An Architecture for Integrated Near-Data Processors

ERIK VERMIJ, LEANDRO FIORIN, and RIK JONGERIUS, IBM Research, The Netherlands
CHRISTOPH HAGLEITNER and JAN VAN LUNTEREN, IBM Research–Zurich, Switzerland
KOEN BERTELS, Delft University of Technology, The Netherlands

To increase the performance of data-intensive applications, we present an extension to a CPU architecture that enables arbitrary near-data processing capabilities close to the main memory. This is realized by introducing a component attached to the CPU system-bus and a component at the memory side. Together they support hardware-managed coherence and virtual memory support to integrate the near-data processors in a shared-memory environment. We present an implementation of the components, as well as a system-simulator, providing detailed performance estimations. With a variety of synthetic workloads we demonstrate the performance of the memory accesses, the mixed fine- and coarse-grained coherence mechanisms, and the near-data processor communication mechanism. Furthermore, we quantify the inevitable start-up penalty regarding coherence and data writeback, and argue that near-data processing workloads should access data several times to offset this penalty. A case study based on the Graph500 benchmark confirms the small overhead for the proposed coherence mechanisms and shows the ability to outperform a real CPU by a factor of two.

CCS Concepts: • **Computer systems organization** → **Other architectures; Heterogeneous (hybrid) systems**;

Additional Key Words and Phrases: Computer architecture, data locality, near-data processing, coherence, virtual memory, graph500

ACM Reference format:

Erik Vermij, Leandro Fiorin, Rik Jongerius, Christoph Hagleitner, Jan van Lunteren, and Koen Bertels. 2017. An Architecture for Integrated Near-Data Processors. *ACM Trans. Archit. Code Optim.* 14, 3, Article 30 (August 2017), 25 pages.

<https://doi.org/10.1145/3127069>

1 INTRODUCTION

The world's population is producing and storing more digital data than ever before, by means of a wide variety of social media, and modern voice-, text-, or image-based communication methods. While Facebook and Google are traditional examples of data-driven companies, big-data has found its way to many different businesses and fields [39]. Creating business value from huge amounts of data is becoming an ever more important task for computer systems. Real-time graph analytics,

This work was conducted in the context of the joint ASTRON and IBM DOME project and is funded by the Netherlands Organization for Scientific Research (NWO), the Dutch Ministry of EL&I, and the Province of Drenthe.

Authors' addresses: E. Vermij, L. Fiorin, and R. Jongerius, IBM Research, ASTRON & IBM Center for Exascale Technology, P.O. Box 2, 7990 AA Dwingeloo, the Netherlands; emails: {erik.vermij, leandro.fiorin, r.jongerius}@nl.ibm.com; C. Hagleitner and J. van Lunteren, IBM Research - Zurich, Säeumerstrasse 4, Rüschlikon 8803, Switzerland; emails: {hle, jvl}@zurich.ibm.com; K. Bertels, EEMCS, Computer Engineering, P.O. Box 5031, 2600 GA Delft, the Netherlands; email: k.l.m.bertels@tudelft.nl.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM 1544-3566/2017/08-ART30 \$15.00

<https://doi.org/10.1145/3127069>

online fraud-detection, and cognitive computing are examples of data-intensive workloads posing even more challenging requirements on computer systems. Because of this, we have seen a growing interest in the role of data in computer systems. Unfortunately, not only are the tasks becoming harder, but also computer systems are becoming relatively worse at handling them: the increase in memory bandwidth and the reduction in memory latency do not keep up with the increase in processing capabilities, making compute nodes increasingly worse at handling data-intensive workloads.

One solution to cope with ever-increasing data-rates and data-sizes is to use more, relatively cheap, nodes. This is known as a *scale-out* solution, which has attracted much attention in the past few years and works well for embarrassingly parallel workloads. Many frameworks, such as, for instance, *Spark*, exist, which are either generic or workload-specific and offer massively parallel workload distribution on a standard cluster [55].

Scaling-up the hardware by making it more powerful or more specific for big-data problems is another way of increasing performance and is especially suited for problems that do not scale linearly with the number of nodes. An increase of performance can be achieved in several ways, ranging from the use of accelerators and co-processors to the support of large and different type of memories. For example, the IBM POWER8 systems offer a high memory bandwidth and large amounts of DRAM per socket, making it an interesting platform for data-intensive applications. GPUs are well recognized as a platform for boosting the performance of data analytics with respect to CPUs, due to their high bandwidth, parallelism, and latency hiding mechanism [40], making it the fastest single-node solution for the data-intensive Graph500 benchmark [22]. With the rise of various types of non-volatile memory [32, 34], we get high bandwidths to large persistent storage volumes, much higher than what disks can offer. All these and likewise products have useful memory-related features, but lack others. For example, the CPU's DRAM offers the storage capacity, but only mediocre bandwidths compared to GPUs, and cacheline-sized accesses. GPUs and the Xeon Phi have a high memory bandwidth, but suffer from a high latency and a relatively small storage volume.

To bridge the gap in memory bandwidth and latency between what today's data-intensive workloads demand and what existing computer architectures can offer, the processing should take place as close to the data as possible, avoiding interconnect bottlenecks. This paradigm is referred to as "near-data processing" and has recently been re-discovered [6], motivated by the availability of new technologies such as 3D stacking, big-data workloads with high degrees of parallelism, and programming models for distributed big-data applications. By introducing processing capabilities very close to the main memory of a CPU, it is possible to benefit from memory access features characterized by low latency, high bandwidth, small access granularity, and support for large memories. In this work, we propose an architectural extension to support near-data processing in an existing CPU ecosystem.

A high-level view of the proposed architecture is shown in Figure 1(a). The essential aspect is the two levels of memory controllers: at the CPU level, we have memory-technology agnostic memory controllers, while the technology-specific memory controllers are tightly coupled to the main memory. An industry example of such a setup is the memory system of the IBM POWER8 CPU [50], which has eight memory-technology agnostic memory channels each connecting to a "memory buffer" chip, holding four DDR3/4 memory controllers. Another example is a CPU connected to novel stacked memory like the Hybrid Memory Cube [25], having the technology-specific memory controller in the logic layer of the memory device. We propose the addition of near-data processing capabilities close to the technology-specific memory controllers, as shown in Figure 1(a). For completeness, we show default memory channels as well, to highlight the generality of the approach. It would also be possible to extend it with a second CPU socket or even a

Fig. 1. (a) High-level view of the proposed architecture, where compute capabilities are added close to the main memory of a CPU. (b) Shows the problem of integrating NDPs with a CPU: all relevant integration features stop at the memory controllers.

modern GPU, without breaking the concept of the proposal. The focus of this article is to discuss the architectural implications of adding the near-data processing capabilities and to evaluate how that affects software and workload partitioning.

The main contributions of this article are as follows:

- We propose a novel memory-centric architecture to enable *arbitrary* near-data processing in a contemporary server environment, by introducing a minimally invasive component at the CPU's system bus and a component in the memory system;
- We propose generic methods for data allocation and data locality management based on existing NUMA functionality for the near-data processors (NDPs) and the CPU;
- We propose generic hardware-managed methods for coherence and for accessing data in the global address space;
- We quantify the coherence and data writeback overheads and show the performance of the proposed architecture for a variety of small workloads as well as the Graph500 benchmark.

The remainder of this article is organized as follows: Section 2 describes related work and the motivation for the proposed research. In Section 3 data placement and memory management are discussed, followed by the proposed necessary hardware additions in Section 4. Section 5 presents a detailed discussion on coherence, while Section 6 discusses virtual memory management and access to remote data. A custom system simulator is presented in Section 7, followed by synthetic results in Section 8 and a case study in Section 9. Section 10 concludes the paper.

2 MOTIVATION AND RELATED WORK

2.1 Integrating Arbitrary Near-data Processing Capabilities

Several platforms have been proposed to deal with big-data applications. A case study of using existing node and cluster technology to execute big-data workloads is presented in Reference [26]. The work in Reference [11] proposes a system and board design around Flash and DRAM, targeting energy-efficient execution of big-data workloads. Another system-design proposal [1] targets the specific class of graph processing algorithms. Work in Reference [18] shows a generic architecture for big-data analytics, using stacked DRAM and its logic layer. In Reference [12], near-data processing has been investigated for accelerating big-data workloads with poor locality in solid-state disks.

Work in References [1, 4, 18, 41, 43] proposes architectures all based on exploiting the logic layer found in 3D-stacked DRAM devices, such as the Micron Hybrid Memory Cube (HMC). In Reference [3] the integration of light-weight cores on top of traditional DRAM is proposed, while Reference [35] adds small compute elements close to the DRAM banks.

We differentiate from all this work by not focusing on a particular type of NDP architecture (general purpose, reconfigurable, etc.), a particular type of memory technology (DRAM, HMC, HBM, etc.), or a particular type of applications, but on adding the hardware components and mechanisms needed for integrating near-data processing. This is done under the assumption of a high-level architectural concept as described in Section 1 and shown in Figure 1(a). To make a solid contribution, a clear definition of “integrated” needs to be specified. The objectives of a generic integration of any type of NDP following the architecture in Figure 1(a) are:

- (1) To not limit standard CPU performance;
- (2) To limit changes to the CPU and the OS (Linux);
- (3) To use standard OS level memory management;
- (4) To provide global, virtualized, and coherent memory.

First, workloads not using the NDPs should not be negatively affected, as that would break the general usability of the system. Second, changes to the CPU must be limited and as non-invasive as possible, to not break the delicate workings, overhaul roadmaps, and incur major costs. Changing non-trivial aspects of the OS kernel in a maintainable way can be considered equally challenging, given the community-like organization, the enormous complexity, and the high costs of quality software engineering. The last two items follow the trend in industry towards coherently attached coprocessors [51], the trend in industry towards a unified and coherent address space between CPU and coprocessor [28], and the literature (many are cited in the remainder of this section), which almost always describes one or more of these aspects to some degree, thus acknowledging the need for them. This is also reflected in a recent expert-opinion overview [14] about near-data processing, which, among others things, stresses the need for a unified shared address space, a unified memory model for both the CPU and the NDPs, communication between the various devices, and a re-visitation of solutions for coherence.

Before proposing a solution, it is necessary to have an understanding of the challenges involved in integrating NDPs to this extent. Figure 1(b) shows both the most fundamental property as well as the most fundamental problem associated with near-data processing. Being the barrier between the CPU and the memory system, all features relevant for system integration stop at the generic memory controllers. This implies that:

- The NDP cannot issue a load/store to an address outside its own memory channel, or to the caches at the CPU;
- The NDP cannot know whether a piece of data is stored in a cache at the CPU;
- The NDP cannot have a notion of virtual memory;
- The NDP observes only a subset of a data set, since data is striped over the various memory channels.

To integrate NDPs successfully, they need to be connected to the functionalities offered by the system bus.

2.2 Proposed Solutions and their Pitfalls

2.2.1 Extending the System Bus. Although a first idea would be to extend the system bus towards the NDPs, this is very undesirable for various reasons:

- Several operations on the coherence fabric of the system bus are latency critical, for example, a *snoop* or an *invalidate*. Extending these operations to the NDPs would have a dramatic negative impact on system performance [51].

- The hardware TLB *sync* and *invalidate* mechanism to support virtual memory in multi-processor systems [24] are latency critical as well, and extending these operations to the NDPs would also have a dramatic negative impact on system performance.
- For consistency reasons, caches at the CPU have to keep track of an evicted cache line until it is committed to the memory controller. By increasing the latency towards this commitment, caches will require more resources (tracking state-machines, etc.) for the same performance.

Therefore, extending the system bus would result in a much poorer CPU performance, which breaks the first goal described in Section 2.

2.2.2 Virtual Memory Solutions. Regarding virtual memory support, many NDP solutions propose the usage of a small TLB holding the translations, and describe this TLB to be filled by means of a driver or library [5, 18], or lack the description on how the TLB gets filled [41, 53]. Unfortunately, the effective to real address translation mapping is not static: memory pages allocated by the default OS allocation methods can be subject to physical migration, stealing, and several other operations [52]. For this reason, and many others, the system bus offers the TLB synchronization mechanisms stated before. A software managed solution would require a deep integration with the OS to monitor and temporally hold these migrations, which is a very difficult task resulting in a non-portable solution. Furthermore, a software managed TLB would be much slower compared to hardware managed ones.

Using pinned or memory-mapped (physically contiguous) memory [19, 35] avoids the need of virtual memory at the NDP but will not work when having data sets with a size close to the main memory capacity due to physical fragmentation, and requiring significant allocation times. Work in Reference [27] proposes to use a novel and completely decoupled page table for the NDPs, allowing for various optimizations and efficient translations, at the expense of integratability. Their proposal requires changes in the operating system and special allocation routines. Several of the already mentioned articles name the exploitation of transparent huge pages [42], or the exploitation of the fact that many page level translations can be grouped into larger chunks due to how the allocators work [21]. But, although important optimizations, they do not solve the underlying synchronization problem. None of the proposed solutions will provide true virtual memory support without substantial OS modifications, which is in contrast to the second goal of limiting OS changes as much as possible.

2.2.3 Coherence Solutions. Regarding coherence, various options have been explored. Putting the NDPs in a separate address space [1], with explicit data management by a driver, avoids the need of coherence, but is also not an integrated solution. Marking pages cache-inhibited [1, 16, 19, 35] makes working with them incredibly slow from the CPU's perspective as every access has to go through the non-cacheable unit [48], which is designed for managing memory-mapped devices, not for handling large amounts of data. Work in Reference [18] proposes the identification of a single cache in the entire system that can hold a certain page, which is stored and managed via the page tables. This is a strong idea, as it makes things explicit and thus simple but also raises many issues. For example, the method is not scalable to larger systems as the page tables have only so many free bits to identify caches, assuming that all memory accesses are done via a TLB, which is not the case for hardware prefetches, and it limits the sharing of data between cores. Lazy methods based on a rollback mechanism have been proposed in Reference [9], but the article lacks details required for a detailed comparison (e.g., description of the rollback mechanism and the proposed "conflict detection hardware"). Work in Reference [2] proposes a form of integrated near-data processing offloading small instructions to the memory system but requires significant changes to the delicate inner-core of a CPU and lacks significant details on the used coherence

Fig. 2. (a) A CPU environment enhanced with near-data processing capabilities. By introducing a component in the memory system and one at the system bus, we can extend the necessary integration features towards the NDPs, supporting arbitrary functionality. This connection is illustrated with the two arrows. (b) Both components have a virtualized and coherent view towards each other and towards the main memory.

methods and their performance. It, for example, assumes that data can only be in the local last-level cache of a core, which will not be true for modern CPUs. Work in Reference [41] proposes a generic hardware-managed method for coherence but lacks the ability to share data between CPU and NDP and furthermore does not support the access of remote data by an NDP, limiting the useability of the proposal. Handling coherence manually (flushing cache lines explicitly) is both hard and unpredictable as the hardware prefetchers can prefetch data the user/programmer believes is still in memory. Besides, software-driven cache flushes are very slow, especially on multi-socket systems.

Furthermore, both the cache-inhibited and the manual approach above do not provide us with a consistency model keeping track of when a write from the CPU has actually appeared in DRAM and can thus be read by the NDP. Without a hardware-managed coherence mechanism, a hardware synchronization mechanism is still needed to solve this issue. Other related works do not discuss coherence [5, 15, 53]. None of the proposed solutions will provide a workable coherence and consistency model without breaking the goals stated in Section 2. A similar analysis holds for accessing non-local data: none of the above mentioned works describes how an NDP can issue a load at the CPU's fabric to access data stored in a CPU cache, in another memory channel, or on another socket.

IBM's CAPI interface [51] solves a part of the problem: it offers a device a coherent view towards the CPU, but not towards local memory. This is visualized in Figure 2(b), where CAPI provides the line from the NDP towards the CPU, but not the lines towards the memory.

2.2.4 Memory Management Solutions. The problem regarding memory management is that the CPU wants its data striped over the various memory channels, while the NDP wants entire data sets to reside in the same memory channel. Furthermore, the OS does not allow the allocation of data within a specific memory channel. The work presented in Reference [41] solves this by using a custom allocator (or extending the OS allocator), which is able to allocate memory in both a striped as well as a contiguous fashion. Research in Reference [27] describes the use of a custom allocator as well, again requiring OS modifications. Both these works break our goal of limiting the OS changes as much as possible. Work in References [19] and [54] describes the concept of

page-level distribution of data across memory channels, but lacks a description of how that is implemented/realized.

2.3 Concept of the Proposed Solution

In this work, we propose a novel method for integrating an NDP that keeps the overhead to a minimum and achieves all four goals stated at the top of this section. This involves a component in the memory system, responsible for, for example, handling the coherence requirements of the NDP, as well as a single component at the CPU, together creating a “bridge” between CPU and NDP. This work complements some of the literature cited before. For example, a proposal for a specific core design utilizing the memory characteristics of the HMC could be extended with the methods proposed in this work, to coherently integrate it in the global address space.

3 DATA PLACEMENT AND MEMORY MANAGEMENT

By default, a CPU stripes accesses to subsequent cache lines or memory pages across different memory channels, to optimize bandwidth for common access patterns [50]. In this case, all memory channels represent a single big contiguous memory space called a *group*, with a certain modulo or hashing scheme to map accesses to the right channel, illustrated by the two default memory channels in Figure 1(a) contained in a single contiguous memory region. It is, however, also possible to make every memory channel its own *group*, implying that every memory channel holds a contiguous part of the physical address space, shown by the NDP memory channels in Figure 1(a). The actual configuration is set in firmware and typically multiple groups are only used when memory channels have different types or sizes of memory. For the architecture presented in this work, however, every memory channel constitute its own group, and by doing this, data sets can be mapped into a single memory channel. The methods to achieve this and the performance implications of this are discussed below, and are shown in Figure 2(a).

3.1 NDP Allocations

The OS has limited knowledge of the physical memory layout. The most detailed hardware level it is aware of are the NUMA domains, consisting of physical memory regions belonging to a certain node in a multi-processor system. The memory-allocator typically returns pages located at the node running the thread first accessing the page. The OS is not aware of the discussed memory channel *groups*, and does not care about how a NUMA domain is physically constructed. Therefore, to allocate space in the memory of a specific NDP is not trivial, and we solve it relying on the existing NUMA organization capabilities of the OS, complying with the goal of using standard OS level memory management.

The OS builds its NUMA organization tree [13] based on information supplied by the firmware. This tree is a hierarchical representation of nodes and possible subnodes in the system, including both the CPU and the memory. We can adjust the firmware in such a way that it reports the various memory channels and their physical address regions as NUMA memory nodes. In this way, we can use industry standard functionality to bind a process to a specific memory channel using *numactl* or bind a memory allocation to a specific memory channel using the *numa_alloc_onnode()* functionality supplied in *numa.h*, all possibly hidden in an NDP runtime library. By doing so, we introduce, from an NDP point of view, the concept of *local* and *remote* data. *Local* data should, whenever possible, be stored in the memory of the specific NDP, while *remote* data can be stored anywhere, even on a different socket or in GPU memory. It is clear that managing data locality is the key for obtaining reasonable NDP performance on the proposed architecture; this is, however, exactly the same as the optimizations usually performed on existing NUMA systems (e.g., a multi-socket CPU system).

In case a data set is allocated without a specific NUMA binding, but is accessed for the first time by an NDP, the data locality solves itself. In this case, the NDP access will raise a page-fault (discussed in Section 6), and the OS is asked for physical backing of the accessed virtual memory. Since the request for physical memory comes from a specific NUMA domain (the NDP), the physical pages will be taken from the NDP's local memory. This is a very valuable effect foremost of interest when the NDP initializes data sets.

3.2 Default Allocations

To make sure default allocations (i.e., NDP unrelated) do not end up in a single memory channel, dramatically reducing the achievable bandwidth to that data set and thereby breaking with the integration goals, extra care is needed. This can be solved by configuring the CPU node as having no memory of its own (*MEMORYLESS_NODE*) and changing the default OS allocation policy to *INTERLEAVE*. That way a page request will be forwarded, in a round-robin fashion, to the other nodes: the memory channels, either with or without NDP. By doing so, we realize a best-effort page level distribution of data across the available memory channels. Accesses to a single page still have to go through the same memory channel, but if we have many threads running at the CPU, as we have in modern big-data applications, the bandwidth achievable on a system-level will be like the striped cache line approach, as also indicated in Reference [18]. This mechanism does not imply a large penalty at allocation time, as physical page allocation is only done when the page is touched, and forwarding the allocation to another NUMA node is nothing more than traversing a small data-structure in the OS routines.

4 HARDWARE EXTENSIONS TO ENABLE NEAR-DATA PROCESSING

In Figure 2(a) we show the proposed architectural extension. At the memory side we introduce the NDP Manager (NDP-M), and at the CPU side we introduce the NDP Access Point (NDP-AP). Between the two components, a communication channel exists. In the remainder of this section these components will be discussed.

4.1 Communicating with the Memory System

As discussed, we include processing capabilities at the memory-side of the generic MC. To not interfere with the existing operations of the CPU, the communication with an NDP must be done *on top* of existing functionality. We added a lightweight NDP messaging extension to the protocol of the high-speed serial link of the memory channel. Instead of being load/store-only, it is extended to be able to carry both the original traffic, as well as communication between the NDP-M and NDP-AP introduced below. This is shown in Figure 2(a) with the two arrows. Communication between the NDP-AP and the generic memory controllers is done by using direct-addressing on the system bus. To ensure the correct functioning of the coherence mechanisms described in this work, messages can by default not overtake “normal” cache lines traveling up or down the memory channel.

4.2 Near-data Processor Manager

To support virtual memory, coherence, remote accesses, and communication capabilities for NDPs in the memory system, we introduce the NDP-M, as shown in Figure 2(a). In Figure 3(a), we show the architecture of the NDP-M, with its main functional blocks, described below. At the bottom the interface towards the high-speed link is shown, and at the top the interfaces towards the memory controllers are shown. On the side there are two types of interfaces for the NDP: one for handling messages between the NDP and applications and one or several interfaces for the memory accesses. The NDP-M is responsible for:

Fig. 3. (a) Main functional blocks within the near-data processor manager (NDP-M). In reality, the CPU Rd/Wr queue can consist of several queues, for example a separate one for prefetch loads. For simplicity, only the forward path towards the DRAM is shown. (b) The main components in the near-data processor access point (NDP-AP) and its connections to the system bus.

- Offering an interface for any type of NDP;
- Managing coherence and consistency between the CPU and the NDP, the former by means of a distributed directory stored in DRAM, discussed in Section 5, and represented by the *Coherence Manager* in Figure 3(a);
- Translating the virtual addresses received from the NDP, discussed in Section 6, and represented by the *ERAT* (effective-to-real translation) in Figure 3(a);
- Providing a gateway to access non-local data, discussed in Section 6.3, illustrated by the *NDP Remote Rd/Wr Q.* in Figure 3(a);
- Providing information towards the NDP-AP (described below) about address ranges involved with near-data processing.

NDP accesses are, after translation, stored in either the local or the remote queue. CPU accesses are stored in the CPU queue. An arbiter decides, based on their priorities and the right address protection (ordering) rules, which queue can issue a memory access. The coherence manager makes sure the access is performed coherently, after which it is dispatched to the memory controllers.

4.3 Near-data Processor Access Point

The CPU needs to support the NDP-M in all its needs regarding address translations, remote accesses, and so on. To avoid a significant software overhead and unnecessary CPU load, we introduce a single hardware component attached to the system bus, called *near-data processor access point* (NDP-AP). From the NDP-M’s perspective, the NDP-AP is the access point into the global coherent memory space. From the software perspective (either OS or user), it is the component handling communication with the NDP-Ms. In Figure 3(b), we show the architecture of the NDP-AP, with its main functional blocks. The NDP-AP is responsible for:

- Issuing load and stores on remote data on behalf of the NDP-M(s);
- Supporting virtual memory and coherence functionality towards the NDP-M(s), without affecting the CPU performance by means of scope filtering based on information provided by the NDP-M(s);
- Caching of remote accessed data to exploit locality in various communication patterns.

Applications, or the OS, can send messages to the NDP-AP by means of a special instruction (like *icswx* for the POWER ISA [17]), which forces a cache line on the system bus directly addressed towards the NDP-AP. This gives us the possibility to dispatch messages with very little latency and overhead. The NDP-AP can send messages to the NDP-M by issuing a special transaction on the system bus, directly addressed towards the correct generic MC, with the reverse path working likewise.

When designing a balanced system, the NDP-AP throughput must match or exceed the accumulated data and coherence throughput of the NDP-M enabled memory channels. When scaling up the system size, the NDP-AP concept has two degrees of freedom. First, an NDP-AP can have multiple connections to the system bus, and second, additional NDP-APs can be integrated at the CPU, each supporting a bandwidth-matched subset of NDP-Ms. The best solution depends on the bandwidth of the system bus connection, the system bus organization, technology-specific features (e.g., delay), and cost. In this work, we consider the NDP-AP to be a single component.

4.4 The System Bus

Access to the system bus is necessary to communicate between the NDP-AP and the NDP-Ms. We base our work on a system bus as found in modern CPUs such as the POWER8. Such a bus is a capable interconnect fabric supporting the integration features described in Section 2. POWER8 features several wide bidirectional buses, with up to 3.7TB/s of bandwidth between all components. This is much more than the accumulated memory channel bandwidth of 230GB/s [44, 47, 50]. Note that the system bus, for any system design point, by definition, can support the accumulated memory channel bandwidth, making this a perfectly scalable architecture. The system bus has two snoop buses and thus supports two coherence requests per clock cycle from all attached components combined. Given the bus frequency of 2GHz and the 128B cache lines, this results in a snoop and invalidate throughput of 512GB/s [47, 50].

5 MEMORY CONSISTENCY AND COHERENCE

In this section, we discuss both consistency as well as coherence between CPU and NDP. No assumptions on the NDP design are made, and they can have hardware managed caches or not. Most attention will, however, go to NDPs with hardware managed caches, as it shows the more complete picture.

5.1 NDP-M Memory Interfaces and Consistency

The NDP-M offers memory interfaces to let an NDP access the memory. The number of available interfaces and their width depend on the memory technology used. The NDP-M does not apply any bandwidth optimization techniques (grouping, etc.) to the received memory request, as it assumes the NDP already makes use of the memory interfaces in an optimal way. Nonetheless, the NDP-M implements an unordered memory model, to be able to hide latency for address-translation misses or remote memory accesses. The *Rd/Wr queues* shown in Figure 3(a) allow for out-of-order issuing of memory operations, where the *Address Protection* unit also shown in Figure 3(a) ensures ordering rules.

The NDP can be considered as a thread in the same shared memory as the CPU, and therefore a memory-consistency model is necessary. The recent POWER multicore CPUs implement an unordered memory model [46], relying on (*lightweight*) *sync* instructions to enforce consistency between concurrent threads. After a *sync*, all memory accesses are committed to the global coherent space, and it is ensured that every thread will observe the right value. A *sync*, or likewise *barrier* instruction, executed by the NDP will have to ensure the same. In case the NDP has a hardware managed cache, this means all loads and stores have to be committed to the cache, and the NDP-M

Fig. 4. Two activity diagrams showing possible coherence interactions between the NDP and the CPU. In (a) the NDP wants to write to a line currently owned by the CPU, and in (b), the CPU wants to read a line currently owned by the NDP.

will ensure the CPU observes this value as described in the following sections. In case the NDP does not have hardware managed caches, a *barrier* has to travel to the *NDP Rd/Wr queue* shown in Figure 3(a) and ensure that this queue is empty before returning. Only after the queue is empty, have all accesses passed the *Coherence Manager* and entered the global coherent space, meaning that the CPU is able to see the latest value.

5.2 Extended Coherence between CPU and NDP

We introduce a hardware managed *Extended Coherence* solution to enforce coherence between the CPU, the NDP, and the main memory in a generic way. This is illustrated on a functional level in Figure 2(b), showing the coherent views of both devices towards each other and the main memory. From the devices' (CPU and NDP) perspective towards the NDP-M, we implement a basic MSI (modified-shared-invalid) protocol [49]. The devices can send messages like "Get for read" (*GetS*), "Get for write" (*GetM*), "Upgrade for write" (*Invalidate*), and so on, to the NDP-M. As MSI is a subset of the more complex polling-based coherence protocol found in contemporary CPUs, this **does not** require changes in the CPU protocol. The *GetS* or *GetM* messages from the CPU travel down the 'normal' path to the NDP-M, as these are just default loads from the CPU. We discuss *Invalidate* messages separately in Section 5.4.

At the NDP-M, the state of the memory is managed by the *Coherence Manager*, and every line has either the *CPU owned* state, the *NDP owned* state, or the *shared* state. The *shared* state is introduced to allow effective processing with shared data sets, such as for example reading data of one NDP by another NDP, without having to invalidate the data on the originating NDP. How the NDP-M manages these states is discussed in Section 5.3. We do not make a distinction between cached and non-cached (i.e., in main memory) lines: lines in an *owned* state will not leave that state until the other device takes action. This enables us to minimize coherence state changes, making it as transparent as possible. If we had introduced a *home* state (the line is only valid in the main memory), then both the CPU as well as the NDP would be claiming and returning ownership all the time.

In Figure 4, we show two possible coherence interactions between the CPU and the NDP. Although not exhaustive, this figure shows the interaction between the various components in the architecture and illustrates that the proposed mechanism can seamlessly integrate coherent NDP usage with existing CPU coherence. Figure 4(a) shows how the NDP can claim a cache line to modify it. The request from the NDP enters the NDP-M, which checks the state of the memory and finds it as *CPU owned*, leading to a *GetM* message towards the NDP-AP. In Figure 4(b) we show

how the CPU can get shared access to a cache line currently owned by the NDP. In both figures it is clearly visible how the NDP-M and NDP-AP together orchestrate coherence interactions between the system bus (the CPU) and the NDP. In these figures, the NDP-AP communicates with the system bus. This is the bus to which every component and device in the system is connected, for example, all the CPU cores, the (semi-)shared last-level caches, the memory controllers, other CPUs, or even FPGAs using a coherent external link such as CAPI [51]. Therefore, the NDP-AP is a completely generic interface into the entire system, and the system bus *guarantees* that an *Invalidate* message, for example, will invalidate the requested line in the entire system, and will be acknowledged back to the NDP-AP eventually.

5.3 Implementation of the NDP-M Coherence Manager

At the NDP-M, a combined fine- and coarse-grained distributed directory method is used to keep track of the coherence state of the memory. The *Coherence Manager* stores the state of the complete memory attached to the respective memory channel in an *Extended Coherence Directory* (ECD). When a coherence request is received, the directory is accessed to check and/or change the state of a line. To avoid having to access the memory for every state lookup, which would basically double the number of memory accesses, an *Extended Coherence Directory Cache* (ECD-\$ in Figure 3(a)) is introduced, holding the most recently accessed directory items. Directory items are loaded by means of the dedicated Rd/Wr queue shown in Figure 3(a), which has priority over the other queues.

Given that our architecture targets large memories, the size of the directory, even if the three discussed states are stored in two bits, becomes quite large, while relatively occupying only 0.2% of the main memory. If workloads access memory with very little locality, then the introduced cache will suffer from misses, which is especially not acceptable for CPU traffic. Therefore, we introduce a technique similar to the one used to minimize snooping traffic between different processors in a SMP environment [50]. A tiny *Extended Coherence Directory Summary* (ECD-S in Figure 3(a)) is introduced to store the coherence state for every, for example, 16 megabytes of memory, given that the entire 16 megabytes have the same coherence state. This requires a storage capacity in the kilobyte range. The summary entries are initialized to the *CPU owned* state. The NDPs are “transparent” to the CPU and there is no penalty in memory bandwidth or latency. Only in cases where the coherence state of a single cache-lines in the memory block varies and the ECD-S item is *Undefined*, the full directory is required through the ECD-\$. The management of the ECD-S is based on existing technology as found in the POWER8 CPU for managing coherence snoop ranges between CPUs [50]. Summary items can either be changed and reconstructed in a transparent, hardware managed way, or by claims issued by an NDP-runtime or the code generated by a compiler (e.g., combined with *offload* pragmas as found in OpenMP 4+). The software methods enable optimized coherence interactions in the system for well understood workloads, without losing correctness.

5.4 Extended Invalidate and Exclusiveness

As discussed in Section 2, we cannot let every *Invalidate* message from the CPU travel to the NDP-M *Coherence Manager*, as it would hurt CPU performance. To solve this, the *ECD-S shadow* is introduced at the NDP-AP, as shown in Figure 3(b). When an ECD-S item is either *Shared* or *Undefined*, a CPU *Invalidate* will be forwarded by the NDP-AP to the NDP-M. In case an ECD-S item is *CPU owned*, no extra action is required, and a CPU *Invalidate* can have its scope reduced to the CPU alone, and the NDP-AP filters and acknowledges this message.

Care is necessary to handle the *Exclusive* coherence state at the CPU [49]. When a cache requests a cache line to read, and the memory controller is the only component acknowledging the snoop,

the cache gets the line in *Exclusive* state, meaning it can upgrade the line for write access without having to issue an *Invalidate*. To avoid CPU caches upgrading cache lines without the NDP-M knowing, the NDP-AP, by means of the ECD-S shadow, has to acknowledge a snoop for a line currently in the *Shared* or *Undefined* state, letting the CPU know the NDP also has access to the line, and thus avoiding *Exclusive* access.

Both the *Invalidate* as well as the *Exclusiveness* handling do not introduce a penalty for CPU accesses to NDP-unrelated memory regions.

5.5 Coherence Scalability

For the CPU it has been established that only the coherence actions strictly necessary are forwarded to the NDP-M. From the NDPs' perspective we consider two scalability concerns. The first consideration is the number of actions that need to travel from the NDP-M towards the NDP-AP. When the coherence state returned by the ECD lookup at the NDP-M corresponds to the state required by the NDP, the scope of the coherence request can stay local at the NDP-M, and no interaction between the NDP-M and the NDP-AP is required. This holds, when the application shows basic compute-data affinity, for the great majority of memory accesses, just as would be the case in a multi-core or multi-CPU environment. This is reflected in Section 9.2.2, where it is shown that coherence messages only take up 2% of the NDP-M-NDP-AP traffic, when considering a graph traversal workload. When the NDP wants to write to a cache line not already *NDP owned*, or read a line currently *CPU owned*, an *Invalidate* message or *GetS* message has to be sent to the NDP-AP, respectively. These scenarios will occur when the NDP starts working, and when there is read-write interaction with the CPU for a data set. In Section 8 we will show that this start-up performance penalty is small, or can be completely hidden if there is enough memory access parallelism. The second consideration is whether the NDP-AP is a bottleneck in handling all NDP coherence messages. As is shown above, this question is only relevant when looking at the extreme cases. As discussed in Section 4.4, the system bus we consider in this work supports two coherence requests per clock cycle, and implementing a single NDP-AP component able to fully utilize this is trivial. These bus specifications mean that, when the NDPs need to invalidate data at a rate higher than 512GB/s, the system bus is a bottleneck. Note that this holds for *any* method of invalidating the data and is unrelated to the NDP-AP based mechanism: also letting the maximum number of software threads invalidate data at their full capacity results in a 512GB/s peak invalidation rate. This is confirmed by experiments on a 10-core POWER8 CPU, showing a peak software-driven invalidate throughput of around 420GB/s. The invalidate throughput is therefore a bottleneck unrelated to our architectural proposal, as it would occur in any NDP or coprocessor proposal that takes coherence into account. In case the number of snoop buses at the CPU increases, multiple NDP-APs could be used to make full use of them.

5.6 NDPs Without Hardware Managed Caches

In this work we focus on supporting arbitrary near-data processing, and not all NDPs need to have hardware managed caches. In fact, when considering for example workload-optimized NDPs implemented on reconfigurable fabric, it is far more likely that the NDP uses an application specific local memory, and handles coherence manually. In this case, the NDP-M receives loads and stores from the NDP, instead of *GetS*, and so on. When receiving a load from the NDP, the NDP-M *coherence manager* still has to check whether the corresponding cache line has to come from the CPU or the main memory. When receiving a store from the NDP, the NDP-M *coherence manager* still has to invalidate the line at the CPU. As explained in Section 5.1, the NDP accesses now enter the global coherent space at the point of the NDP-M *coherence manager*.

6 VIRTUAL MEMORY MANAGEMENT AND ACCESSING REMOTE DATA

6.1 Address Translation Implementation at the NDP-M

In the proposed architecture, the NDPs work with virtual addresses, and all address translation mechanisms are implemented at the NDP-M. The NDP-M holds ERAT content addressable memories (CAM), and a translation lookaside buffer (TLB) (see Figure 3(a)). TLB hit rates have been identified as a problem for big-data workloads [7]. However, when considering a 2013 big-data optimized CPU core [48], using huge pages in a transparent way [42], TLB reaches of 32GB directly translatable memory are possible. We consider 32GB or 64GB a realistic size for NDP local memory. The NDPs are able to access remote memories as well, but this will be limited to, when the system is used correctly, a few data sets, such as for instance *boundary values* when running a distributed grid-based solver, thus not increasing the physical address range significantly. Therefore, realizing a modern, big-data optimized, NDP-M based address translation mechanism is not a particular challenge, and is not further discussed in this work.

6.2 Extended Virtual Memory Management and TLB Synchronization

To populate the TLB, and to keep the TLB synchronized with the rest of the system (supporting TLB shutdowns, etc.), the NDP-M must be connected to the relevant fabric at the system bus. As described in Section 2, this cannot be done in a naive way, as it would impact the CPU performance in a very negative way. Therefore, as with coherence, the NDP-M works in close collaboration with the NDP-AP to enable TLB synchronization without overhead. In case of a (demand) miss in the NDP-M TLB, the miss is forwarded to the NDP-AP, which has its own page-walker (see Figure 3(b)), to be able to serve the request without having to ask the OS for help. As described in Section 3, a page being accessed for the first time will automatically be allocated in the local memory of the requesting NDP(-M). To limit the amount of TLB synchronization traffic that needs to be forwarded to the NDP-M, we use the *ECD-S Shadow* at the NDP-AP to filter the TLB synchronization commands. Only commands relevant for data ranges marked either *NDP owned*, *Shared*, or *Undefined* are forwarded from the NDP-AP towards the NDP-M. These are the data ranges for which the CPU knows that, or cannot be sure whether, the NDP is working with them. TLB commands related to data ranges marked *CPU owned*, thus all non-NDP related memory (or NDP-related data owned by the CPU at that moment), are reflected directly by the NDP-AP, incurring no penalty.

6.3 Accessing Remote Data

Accesses to remote data are issued to the NDP-M in the same way as local accesses. The NDP-M will, after the address translation, recognize the resulting real address as being non-local. The request is forwarded to the NDP-AP, which issues the request into the global coherent address space. Once the data has arrived from, for example, a remote memory channel, it is placed in the data cache of the NDP-AP, and returned to the NDP-M. This mechanism does not limit remote accesses to being between NDPs. Since the NDP-AP does the request in the global coherent space, NDPs can also access data in another CPU socket, or perhaps even data on a GPU being connected in a virtualized and coherent way.

The NDP-AP can only access data with the full 128-byte cache line size granularity of the system, meaning that every access from the NDP will result in a 128-byte line stored in the NDP-AP's cache. Therefore, in case the NDPs have a smaller access granularity than the CPU (e.g., 32 B), consecutive accesses to the same line can directly be served by the NDP-AP, and do not have to go to a distant memory. The cacheability of remote data at the CPU also implies that NDPs can exploit mutual locality for various access patterns, like one-to-all. This is shown in Figure 5(a), where the remote

Fig. 5. Two activity diagrams showing the process of a remote load (a) and a remote store (b).

read from the second NDP has a hit in the NDP-AP cache. This property will be a key feature when exploring the Graph500 benchmark in Section 9.

6.3.1 Cacheability of Remote Data. As highlighted in Section 2.2.1, the snooping range of the CPU cannot be extended to the NDPs. Therefore, NDPs with hardware-managed caches holding lines of remote data are a problem, as the lines would be invisible for the system. The snooping traffic could be extended in an optimized way as done with the *Invalidation* messages described in Section 5.4, but that would still mean that a snoop has to travel to *every* NDP, potentially wasting bandwidth and energy. Another option can be to allow a little bit of cached remote data by introducing a shadow data cache directory at the CPU side [51]. This has as downsides that (1) it does not resolve the issue when considering large data sets, and (2) it implies that the NDP has to access remote data at the granularity of a CPU cache line, which is typically larger than the NDP access granularity, wasting valuable interconnect bandwidth.

Therefore, the proposed architecture does not allow NDPs to cache remote data, which is enforced by the NDP-M, marking a line *inhibited* when provided to the NDP. This is not as strict as the inhibited flag found in the POWER ISA [29], where it implies, for instance when writing to memory mapped device registers, that an access has to be performed by a special cache-inhibited data path. In the proposed architecture, marking a line inhibited means that the line is evicted when the access is completed, or that a write through and eviction are performed. Note that, in case a remote access hits in the NDP-AP data cache, the latency is not much longer than accessing local memory. Thus, as long as high hit rates in the NDP-AP data cache are realized, remote accesses are not heavily penalized. To further alleviate the inconvenience of cache-inhibited remote data, we introduce the concept of user/compiler enhanced coherence. By manually setting a data set as “remote cache uninhibited,” the NDP-M will allow the caching of this data in a remote cache. This comes at the expense of having to keep the system coherent manually, typically by flushing the NDP caches at certain points in the application.

In Figure 5 we show the interactions involved in doing a coherent remote load and a coherent remote store. In case of a remote *GetS*, the NDP-M forwards it to the NDP-AP, which asks for the line on the system bus, puts it in its cache, and sends the correct sub-cacheline to the appropriate NDP-M. In case of a remote *GetM*, extra care is necessary. An NDP cache cannot hold ownership of a remote cache line. Therefore, the NDP-AP locks the line on behalf of the NDP, and unlocks it as soon as the NDP returns it, also shown in Figure 5(b).

As discussed in Section 5.6, NDPs do not have to have hardware managed caches, and for many designs this will likely be the case. When an NDP does not have hardware managed caches, remote data accesses are treated exactly the same as local accesses, and the discussion above does not apply.

6.3.2 Scaling Remote Accesses and Inter-NDP Traffic. All inter-NDP traffic is sent over the CPU's memory channels, the system bus, and through the NDP-AP. In Section 4.4 it is established that the system bus is not a bottleneck. As discussed in Section 4.3, the number of system bus connections of the NDP-AP can scale, as well as the number of NDP-APs. However, when considering the next-generation POWER CPU, a single system bus connection offers already more bandwidth than all memory channels combined [30]. This shows that a single NDP-AP implementation can already serve all inter-NDP traffic.

7 SYSTEM SIMULATOR AND IMPLEMENTATION

To validate the architecture, we implemented the new components in software and integrated them in a custom system simulator. The simulator was then used to obtain performance numbers for both synthetic benchmarks and a representative graph-processing application. Applications can make use of a user-level software library, providing all the functionalities to allocate data structures, communicate with the NDPs, support synchronization between the CPU and the NDPs, and so on. Below we highlight several aspects of the system simulator:

- The NDP-M and NDP-AP and their subsystems are implemented in software as shown in Figures 3(a) and 3(b) and as described in this work;
- A variety of other components (memories, memory channels, system bus, caches, etc.) are implemented in software;
- The system simulator tracks individual loads, stores, atomics, barriers, and so on, through the entire system following the coherence and remote data access characteristics described in this work;
- The system simulator can track the data associated with every load and store, thereby “mirroring” the memory of the host system and being able to check the correctness of the simulator and thus actually “executing” the workload in the simulator;
- The system simulator supports general-purpose CPU and NDP cores running arbitrary applications, which get their loads, stores and operations from a front end based on Intel PIN [38];
- The system simulator supports workload-optimized NDPs developed in a hardware description language (VHDL/Verilog), based on the Verilog Procedural Interface (VPI). This functionality is not used in this work.

The system simulator developed focuses on the discussed aspects of the proposed architecture and is able to provide detailed performance information. The tracking of coherence states and even the data field for every load and store are different from existing performance estimators [45], which typically work at a coarser level.

Table 1 summarizes some of the most important system parameters. The characteristics are based the POWER8 CPU [47, 50]. We consider four DDR4-2600 memory controllers per memory channel, running at a realistic 60% utilization. Memory (controller) latency is also based on POWER8 characteristics. The best case round trip latency for an NDP memory access, based on the numbers provided in Table 1 and others like core-to-NDP-M interconnect, is around 60ns, a 25% improvement over the CPUs access latency. In case of a coherence-directory cache miss or a remote access, this latency is obviously much larger, about double for the former, and about three

Table 1. System Specifications

CPU	
Access granularity	128 byte
Memory channels	8×, 20:10GB/s Upstream:Downstream
Main memory (per channel)	
Technology and bandwidth	4× DDR4-2400 @ 60%: 48GB/s
Memory controller + DRAM latency	40ns
Access granularity	32 byte
General-purpose NDP	
Cores	64 cores, 1GHz, in-order
Data caches	256 byte private per core
Main-memory round-trip latency	60ns (best case)
NDP-M	
ECD-\$ size	variable, 32B lines
ECD-\$ / ECD-S / ERAT latency	4/2/2ns
NDP-M - NDP-AP latency (single trip)	24ns
NDP-AP	
Data cache size	variable
System-bus data interface	128:64GB/s In:Out
System bus	
Frequency	2GHz
Snoop buses	2

to four times as high for the latter. The size of the coherency-directory cache at the NDP-M and size of the data-cache at the NDP-AP are set to an appropriate size for the two types of experiments performed. The NDP-AP bandwidth from and to the system-bus is almost high enough to saturate all the memory channels.

As NDP design point we use 64 slow, 1GHz, in-order cores, with instruction latencies similar to those found in the processor described in Reference [45]. The cache size of the general-purpose NDPs is only 256 bytes. This allows the exploitation of some data locality but also ensures that the observed performance characteristics come from the DRAM and inter-NDP bandwidths, and not from the cacheability of the data sets.

8 SYNTHETIC BENCHMARKS

In this section, we discuss several synthetic benchmarks related to the bandwidths achievable in the proposed architecture and the parameters from Table 1. We explore both streaming and random access patterns for accessing local memory, as well as various communication patterns using multiple NDPs. When using random access patterns, 16GB data sets per memory channel/NDP are used (128GB total). Since these data set sizes are on the order of what one can expect in a real system, the ECD-\$ and NDP-AP data cache sizes are set to values one would find in a modern CPU [50]: 64KB and 512KB, respectively.

8.1 Local Accesses

First we explore the achievable bandwidths on a single NDP, while varying the coherence properties. Figure 6(a) shows the achieved bandwidth when doing a streaming read, while varying the initial coherence state of the data. The ECD-S is not used in this experiment. When the data is

Fig. 6. Synthetic benchmark results. (a) Shows achieved DRAM bandwidth and memory channel bandwidth (coherence messages) for local streaming reads, while varying the initial coherence state. (b) Shows achieved DRAM bandwidth for local random reads, while varying the percentage of data valid in the ECD-S. (c, d) Show memory channel bandwidths and NDP-AP data cache hit rates for various inter-NDP communication patterns.

Table 2. Common Communication Patterns and How They Are Implemented on the Proposed Architecture

MPI Operation	Description	Software implementation on the proposed architecture
Send	Send data one to one	Write one to one
Receive	Receive data one from one	Read one from one
Broadcast	Root to all, same data	All NDPs read from root
Scatter	Root to all, different data	All NDPS read from root
Gather/Reduce	All to root, data ordered/reduced	Root reads from all NDPs
All-gather/All-reduce	All to all, data ordered/reduced	All NDPs read from all NDPs

already in the *Shared* state, a 29GB/s DRAM bandwidth is achieved. When the data is in the *CPU* state, coherence rights have to be claimed from the NDP-AP, resulting in a slightly lower DRAM bandwidth of 24GB/s, and a 2.25GB/s up- and downstream bandwidth. For both cases, the bandwidth required to load *ECD* items from DRAM is negligible, as the accesses have a high degree of spatial locality. Also included in Figure 6(a) are the bandwidths when including software prefetching. This shows that the penalty of a coherence round trip to the NDP-AP is moderate, or can even be completely hidden.

Figure 6(b) shows the achieved bandwidth when doing random reads, while varying the portion of the data set with a valid *ECD-S* item, and the coherence state all data is *Shared*. When the portion with a valid summary item shrinks, more directory items have to be fetched from DRAM, and given the completely random nature of the accesses, this doubles the number of memory accesses in the worst case, and halves performance.

8.2 Remote Accesses Using Various Communication Patterns

Table 2 shows the most common MPI communication patterns and how they are implemented in software on the proposed architecture. First, we explore one to one communication, followed by many-NDP communication. In Figure 6(c) the performance for various one to one communication patterns is shown, using again streaming and random access patterns. A streaming read is bounded by the downstream bandwidth of the receiving NDP at 10GB/s, with an NDP-AP data cache hit rate of 75%. Similarly, streaming writes are bounded by the downstream of the receiving NDP at

Fig. 7. (a) Using an NDP to copy a data set, of which 8 MB is stored in the caches of the CPU. The initial coherence state is *CPU owned*. (b) Accumulated streaming read performance using multiple NDPs, while varying the number of snoop buses at the CPU's system bus. (c) Accumulated copy performance using multiple NDPs, while varying the amount of snoop buses at the CPU's system bus.

10GB/s, with an NDP-AP data cache hit rate of 87%. This is higher than with streaming reads, because both the read retrieving the data from the NDP-AP as well as the write delivering it are counted as hits. Random reads are bounded by the upstream bandwidth of the providing NDP at 20GB/s, thus realizing a 5GB/s bandwidth towards the receiving NDP (32B out of 128B). The NDP-AP data cache hit rate is almost zero as expected for such a large data set. Random writes are bounded by the downstream bandwidth of the receiving NDP at 10GB/s, thus realizing a 2.5GB/s bandwidth for the issuing NDP. The NDP-AP data cache hit rate is 50%, as every read misses but every write hits, as discussed in Section 6.3.

In Figure 6(d) the performance for various many-node communication patterns is shown, for which we use eight NDPs (root + seven). Broadcast results in an aggregate bandwidth of 70GB/s, bounded by the downstream bandwidth of the receiving nodes. Since every node needs the same data, the NDP-AP data cache hit rate is very high, and the upstream bandwidth of the root is not a bottleneck. Scatter shows an aggregate bandwidth of 20GB/s, bounded by the upstream bandwidth of the root, since now every receiving node needs different data. Gather is bounded by the downstream bandwidth of the root, and shows an aggregate bandwidth of 10GB/s. All-gather, where every NDP reads from another NDP, shows a 64GB/s aggregate bandwidth, bounded by the upstream bandwidth of the NDP-AP.

8.3 Copying a Data Set

In Figure 7(a) we show the performance for the essential operation of copying data, using a single NDP and a single memory channel. The source and destination addresses are both in the local memory of the NDP. When starting, all data has the coherence state *CPU owned*, and the first 8 MB of the data set is stored in a cache of the CPU. It can be observed that for small data sets, the realized copy performance is very low. This can be explained by the fact that the 8 MB stored at the CPU needs to be written back to the NDP-M over the “slow” 10GB/s downstream link. For larger data sets this becomes a negligible portion, and the memory channel is foremost used for coherence interactions between the NDP-M and the NDP-AP. In the case of using a single NDP and the largest data set, the NDP-AP has to process one coherence messages roughly every four clock cycles. The copy performance of the CPU would be 10GB/s, bounded by its downstream bandwidth. This shows that, although copying data seems a natural fit for NDPs, it only pays

off for large data sets when one includes the, for natural applications inevitable, writeback and coherence overheads.

8.4 Coherence Bottlenecks

In Figure 7(b) we show the streaming read performance, while varying the number of snoop buses available at the CPU. The initial coherence state of the data is *CPU owned*. It can be observed that, when considering a limited number of snoop buses and many NDPs, the read performance of the NDPs is bounded by the *invalidate* capacity of the system bus. A single snoop bus can invalidate 256GB/s, and thus the NDPs cannot read data faster, regardless of their local memory bandwidth. Figure 7(c) shows the same experiment when considering the copying of a data set. Since copying data requires two reads and one write to local memory, but only two coherence claims, the system bus limitations are less profound. Both experiments show that for the default design point of having two snoop buses in the system bus [47, 50], no scalability problems become apparent, since the system bus can invalidate data at 512GB/s while the NDPs have a peak accumulated local memory bandwidth of 384GB/s.

8.5 Synthetic Benchmarks Insights

The synthetic benchmarks offer two insights. First, the architectural proposal, implemented with the design point of Table 1, shows excellent performance for local memory accesses and various communication patterns. Second, the simple experiments represent an important range of operations discussed to be suitable for near-data processing [37]: copy, search, indexed lookups, very wide vector operations, and so on. Our results show that a single such operation incurs a significant penalty in terms of coherence traffic and data writeback and possibly performance. This means that, for example, searching for a key in a data set should be followed by many more key searches to offset the initial cost. For other workloads, like copying, this is much harder, as the data is likely made dirty by the CPU and possibly still (partly) in its caches.

This argues in favor of using NDPs to run workloads for a prolonged period of time, touching the data many times, using large data sets. One example of such a workload is doing many operations on a large graph data structure, as is discussed in the case study below.

9 CASE STUDY: GRAPH500

Graph500 [22] is a well-known data-intensive benchmark and a typical near-data processing problem [23], doing several breadth-first searches (BFS) in a graph, representing workloads like big-data analytics. In several steps, called *levels*, it explores the entire graph, from a given starting node to all its neighbors, and in the next level from all its neighbors to their neighbors and so on. The set of nodes visited in the previous level is called the *frontier* of the current level. The size of the problem is denoted as scale n , meaning the graph contains 2^n nodes. A detailed explanation of distributed Graph500 can be found in Reference [10].

As we are, in contrast to the synthetic benchmarks, unable to run this complex workload with data sets in the 100+ GB range due to unfeasibly long runtimes, the ECD-\$ and NDP-AP data cache sizes are set to smaller values: 8KB and 64KB, respectively. This way, we assume that our results for giga-scale simulations are relevant and can be scaled to tera-scale problems. We used graphs with up to 64 million nodes and one billion edges (scale 26), which consume 22GB of memory when using the maximum of eight distributed processes.

9.1 Implementation

To evaluate Graph500, we use a system organization where every memory channel has an NDP consisting of 64 general-purpose NDP-cores (see Table 1). We use the MPI-parallel *bfs_replicated*

Fig. 8. (a) Performance for Graph500 running at our NDP architecture, with an IBM POWER8 and Intel Xeon CPU serving as an illustrative reference. (b) Various system bandwidths when running a scale 24 problem.

Graph500 reference code [22] to generate the distributed graph data sets. The MPI + OpenMP parallelization is replaced by a nested OpenMP parallelization, where we first spawn a thread per NDP, followed by many threads for every NDP-core. The BFS code is extended to use the “direction optimization” to switch between bottom-up and top-down frontier expansions [8]. In the levels explored top-down (the first levels and the last levels), every node in the frontier visits all its neighbors and sets their parent. The levels explored bottom-up (the middle levels), loop over all remaining nodes and try to find a valid parent in the current frontier. Various data accesses, for example, a frontier check in the bottom-up phase, can be remote.

9.2 Results and Discussion

The performance results of our Graph500 implementation are shown in Figure 8(a). As a reference we use the performance of the Graph500 reference code running on a real POWER8 CPU with 10 cores at 3.5GHz, 230GB/s memory bandwidth, and 85MB cache. For completeness, also results obtained on an Intel E5 2683v3 (14 cores, 28 threads, 2GHz) server CPU are shown. When looking at a single NDP, we see around two tera traversed edges per second of realized performance. The performance of multiple-NDP configurations steadily increases towards larger problems and shows no sign of dropping for the sizes of the problems we explored. Clearly, a lot of work is needed to utilize all the available parallelism (512 threads in total) and to offset the low-thread-count regions at the start and the end of the execution, inherent to the application. Scaling to multiple NDPs is not perfect, since we now have to perform remote accesses that have a (in case of a NDP-AP cache miss) much higher latency than a local access. When using two NDPs, the percentage of remote accesses is 10%, growing to 20% for eight NDPs. Due to this penalty, we see a strong performance scaling of a factor 1.5× and 1.7× when doubling the NDPs. Weak performance scaling is better, showing factors between 1.6× and 1.9×.

We expect the performance to increase further for larger problems, and not to drop due to caching effects, based on the trend lines shown in Figure 8(a) and the analysis below. For a scale 26 problem, the *visited* and *out_queue* bitmaps, essential data sets keeping track of the state of every vertex with a single bit and both accessed in a “random” way, are 8MB in size, or 512 times larger than the combined cache size of 64 NDP-cores (64 times larger than all caches in eight NDPs). This implies that our results do not depend on the cacheability of essential data sets but on the access characteristics of our architecture, foremost the low latency and 32B access granularity.

Fig. 9. (a) The impact of the NDP-AP data cache size on hitrate and performance running a scale 24 problem, using four NDPs. (b, c) Performance of the coherence methods, with and without the use of the *ECD-S*.

In Figure 8(b) we show the measured bandwidth for various aspects of the architecture. The bandwidth into the NDP-AP (the responses to reads issued by the NDP-AP) grows with the number of NDPs, but does not reach its limit yet. The bandwidth going out of the NDP-AP (the responses towards the various NDP-Ms) is a factor four lower, because the reads done by the NDP-AP are at the CPU's 128B granularity, while the responses towards the NDP-M are at a 32B granularity. The main memory bandwidth slowly decreases when using more NDPs, driven by the fact that the performance achieved per NDP is lower due to a smaller problem size per NDP and a larger portion of remote accesses. The memory channel bandwidths are stable between system configurations, balancing out the relative longer runtime and the larger number of remote accesses. The memory channels show the same 4:1 bandwidth ratio for upstream and downstream as the NDP-AP.

9.2.1 NDP-AP Data Cache Performance. Accesses to remote data are resolved by means of the NDP-AP, as described in Section 6.3. When a remote access hits the NDP-AP data cache, the latency is reduced. Furthermore, a cache hit eliminates the need for data to be accessed in a DRAM and transported to the NDP-AP. In Figure 9(a), we show the reduction in runtime, as well as the reduction in data transports achieved when we increase the NDP-AP data cache size, when running a scale 24 problem with four NDPs. For the cache size of 32KB, we have a hit percentage of less than 2%, due to the limited capacity as well as the fact that for Graph500 the remote accesses have little spatial locality. When increasing the cache capacity to 512KB, the hit rate is 26%, increasing the application performance by 12% and reducing the amount of data traveling from the various DRAMs to the NDP-AP by 23%. As illustrative example, in case the NDP-AP data cache were the size of the entire L3 cache of the CPU (80MB), the hit rate becomes 97%. The application performance increases by 33%, clearly indicating that the latency for remote accesses is an important metric.

9.2.2 Impact of the Coherence Methods. An important aspect of the proposed architecture is the handling of coherence between the NDPs and the CPU, as discussed in Section 5. The NDP-M checks the coherence state of every access from the NDP or the CPU in the ECD via the ECD-\$. To limit the ECD-\$ accesses, the ECD-S is used as discussed in Section 5.3. When starting the benchmark, we set the entire graph structure, which is read-only after initialization, to *Shared* in the ECD-S, meaning accesses to this data set do not have to access the ECD-\$. In Figure 9(b), we show the performance of the ECD-\$ for increasing problem sizes. When using the ECD-S as described above, the number of ECD-\$ misses is very limited, with a maximum of 17% for the

largest problem. This means almost all accesses are made coherent in either the ECD-S latency or the ECD-\$ latency. When not using the ECD-S, also shown in Figure 9(b), the number of ECD-\$ accesses increases by 35%, and the ECD-\$ misses accordingly.

In case of an ECD-\$ miss, the item has to be loaded from DRAM. In Figure 9(c), we show a breakdown of the DRAM bandwidth with respect to the component doing the access. When using the ECD-S, a very small portion of the bandwidth is used for loading and storing directory items. Without the ECD-S, this percentage increases to over 20% of the bandwidth, introducing a significant overhead in time and energy. In Figure 8(b), we show the bandwidths for the memory channels. Roughly 2% of this bandwidth is spent on coherence traffic between the NDP-M and the NDP-AP. When using only one NDP, we observe around 0.01GB/s of coherence traffic between the NDP-M and NDP-AP to claim all data previously owned by the CPU.

These results show that, with a good use of the proposed coherence mechanisms, the overhead is very small, even with a very small ECD-\$. The ECD-S increases the application performance by 10% for large problems, and reduces the number of accesses to the ECD-\$ and DRAM significantly.

9.3 Power Analysis

We used McPat 1.0 [36] to evaluate the power consumption of the general-purpose NDP. When using low standby power (LSTP) 22nm technology, the power of the entire 64 core NDP is 5.9 W. The power usage for the NDP-AP is estimated at 4W, based on comparable bus-attached components discussed in Reference [33]. The NDP-M works at a much lower frequency, and we estimate a power of 2 W, giving the entire memory side (memory-side chip + DRAMs) a 20% increase in power [31]. This results in a power budget of the entire proposal of 67 W. Given that the CPU is practically idle when running workloads on the NDPs, the CPU cores go to sleep mode, for a power saving of 100W [20, 31]. This delivers a net power saving of 33W for our proposal. Given the much shorter runtime, a significant energy-to-solution saving will be realized, using about half of the energy compared to the reference, depending on the exact system configuration and problem.

10 CONCLUSION

In this work, we introduced an architecture for a seamless integration of arbitrary near-data processing capabilities in an existing server environment. By introducing a component in the memory system as well as a single component attached to the system bus of the CPU, we enable a deep integration of NDPs, supporting coherence, virtual memory, and accessing the global address space. The proposed architecture limits the negative impact on the normal CPU performance, making sure that other workloads can continue working as expected. The use of standard OS methods for data allocation and data locality management are important, as they make the architecture easy to use. By means of synthetic benchmarks and an graph traversal workload, we show that the proposed mixed fine- and coarse-grained coherence mechanisms as well as the remote data access mechanisms provide excellent performance. By quantifying the coherence and data writeback overhead, we argue that NDP workloads should work for a prolonged period of time on large data sets. Furthermore, we showed that four NDPs with an almost completely flat memory hierarchy can easily outperform a modern CPU. This work can serve as a base for further research in how NDPs can be integrated tightly with CPUs and other processing devices.

REFERENCES

- [1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, 105–117.
- [2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, 336–348.

- [3] H. Asghari-Moghaddam, A. Farmahini-Farahani, K. Morrow, J. H. Ahn, and N. S. Kim. 2016. Near-DRAM acceleration with single-ISA heterogeneous processing in standard memory modules. *IEEE Micro*. 36, 1 (2016), 24–34.
- [4] E. Azarkhish, D. Rossi, I. Loi, and L. Benini. 2015. High performance AXI-4.0 based interconnect for extensible smart memory cubes. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 1317–1322.
- [5] E. Azarkhish, D. Rossi, I. Loi, and L. Benini. 2016. Design and evaluation of a processing-in-memory architecture for the smart memory cube. In *International Conference on Architecture of Computing Systems*. Springer, 19–31.
- [6] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. 2014. Near-Data processing: Insights from a MICRO-46 workshop. *IEEE Micro*. 34, 4 (2014), 36–42.
- [7] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. 2013. Efficient virtual memory for big memory servers. *SIGARCH Computer Architecture News* 41, 3 (June 2013), 237–248.
- [8] S. Beamer, K. Asanović, and D. Patterson. 2012. Direction-optimizing breadth-first search. In *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society, 1–10.
- [9] A. Boroumand, S. Ghose, B. Lucia, K. Hsieh, K. Malladi, H. Zheng, and O. Mutlu. 2017. LazyPIM: An efficient cache coherence mechanism for processing-in-memory. *IEEE Comput. Arch. Lett.* 16, 1 (2017), 46–50.
- [10] A. Buluç and K. Madduri. 2011. Parallel breadth-first search on distributed memory systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 65.
- [11] A. M. Caulfield, L. M. Grupp, and S. Swanson. 2010. Gordon: An improved architecture for data-intensive applications. *IEEE Micro*. 30, 1 (2010), 121–130. IEEE Micro Top Picks.
- [12] A. De, M. Gokhale, R. Gupta, and S. Swanson. 2013. Minerva: Accelerating data analysis in next-generation SSDs. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 9–16.
- [13] M. Dobson, P. Gaughe, M. Hohnbaum, and E. Focht. 2003. Linux support for NUMA hardware. In *Proceedings of the Linux Symposium*. 169–184.
- [14] B. Falsafi, M. Stan, K. Skadron, N. Jayasena, Y. Chen, J. Tao, R. Nair, J. Moreno, N. Muralimanohar, K. Sankaralingam, and C. Estan. 2016. Near-memory data services. *IEEE Micro*. 36, 1 (2016), 6–13.
- [15] A. Farmahini-Farahani, J. Ho Ahn, K. Morrow, and N. Sung Kim. 2015. DRAMA: An architecture for accelerated processing near memory. *Comput. Arch. Lett.* 14, 1 (2015), 26–29.
- [16] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *High Performance Computer Architecture*. IEEE, 283–295.
- [17] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson. 2010. Introduction to the wire-speed processor and architecture. *IBM J. Res. Dev.* 54, 1 (2010), 3–1.
- [18] M. Gao, G. Ayers, and C. Kozyrakis. 2015. Practical near-data processing for in-memory analytics frameworks. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*. IEEE, 113–124.
- [19] M. Gao and C. Kozyrakis. 2016. HRL: Efficient and flexible reconfigurable logic for near-data processing. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. IEEE, 126–137.
- [20] H. Giefers, R. Polig, and C. Hagleitner. 2015. Accelerating arithmetic kernels with coherent attached FPGA coprocessors. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition*. EDA Consortium, 1072–1077.
- [21] M. Gorman. 2009. Understanding the Linux Virtual Memory Manager. Available online: <https://www.kernel.org/doc/gorman/>.
- [22] Graph500. 2017. Graph500. Retrieved from <http://www.graph500.org/>.
- [23] Z. Guz, M. Awasthi, M. Balakrishnan, G. Mrinmoy, A. Shayesteh, and T. Suri. 2014. Real-time analytics as the killer application for processing-in-memory. *Proceedings of the 2nd Workshop on Near-Data Processing* (2014). Available online: <http://www.cs.utah.edu/wondp/guz.pdf>.
- [24] G. C. Harvey. 1996. *Memory Systems and Pipelined Processors*. Jones and Bartlett Publishers. Page 178.
- [25] HMC Consortium. 2017. Homepage. Retrieved from <http://www.hybridmemorycube.org/>.
- [26] H. P. Hofstee, G. C. Chen, F. H. Gebara, K. Hall, J. Herring, D. Jamsek, J. Li, Y. Li, J. W. Shi, and P. W. Y. Wong. 2013. Understanding system design for big data workloads. *IBM J. Res. Dev.* 57, 3/4 (2013), 3–1–3.10.
- [27] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *Proceedings of the International Conference on Computer Design*. IEEE, 25–32.
- [28] IBM. 2014. Summit and Sierra Supercomputers. Retrieved from http://www.teratec.eu/actu/calcul/Nvidia_Coral_White_Paper_Final_3_1.pdf.
- [29] IBM. 2017. OpenPower. Retrieved from <http://open-power.org/> (2017).

- [30] IBM. 2017. POWER9 CPU. Retrieved from http://www.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.921-.POWER9-Thompto-IBM-final.pdf.
- [31] IBM. 2017. System Energy Estimator. Retrieved from <http://www-912.ibm.com/see/EnergyEstimator> (2017).
- [32] Intel. 2017. A revolutionary breakthrough in memory technology. Retrieved from http://www.intel.com/newsroom/kits/nvm/3dxdpoint/pdfs/Launch_Keynote.pdf (2017).
- [33] C. Johnson, D. H. Allen, and J. Brown. 2010. A wire-speed power processor: 2.3GHz 45nm SOI with 16 cores and 64 threads. In *IEEE International Solid-State Circuits Conference*. IEEE, 104–105.
- [34] J. Jung and Y. Won. 2016. Nvramdisk: A transactional block device driver for non-volatile RAM. *IEEE Trans. Comput.* 65, 2 (2016), 589–600.
- [35] J. Lee, J. H. Ahn, and K. Choi. 2016. Buffered compares: Excavating the hidden parallelism inside DRAM architectures with lightweight logic. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition*. IEEE, 1243–1248.
- [36] S. Li, J. Ho Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of MICRO-42*. IEEE, 469–480.
- [37] G. H. Loh, N. Jayasena, M. H. Oskin, M. Nutter, D. Roberts, M. M. Dong, P. Zhang, and M. Ignatowski. 2013. A processing-in-memory taxonomy and a case for studying fixed-function PIM. In *Workshop on Near-Data Processing*. Available online: <http://www.cs.utah.edu/wondp/wondp2013-paper2-final.pdf>.
- [38] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.* 40, 6 (2005), 190–200.
- [39] McKinsey Global Institute. 2011. Big data: The next frontier for innovation, competition and productivity (2011). Available online: <http://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/big-data-the-next-frontier-for-innovation>.
- [40] A. McLaughlin and D. A. Bader. 2014. Revisiting edge and node parallelism for dynamic GPU graph analytics. In *Proceedings of the Parallel Distributed Processing Symposium Workshops*. IEEE, 1396–1406.
- [41] R. Nair, S. F. Antao, and C. Bertolli. 2015. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM J. Res. Dev.* 59, 2/3 (2015), 17-1–17-14.
- [42] J. Navarro, S. Iyer, P. Druschel, and A. Cox. 2002. Practical, transparent operating system support for superpages. *SIGOPS Operat. Syst. Rev.* 36, SI (2002), 89–104.
- [43] S. H. Pugsley, J. Jestes, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and F. Li. 2014. Comparing implementations of near-data computing with in-memory mapreduce workloads. *IEEE Micro*. 34, 4 (2014), 44–52.
- [44] R. Raghavan, R. J. Eickemeyer, and A. C. Sawdey. 2015. IBM POWER8 performance and energy modeling. *IBM J. Res. Dev.* 59, 1 (Jan. 2015), 10:1–10:10.
- [45] D. Sanchez and C. Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. *SIGARCH Comput. Arch. News* 41, 3 (2013), 475–486.
- [46] S. Sarkar, P. Sewell, and J. Alglave. 2011. Understanding POWER multiprocessors. *SIGPLAN Not.* 46, 6 (2011), 175–186.
- [47] B. Sinharoy, R. Kalla, and W. J. Starke. 2011. IBM POWER7 multicore server processor. *IBM J. Res. Dev.* 55, 3 (May 2011), 1:1–1:29.
- [48] B. Sinharoy, J. A. Van Norstrand, and R. J. Eickemeyer. 2015. IBM POWER8 processor core microarchitecture. *IBM J. Res. Dev.* 59, 1 (2015), 2:1–2:21.
- [49] D. J. Sorin, M. D. Hill, and D. A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers.
- [50] W. J. Starke, J. Stuecheli, and D. M. Daly. 2015. The cache and memory subsystems of the IBM POWER8 processor. *IBM J. Res. Dev.* 59, 1 (2015), 3:1–3:13.
- [51] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel. 2015. CAPI: A coherent accelerator processor interface. *IBM J. Res. Dev.* 59, 1 (2015), 7:1–7:7.
- [52] M. Y. Thompson, J. M. Barton, T. A. Jermoluk, and J. C. Wagner. 1988. Translation lookaside buffer synchronization in a multiprocessor system. In *Proceedings of the USENIX Winter Conference*. 297–302.
- [53] S. F. Yitbarek, T. Yang, R. Das, and T. Austin. 2016. Exploring specialized near-memory processing for data intensive operations. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition*. IEEE, 1449–1452.
- [54] D. Zhang, N. Jayasena, and A. Lyashevsky. 2014. TOP-PIM: Throughput-oriented programmable processing in memory. In *Proceedings of the International Symposium on High-performance Parallel and Distributed Computing*. ACM, 85–98.
- [55] H. Zhang, G. Chen, B. Chin Ooi, K. Tan, and M. Zhang. 2015. In-memory big data management and processing: A survey. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1920–1948.

Received January 2017; revised May 2017; accepted July 2017