of NDP cores, the memory capacity, and the topology of NDP components. Applications, or more typically frameworks, can use this information to optimize their data partitioning and placement strategy. The runtime provides each NDP thread a private stack and a heap, and works with the OS to allocate memory pages. Applications can optionally specify the owner cache for each page (by default the caller thread's local cache), and preferred stacks or vaults in which to allocate physical memory. This is useful when the host thread partitions the input dataset. The runtime prioritizes allocation on the vault closest to the owner cache as much as possible before spilling to nearby vaults. However, memory allocations do not have to perfectly fit within the local vault since if needed an NDP thread can access remote vaults with slightly worse latency and power. Finally, the runtime starts and terminates NDP threads with an interface similar to POSIX threads, but with hints to specify target cores. This enables threads to launch next to their working sets and ensures that most memory accesses are to local memory.

Communication model: In-memory analytics workloads usually execute iteratively. In every iteration, each thread first processes an independent sub-dataset in parallel for a certain period (*parallel phase*), and then exchanges data with other threads at the end of the current iteration (*communication phase*). MapReduce [49], graph processing [43], [50], and deep neural networks [44] all follow this model. While the parallel phase is easy to implement, the communication phase can be challenging. This corresponds to the shuffle phase in MapReduce, scatter/gather across graph tiles in graph, and forward/backward propagation across network partitions in deep neural networks.

We build on the hardware support for direct communication between NDP threads to design a *pull* model for data exchange. A producer thread will buffer data locally in its own memory vault. Then, it will send a short message containing an address and size to announce the availability of data. The message is sent by writing to a predefined mailbox address for each thread. The consumer will later process the message and use it to pull (remote load) the data. The remote load buffers ensure that the overheads of remote loads for the pull are amortized through prefetching. We allocate a few shared pages within each stack to implement mailboxes. This pull-based model of direct communication is significantly more efficient and scalable than communication through the host processor in previous work [25], [26]. First, communication between nearby cores does not need to all go through the host processor, resulting in shorter latency and lower energy cost. Second, all threads can communicate asynchronously and in parallel, which eliminates global synchronization and avoids using only a limited number of host threads. Third, consumer threads can directly use or apply the remote data

while pulling, avoiding the extra copy cost.

Exception handling and other services: The runtime initiates all NDP cores with a default exception handler that forwards to the host OS. Custom handlers can also be registered for each NDP core. NDP threads use the runtime to request file I/O and related services from the host OS.

B. In-memory Analytics Frameworks

While one can program straight to the NDP runtime API, it is best to port to the NDP runtime domain-specific frameworks that expose higher-level programming interfaces. We ported three analytics frameworks. In each case, the framework utilizes the NDP runtime to optimize access patterns and task distribution. The NDP runtime hides all low-level details of synchronization and coherence.

First, we ported the Phoenix++ framework for in-memory MapReduce [51]. Map threads process input data and buffer the output locally. The shuffle phase follows the pull model, where each reduce thread will remotely fetch the intermediate data from the map threads' local memory. Once we ported Phoenix++, all Phoenix workloads run without modification. Second, we developed an in-memory graph framework that follows the gather-apply-scatter approach of GraphLab [43]. The framework handles the gather and scatter communication, while the high-level API visible to the programmer is similar to GraphLab and does not expose any of the details of NDP system. Third, we implemented a parallel deep neural network (DNN) framework based on Project Adam [44]. This framework supports both network training and prediction for various kinds of layers. Each layer in the network is vertically partitioned to minimize cross-thread communication. Forward and backward propagation across threads are implemented using communication primitives in the NDP runtime.

Applications developed with the MapReduce framework perform mostly sequential (streaming) accesses as map and reduce threads read their inputs. This is good for energy efficiency as it amortizes the overhead of opening a DRAM row (most columns are read) and moving a cacheline to the NDP core (most bytes are used). This is not necessarily the case for the graph framework that performs random accesses and uses only a fraction of the data structure for each vertex. To explore the importance of optimizing software for spatial locality for NDP systems, we developed a fourth framework, a second version of the graph framework that uses the same high-level API. While the original version uses a vertex-centric organization where computation accesses vertices and edges randomly, the second implementation is modeled after the X-Stream system that is edge-centric and streams edges which are arranged consecutively in memory [50]. We find that the edge streaming method is much better suited to NDP (see section VI).

C. Discussion

The current version of the NDP runtime does not implement load balancing. We expect load balancing to be handled in each analytics framework and most frameworks already do this (e.g., MapReduce). Our NDP runtime can support multiple analytics applications and multiple frameworks running concurrently by partitioning NDP cores and memory. The virtual memory and TLB support provide security isolation.

The NDP hardware and software are optimized for executing the highly-parallel phases with little temporal locality on NDP cores, while less-parallel phases with high temporal locality run on host cores. Division of labor is managed by framework developers given their knowledge about the locality and parallelism. Our experience with the frameworks we ported shows that communication and coordination between host and NDP cores is infrequent and involves small amounts of data (e.g., the results of a highly-parallel memory scan). The lack of fine-grained cache coherence between NDP and host cores is not a performance or complexity issue.

A key departure in our NDP system is the need for coarse-grained address interleaving. Conventional systems use fine-grained interleaving where sequential cachelines in the physical address space are interleaved across channels, ranks, and banks in a DDRx memory system. This optimizes bandwidth and latency for applications with both sequential and random access patterns. Unfortunately, fine-grained interleaving would eliminate most of the NDP benefits. The coarse-grained interleaving we use is ideal for execution phases that use NDP cores but can slow down phases that run on the host cores. In section VI, we show that this is not a major issue. Host cores are used with cache-friendly phases. Hence, while coarse-grained interleaving reduces the memory bandwidth available for some access patterns, once data is in the host caches execution proceeds at full speed. Most code that would benefit from fine-grained partitioning runs on NDP cores anyway.

V. Methodology

A. Simulation Models

We use zsim, a fast and accurate simulator for thousand-core systems [52]. We modified zsim to support fine-grained (cycle-by-cycle) multithreading for the NDP cores and TLB management. We also extended zsim with a detailed memory model based on DDR3 DRAM. We validated the model with DRAMSim2 [53] and against a real system. Timing parameters for 3D-stacked memory are conservatively inferred from publicly-available information and research literature [18], [38], [54]–[56].

Table I summarizes the simulated systems. Our conventional baseline system (Conv-DDR3) includes a 16-core OoO processor and four DDR3-1600 memory channels with 4 ranks per channel. We also simulate another system, Conv-3D, which combines the same processor with eight 3D

Host Processor	
Cores	16 x86-64 OoO cores, 2.6 GHz
L1I cache	32 KB, 4-way, 3-cycle latency
L1D cache	32 KB, 8-way, 4-cycle latency
L2 cache	private, 256 KB, 8-way, 12-cycle latency
L3 cache	shared, 20 MB, 8 banks, 20-way, 28-cycle latency
TLB	32 entries, 2 MB page, 200-cycle miss penalty
NDP Logic	
Cores	in-order fine-grained MT cores, 1 GHz
L1I cache	32 KB, 2-way, 2-cycle latency
L1D cache	32 KB, 4-way, 3-cycle latency
TLB	16 entries, 2 MB page, 120-cycle miss penalty
DDR3-1600	
Organization	32 GB, 4 channels $\times$ 4 ranks, 2 Gb, x8 device
Timing Parameters	$t_{CK} = 1.25$ ns, $t_{RAS} = 35.0$ ns, $t_{RCD} = 12.5$ ns $t_{CAS} = 12.5$ ns, $t_{WR} = 15.0$ ns, $t_{RP} = 12.5$ ns
Bandwidth	12.8 GBps $\times$ 4 channels
3D Memory Stack	
Organization	32 GB, 8 layers $\times$ 16 vaults $\times$ 8 stacks
Timing Parameters	$t_{CK} = 1.6$ ns, $t_{RAS} = 22.4$ ns, $t_{RCD} = 11.2$ ns $t_{CAS} = 11.2$ ns, $t_{WR} = 14.4$ ns, $t_{RP} = 11.2$ ns
Serial links	160 GBps bidirectional, 8-cycle latency
On-chip links	16 Bytes/cycle, 4-cycle zero-load delay

Table I
THE KEY PARAMETERS OF THE SIMULATED SYSTEMS.

memory stacks connected into four chains. The serial links of each chain require roughly the same number of pins from the processor chip as a 64-bit DDR3 channel [18], [37]. Both systems use closed-page policy. Each serial link has an 8-cycle latency, including 3.2 ns for SerDes [55]. The on-chip NoC in the logic layer is modeled as a 4×4 2D-mesh between sixteen vaults with 128-bit channels. We assume 3 cycles for router and 1 cycle for wire as the zero-load delay [57], [58]. Finally, the NDP system extends the conventional 3D memory system by introducing a number of simple cores with caches into the logic layer. We use 64-byte lines in all caches by default.

B. Power and Area Models

We assume 22 nm technology process for all logic, and that the area budget of the logic layer in the 3D stack is 100 mm^2. We use McPAT 1.0 to estimate the power and area of the host processor and the NDP cores [59]. We calculate dynamic power using its peak value and core utilization statistics. We also account for the overheads of the FPUs. We use CACTI 6.5 for cache power and area [60]. The NDP L1 caches use the ITRS-HP process for the peripheral circuit and the ITRS-LSTP process for the cell arrays. The use of ITRS-LSTP transistors does not violate timing constraints due to the lower frequency of NDP cores.

We use the methodology in [61] to calculate memory energy. DDR3 IDD values are taken from datasheets. For 3D memory stacks, we scale the static power with different bank organization and bank numbers, and account for the replicated peripheral circuits for each vault. For dynamic power, we scale ACT/PRE power based on the smaller page size and reduced latency. Compared to DDR3, RD/WR power increases due to the wider I/O of 3D stacking, but drops due to the use of smaller banks and shorter global wires (TSVs). Overall, our 3D memory power model results in roughly 10 to 20 pJ/bit, which is close to but more conservative than the numbers reported in HMC literature [54], [62].

We use Orion 2.0 for interconnect modeling [63]. The vault router runs at 1 GHz, and has 4 I/O ports with 128-bit flit width. Based on the area of the logic layer, we set wire length between two vaults to 2.5 mm. We assume that each serial link and SerDes between stacks and the host processor consume 1 pJ/bit for idle packets, and 3 pJ/bit for data packets [25], [55], [62]. We also model the overheads of routing between stacks in the host processor.

C. Workloads

We use the frameworks discussed in section IV: Phoenix++ for in-memory MapReduce analytics [51], the two implementations of the graph processing based on the gather-apply-scatter model [43], [50], and a deep neural network framework [44]. We choose a set of representative workloads for each framework described in Table II. Graph applications compute on real-world social networks and online reviews obtained from [64]. Overall, the workloads cover a wide range of computation and memory patterns. For instance, histogram (Hist) and linear regression (LinReg) do simple linear scans; PageRank is both read- and write-intensive; ALS requires complex matrix computation to derive the feature vectors; ConvNet needs to transfer lots of data between threads; MLP and dA (denoising autoencoder) have less communication due to combined propagation data.

We also implement one application per framework that uses both the host processor and NDP cores in different phases with different locality characteristics. They are similar to real-world applications with embedded, memory-intensive kernels. FisherScoring is an iterative logistic regression method. We use MapReduce on NDP cores to calculate the dataset statistics to get the gradient and Hessian matrix, and then solve the optimal parameters on the host processor. KCore decomposition first computes the maximal induced subgraph where all vertices have degree at least k using NDP cores. Then, the host processor calculates connected components on the smaller resultant graph. ConvNet-Train trains multiple copies of the LeNet-5 Convolutional Neural Network [65]. It uses NDP cores to process the input images for forward and backward propagation, and the host processor will periodically collect the parameter updates and send new parameters to each NDP worker [44].

VI. EVALUATION

We now present the results of our study, focusing on the key insights and trade-offs in the design exploration. Unless otherwise stated, the graph workloads use the edge-centric implementation of the graph framework. Due to space

Framework	Application	Data type	Data element	Input dataset	Note
MapReduce	Hist	Double	8 Bytes	Synthetic 20 GB binary file	Large intermediate data
	LinReg	Double	8 Bytes	Synthetic 2 GB binary file	Linear scan
	grep	Char	1 Bytes	3 GB text file	Communication-bound
	FisherScoring	Double	8 Bytes	Synthetic 2 GB binary file	Hybrid and iterative
Graph	PageRank	Double	48 Bytes	1.6M nodes, 30M edges social graph	Read- and write-intensive
	SSSP	Int	32 Bytes	1.6M nodes, 30M edges social graph	Unbalanced load
	ALS	Double	264 Bytes	8M reviews for 253k movies	Complex matrix computation
	KCore	Int	32 Bytes	1.6M nodes, 30M edges social graph	Hybrid
DNN	ConvNet	Double	8 Bytes	MNIST dataset, 70k 32×32 images	Partial connected layers
	MLP	Double	8 Bytes	MNIST dataset, 70k 32×32 images	Fully connected layers
	dA	Double	8 Bytes	Synthetic 500-dimension input data	Fully connected layers
	ConvNet-Train	Double	8 Bytes	MNIST dataset, 70k 32×32 images	Hybrid and iterative

Table II
THE KEY CHARACTERISTICS OF MAPREDUCE, GRAPH, AND DNN WORKLOADS AND THEIR DATASETS.

limitations, in some cases we present results for the most representative subset of applications: Hist for MapReduce, PageRank for graph, and ConvNet for DNN.

A. NDP Design Space Exploration

Compute/memory bandwidth balance: We first explore the balance between compute and memory bandwidth in the NDP system. We use a single-stack configuration with 16 vaults and vary the number of cores per vault (1 to 16), their frequency (0.5 or 1 GHz), and the number of threads per core (1 to 4). The maximum bandwidth per stack is 160 Gbytes/s. Figure 4 shows both the performance and maximum vault bandwidth utilization.

Performance scales almost linearly with up to 8 cores and benefits significantly from higher clock frequency and 2 threads per core. Beyond 8 cores, scaling slows down for many workloads due to bandwidth saturation. For PageRank, there is a slight drop due to memory congestion. The graph and DNN workloads saturate bandwidth faster than MapReduce workloads. Even with the edge-centric scheme, graph workloads still stress the random access bandwidth, as only a fraction of each accessed cacheline is utilized (see Figure 6). DNN workloads also require high bandwidth as they work on vector data. In contrast, MapReduce workloads perform sequential accesses, which have high cacheline utilization and perform more column accesses per opened DRAM row.

Overall, the applications we study need no more than eight 2-threaded cores running at 1 GHz to achieve balance between the compute and memory resources. Realistic area constraints for the logic die further limit us to 4 cores per vault. Thus, for the rest of the paper, we use four 2-threaded cores at 1 GHz. This configuration requires 61.7 mm^2 for the NDP cores and caches, while the remaining 38.3 mm^2 are sufficient for the DRAM controllers, the interconnect, and the circuitry for external links.

NDP memory hierarchy: Figure 5 shows the relative performance for using different stack structures. HMC-like stack uses 16 vaults with 64-bit data bus, and HBM-like stack uses 8 vaults with 128-bit data bus. We keep the same total number of data TSVs between the two stacks. Performance is less sensitive to the number of vaults per stack and the width of the vault bus. HBM is preferred for DNN workloads because they usually operate on vectors where wider buses could help with prefetching.

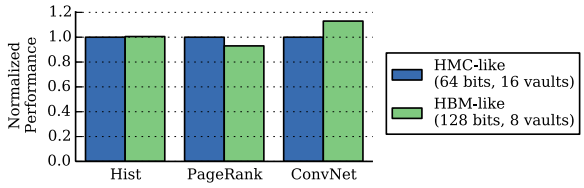

Figure 5. Performance impact of stack design.

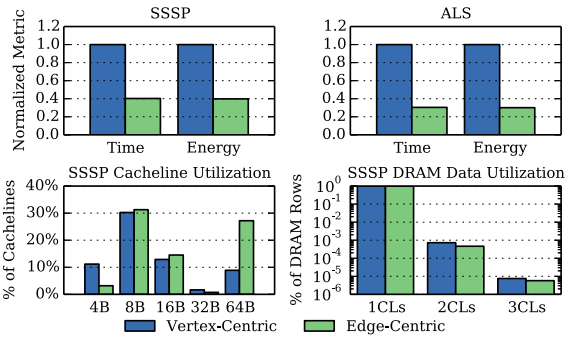

Figure 6. Vertex-centric and edge-centric graph frameworks.

We have also looked into using different cache structures. When varying L1 cacheline sizes, longer cachelines are always better for MapReduce workloads because their streaming nature (more spatial locality) amortizes the higher cache miss penalty in terms of time and energy. For graph and DNN workloads, the best cacheline size is 64 bytes; longer cachelines lead to worse performance due to the lack of spatial locality. Using a 256-Kbyte L2 cache per core leads to no obvious improvement for all workloads, and even introduces extra latency in some cases.

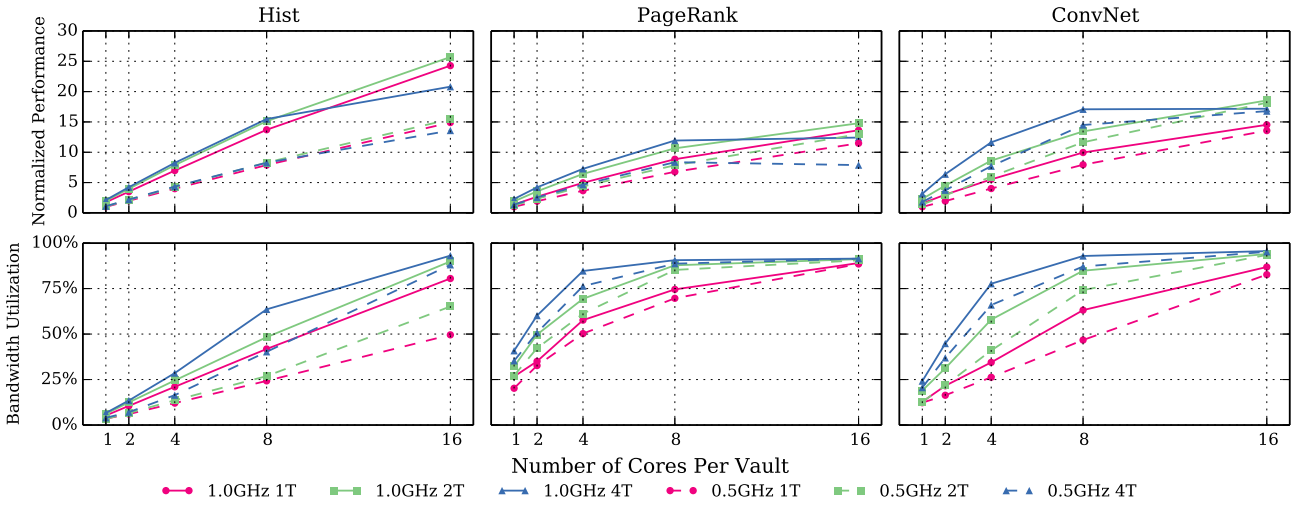

Figure 4. Performance scaling and bandwidth utilization for a single-stack NDP system.

The impact of software optimizations: The most energy-efficient way to access memory is to perform sequential accesses. In this case, the energy overhead of fetching a cacheline and opening a DRAM row is amortized by using (nearly) all bytes in the cacheline or DRAM row. While the hardware design affects efficiency as it determines the energy overheads and the width of the row, it is actually the software that determines how much spatial locality is available during execution.

Figure 6 compares the performance, energy, cacheline utilization, and DRAM row utilization for the two implementations of the graph framework (using open-page policy). The edge-centric implementation provides a 2.9x improvement in both performance and energy over the vertex-centric implementation. The key advantage is that the edge-centric scheme optimizes for spatial locality (streaming, sequential accesses). The cacheline utilization histogram shows that the higher spatial locality translates to a higher fraction of the data used within each cacheline read from memory, and a lower total number of cachelines that need to be fetched (not shown in the figure).

Note that the number of columns accessed per DRAM row is rather low overall, even with the edge-centric design. This is due to several reasons. First, spatial locality is still limited, usually less than column size. Second, these workloads follow multiple data streams that frequently cause row conflicts within banks. Finally, the short refresh interval in DDR3 (7.8μs) prevents the row buffer from being open for the very long time needed to capture further spatial locality. Overall, we believe there is significant headroom for software and hardware optimizations that further improve the spatial locality and energy efficiency of NDP systems.

NDP scaling: We now scale the number of stacks in the NDP system to observe scaling bottlenecks due to communication between multiple stacks. We use two cores per vault with 16 vaults per stack. The host processor can directly connect with up to 4 stacks (limited by pin constraints). Hence, for the configurations with 8 and 16 stacks, we connect up to 4 stacks in a chain as shown in Figure 1. Figure 7 shows that applications scale very well up to 16 stacks. With 8 stacks, the average bandwidth on inter-stack links is less than 2 Gbytes/s out of the peak of 160 Gbytes/s per link. Even in communication-heavy phases, the traffic across stacks rarely exceeds 20% of the peak throughput. Most communication traffic is handled within each stack by the network on chip, and only a small percentage of communication needs to go across the serial links.

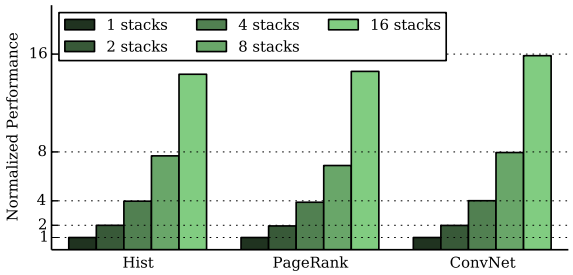

Figure 7. Performance as a function of the number of stacks.

B. Performance and Energy Comparison

We now compare the NDP system to the baseline Conv-DDR3 system, the Conv-3D system (3D stacking without processing in the logic layer), and the base NDP system in section III-A that uses the host processor for communication between NDP threads [25]. We use four 2-threaded cores per

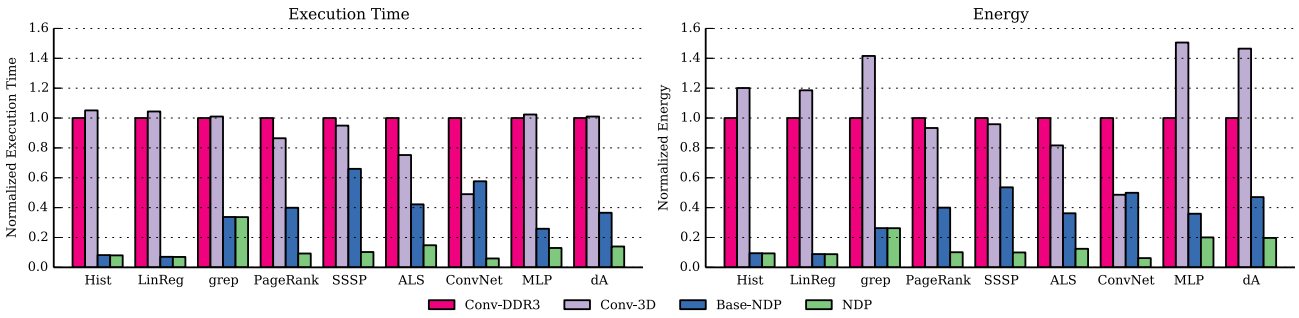

Figure 8. Performance and energy comparison between Conv-DDR3, Conv-3D, Base-NDP and NDP systems.

vault and 16 vaults per stack. This means 512 NDP cores (1024 threads) across 8 stacks.

Figure 8 shows the performance and energy comparison between the four systems. The Conv-3D system provides significantly higher bandwidth than the DDR3 baseline due to the use of high-bandwidth 3D memory stacks. This is particularly important for the bandwidth-bound graph workloads that improve by up to 25% in performance and 19% in energy, but less critical for the other two frameworks. The Conv-3D system is actually less energy-efficient for these workloads due to the high background power of the memory stacks and the underutilized high-speed links. The slight performance drop for Hist, MLP, etc. is because the sequential accesses are spread across too many channels in Conv-3D, and thus there are fewer requests in each channel that can be coalesced and scheduled to utilize the opened row buffers.

The two NDP systems provide significant improvement over both the Conv-DDR3 and the Conv-3D systems in terms of performance *and* energy. The base-NDP system is overall 3.5x faster and 3.4x more energy efficient over the DDR3 baseline. Our NDP system provides another 2.5x improvement over the base-NDP, and achieves 3-16x better performance and 4-16x less energy over the base-DDR3 system. This is primarily due to the efficient communication between NDP threads (see section III and IV). The benefits are somewhat lower for grep, MLP and dA. Grep works on 1-byte char data which requires less bandwidth per computation. MLP and dA are computationally intensive and their performance is limited by the capabilities of the simple NDP cores.

Figure 9 provides further insights into the comparison by breaking down the average power consumption. Note that the four systems are engineered to consume roughly the same power and utilize the same power delivery and cooling infrastructure. The goal is to deliver the maximum performance within the power envelope and avoid energy waste. Both conventional systems consume half power in the memory system and half in the processor. The cores are mostly stalled waiting for memory, but idleness is not

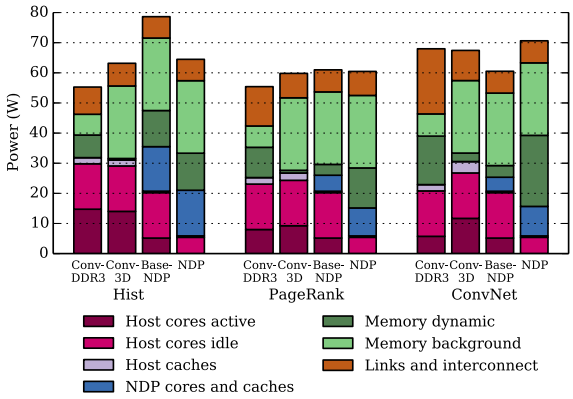

Figure 9. System power breakdown (host processor + 8 stacks).

sufficiently long to invoke deeper low power modes (e.g., C3 to C6 modes). The DDR3 channels are burning dynamic energy to move data with little reuse for amortization, with relatively low background energy due to the modest capacity. In the Conv-3D memory system, background power is much higher due to the large number of banks and replicated peripheral circuitry. Dynamic energy decreases due to efficient 3D structure and TSV channels, but bandwidth is seriously underutilized.

For the NDP system, dynamic memory power is higher as there are now hundreds of NDP threads issuing memory accesses. The host cores in our NDP system are idle for long periods now and can be placed into deep low-power modes. However, for the base NDP system, workloads which have iterative communication require the host processor to coordinate, thus the processor spends more power. The NDP cores across the 8 stacks consume significant power, but remain well-utilized. Replacing these cores with energy-efficient custom engines would provide further energy improvement (up to an additional 20%), but these savings are much lower than those achieved by eliminating data movement with NDP. A more promising direction is to focus the design of custom engines on performance improvements (rather than

power), e.g., by exploiting SIMD for MapReduce and DNN workloads. This approach is also limited by the available memory bandwidth (see Figure 4).

We do not include a comparison with baseline systems replacing OoO cores with many small cores at the host processor side. The Conv-DDR3 system is already bandwidth-limited, therefore using more cores would not change its performance. A Conv-3D system would achieve performance improvements with more small cores but would still spend significant energy on the high-speed memory links. Moreover, as we can infer from Figure 9 it would lead to a power problem by significantly increasing the dynamic power spent on the links. Overall, it is better to save interconnect energy by moving a large number of small cores close to memory *and* maintaining OoO cores on the host for the workloads that actually need high ILP [66].

Regarding thermal issues, our system is nearly power-neutral compared with the Conv-3D system. This can be further improved with better power management of the links (e.g., turn off some links when high bandwidth is not needed), as they draw significant static power but are seriously underutilized. Also, our power overhead per stack is close to previous work [25], which has already demonstrated the feasibility of adding logic into HMC [67].

C. System Challenges

The NDP system uses coarse-grained rather than fine-grained address interleaving. To understand the impact of this change, we run the SPEC CPU2006 benchmarks on the Conv-3D system with coarse-grained and fine-grained interleaving. All processing is performed on the host processor cores. For the benchmarks that cache reasonably well in the host LLC (perlbench, gcc, etc.), the impact is negligible (<1%). Among the memory-intensive benchmarks (libquantum, mcf, etc.), coarse-grained interleaving leads to an average 10% slowdown (20.7% maximum for GemsFDTD). Overall, this performance loss is not trivial but it is not huge either. Hence, we believe it is worth it to use coarse-grained interleaving to enable the large benefits from NDP for in-memory analytics, even if some host-side code suffers a small degradation. Nevertheless, we plan to study adaptive interleaving schemes in future work.

Finally, Figure 10 compares the performance of the hybrid workloads on the four systems. Energy results are similar. The memory-intensive phases of these workloads execute on NDP cores, leading to overall performances gain of 2.5x to 13x over the DDR3 baseline. The compute-intensive phases execute on the host processor on all systems. ConvNet-Train has negligible compute-intensive work. The baseline NDP system uses the host processor for additional time in order to coordinate communication between NDP cores. In our NDP system, in contrast, NDP cores coordinate directly. While this increases their workload (see KCore), this work is parallelized and executes faster than on the host processor.

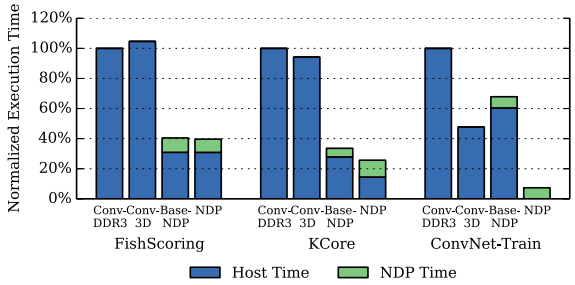

Figure 10. Hybrid workload performance comparison.

VII. Conclusion

We presented the hardware and software features necessary for efficient and practical near-data processing for in-memory analytics (MapReduce, graph processing, deep neural networks). By placing simple cores close to memory, we eliminate the energy waste for data movement in these workloads with limited temporal locality. To support non-trivial software patterns, we introduce simple but scalable NDP hardware support for coherence, virtual memory, communication and synchronization, as well as a software runtime that hides all details of NDP hardware from the analytics frameworks. Overall, we demonstrate up to 16x improvement on both performance and energy over the existing systems. We also demonstrate the need of the coherence and communication support in NDP hardware and the need to optimize software for spatial locality in order to maximize NDP benefits.

Acknowledgments

The authors want to thank Mark Horowitz, Heonjae Ha, Kenta Yasufuku, Makoto Takami, and the anonymous reviewers for their insightful comments on earlier versions of this paper. This work was supported by the Stanford Pervasive Parallelism Lab, the Stanford Experimental Datacenter Lab, Samsung, and NSF grant SHF-1408911.

References

[1] H. Esmaeilzadeh *et al.*, "Dark silicon and the end of multicore scaling," in *ISCA-38*, 2011, pp. 365–376.
[2] S. Keckler, "Life After Dennard and How I Learned to Love the Picojoule," Keynote in MICRO-44, Dec. 2011.
[3] M. Horowitz *et al.*, "Scaling, power, and the future of CMOS," in *IEDM-2005*, Dec 2005, pp. 7–15.
[4] Computing Community Consortium, "Challenges and Opportunities with Big Data," http://cra.org/ccc/docs/init/bigdatawhitepaper.pdf, 2012.
[5] P. M. Kogge, "EXECUBE-A New Architecture for Scaleable MPPs," in *ICCP-1994*, 1994, pp. 77–84.
[6] P. M. Kogge *et al.*, "Pursuing a Petaflop: Point Designs for 100 TF Computers Using PIM Technologies," in *FMPC-6*, 1996, pp. 88–97.
[7] D. Patterson *et al.*, "A Case for Intelligent RAM," *Micro, IEEE*, vol. 17, no. 2, pp. 34–44, 1997.
[8] M. Hall *et al.*, "Mapping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture," in *SC'99*, 1999, p. 57.
[9] M. Oskin *et al.*, "Active Pages: A Computation Model for Intelligent Memory," in *ISCA-25*, 1998, pp. 192–203.

[10] Y. Kang *et al.*, "FlexRAM: Toward an Advanced Intelligent Memory System," in *ICCD-30*, 2012, pp. 5–14.

[11] M. Motoyoshi, "Through-Silicon Via (TSV)," *Proceedings of the IEEE*, vol. 97, no. 1, pp. 43–48, Jan 2009.

[12] A. W. Topol *et al.*, "Three-Dimensional Integrated Circuits," *IBM Journal of Research and Development*, vol. 50, no. 4.5, pp. 491–506, July 2006.

[13] R. Balasubramonian *et al.*, "Near-Data Processing: Insights from a MICRO-46 Workshop," *Micro, IEEE*, vol. 34, no. 4, pp. 36–42, July 2014.

[14] J. Carter *et al.*, "Impulse: Building a Smarter Memory Controller," in *HPCA-5*, 1999, pp. 70–79.

[15] Z. Fang *et al.*, "Active Memory Operations," in *ICS-21*, 2007, pp. 232–241.

[16] S. J. Souri *et al.*, "Multiple Si Layer ICs: Motivation, Performance Analysis, and Design Implications," in *DAC-37*, 2000, pp. 213–220.

[17] S. Das *et al.*, "Technology, Performance, and Computer-Aided Design of Three-dimensional Integrated Circuits," in *ISPD-2004*, 2004, pp. 108–115.

[18] Hybrid Memory Cube Consortium, "Hybrid Memory Cube Specification 1.0," 2013.

[19] JEDEC Standard, "High Bandwidth Memory (HBM) DRAM," JESD235, 2013.

[20] K. Puttaswamy and G. H. Loh, "Implementing Caches in a 3D Technology for High Performance Processors," in *ICCD-2005*, 2005, pp. 525–532.

[21] G. H. Loh, "Extending the Effectiveness of 3D-stacked DRAM Caches with an Adaptive Multi-Queue Policy," in *MICRO-42*, 2009, pp. 201–212.

[22] X. Jiang *et al.*, "CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms," in *HPCA-16*, 2010, pp. 1–12.

[23] G. H. Loh and M. D. Hill, "Supporting Very Large DRAM Caches with Compound-Access Scheduling and MissMap," *Micro, IEEE*, vol. 32, no. 3, pp. 70–78, May 2012.

[24] D. Jevdjic *et al.*, "Die-stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *ISCA-40*, 2013, pp. 404–415.

[25] S. Pugsley *et al.*, "NDC: Analyzing the Impact of 3D-Stacked Memory+ Logic Devices on MapReduce Workloads," in *ISPASS-2014*, 2014.

[26] A. Farmahini-Farahani *et al.*, "NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *HPCA-21*, Feb 2015, pp. 283–295.

[27] J. Ahn *et al.*, "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in *ISCA-42*, 2015, pp. 105–117.

[28] R. Nair *et al.*, "Active Memory Cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17:1–17:14, March 2015.

[29] T. Kgil *et al.*, "PicoServer: Using 3D Stacking Technology to Enable a Compact Energy Efficient Chip Multiprocessor," in *ASPLOS-12*, 2006, pp. 117–128.

[30] A. Gutierrez *et al.*, "Integrated 3d-stacked server designs for increasing physical density of key-value stores," in *ASPLOS-19*, 2014, pp. 485–498.

[31] D. Zhang *et al.*, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *HPDC-23*, 2014, pp. 85–98.

[32] P. Ranganathan, "From microprocessors to nanostores: Rethinking data-centric systems," *Computer*, vol. 44, no. 3, pp. 39–48, 2011.

[33] D. Fick *et al.*, "Centip3De: A Cluster-Based NTC Architecture With 64 ARM Cortex-M3 Cores in 3D Stacked 130 nm CMOS," *JSSC*, vol. 48, no. 1, pp. 104–117, Jan 2013.

[34] G. H. Loh, "3D-Stacked Memory Architectures for Multi-core Processors," in *ISCA-35*, June 2008, pp. 453–464.

[35] Y. Pan and T. Zhang, "Improving VLIW Processor Performance Using Three-Dimensional (3D) DRAM Stacking," in *ASAP-20*, July 2009, pp. 38–45.

[36] D. H. Woo *et al.*, "An Optimized 3D-stacked Memory Architecture by Exploiting Excessive, High-Density TSV Bandwidth," in *HPCA-16*, Jan 2010, pp. 1–12.

[37] E. Cooper-Balis *et al.*, "Buffer-On-Board Memory Systems," in *ISCA-39*, 2012, pp. 392–403.

[38] D. U. Lee *et al.*, "25.2 a 1.2v 8gb 8-channel 128gb/s high-bandwidth memory (hbm) stacked dram with effective microbump i/o test methods using 29nm process and tsv," in *ISSCC-2014*, Feb 2014, pp. 432–433.

[39] P. Dlugosch *et al.*, "An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing," *TPDS*, p. 1, 2014.

[40] T. Zhang *et al.*, "A 3D SoC design for H.264 application with on-chip DRAM stacking," in *3DIC-2010*, Nov 2010, pp. 1–6.

[41] ARM, "Cortex-A7 Processor," http://www.arm.com/products/processors/cortex-a/cortex-a7.php.

[42] R. Alverson *et al.*, "The tera computer system," in *ACM SIGARCH Computer Architecture News*, 1990, pp. 1–6.

[43] J. E. Gonzalez *et al.*, "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," in *OSDI-10*, 2012, pp. 17–30.

[44] T. Chilimbi *et al.*, "Project Adam: Building an Efficient and Scalable Deep Learning Training System," in *OSDI-11*, 2014, pp. 571–582.

[45] J. Navarro *et al.*, "Practical, Transparent Operating System Support for Superpages," in *OSDI-5*, 2002, pp. 89–104.

[46] M. Ferdman *et al.*, "Cuckoo directory: A scalable directory for many-core systems," in *HPCA-17*, 2011, pp. 169–180.

[47] D. Sanchez and C. Kozyrakis, "SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding," in *HPCA-18*, February 2012.

[48] J. H. Ahn *et al.*, "Scatter-Add in Data Parallel Architectures," in *HPCA-11*, Feb 2005, pp. 132–142.

[49] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI-6*, 2004, pp. 10–10.

[50] A. Roy *et al.*, "X-Stream: Edge-centric Graph Processing Using Streaming Partitions," in *SOSP-24*, 2013, pp. 472–488.

[51] J. Talbot *et al.*, "Phoenix++: Modular MapReduce for Shared-memory Systems," in *MapReduce-2011*, 2011, pp. 9–16.

[52] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems," in *ISCA-40*, 2013, pp. 475–486.

[53] P. Rosenfeld *et al.*, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan 2011.

[54] J. T. Pawlowski, "Hybrid Memory Cube (HMC)," Presented in HotChips 23, 2011.

[55] G. Kim *et al.*, "Memory-Centric System Interconnect Design With Hybrid Memory Cubes," in *PACT-22*, Sept 2013, pp. 145–155.

[56] C. Weis *et al.*, "Design Space Exploration for 3D-stacked DRAMs," in *DATE-2011*, March 2011, pp. 1–6.

[57] B. Grot *et al.*, "Express Cube Topologies for On-Chip Interconnects," in *HPCA-15*, Feb 2009, pp. 163–174.

[58] J. Balfour and W. J. Dally, "Design Tradeoffs for Tiled CMP On-chip Networks," in *ICS-20*, 2006, pp. 187–198.

[59] S. Li *et al.*, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *MICRO-42*, 2009, pp. 469–480.

[60] N. Muralimanohar *et al.*, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *MICRO-40*, 2007, pp. 3–14.

[61] Micron Technology Inc., "TN-41-01: Calculating Memory System Power for DDR3," http://www.micron.com/products/support/power-calc, 2007.

[62] J. Jeddeloh and B. Keeth, "Hybrid Memory Cube New DRAM Architecture Increases Density and Performance," in *VLSIT*, June 2012, pp. 87–88.

[63] A. B. Kahng *et al.*, "ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-stage Design Space Exploration," in *DATE-2009*.

[64] "Stanford Large Network Dataset Collection," http://snap.stanford.edu/data/index.html, available Online.

[65] Y. Lecun *et al.*, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[66] U. Hölzle, "Brawny cores still beat wimpy cores, most of the time," *IEEE Micro*, vol. 30, no. 4, 2010.

[67] Y. Eckert *et al.*, "Thermal Feasibility of Die-Stacked Processing in Memory," in *2nd Workshop on Near-Data Processing (WoNDP)*, December 2014.