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

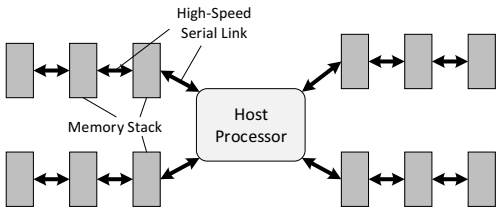

Figure 1. The Near Data Processing (NDP) architecture.

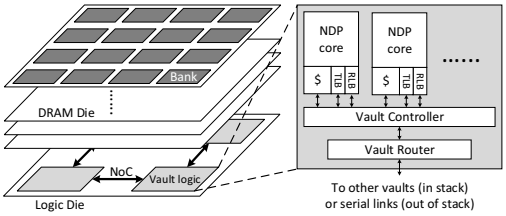

Figure 2. 3D memory stack and NDP components.

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,

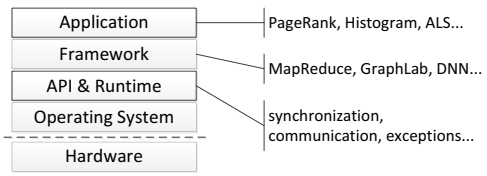

Figure 3. The NDP software stack.

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.

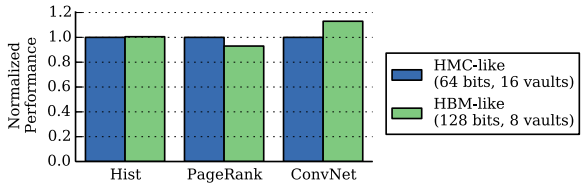

Figure 5. Performance impact of stack design.

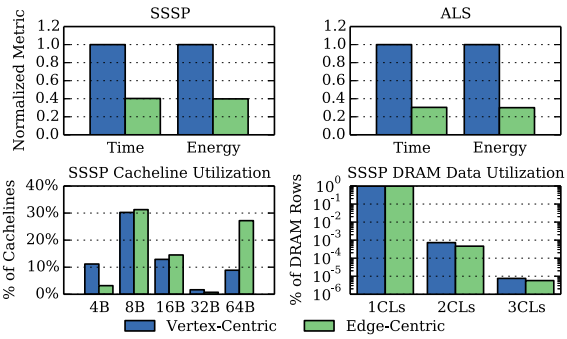

Figure 6. Vertex-centric and edge-centric graph frameworks.

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.

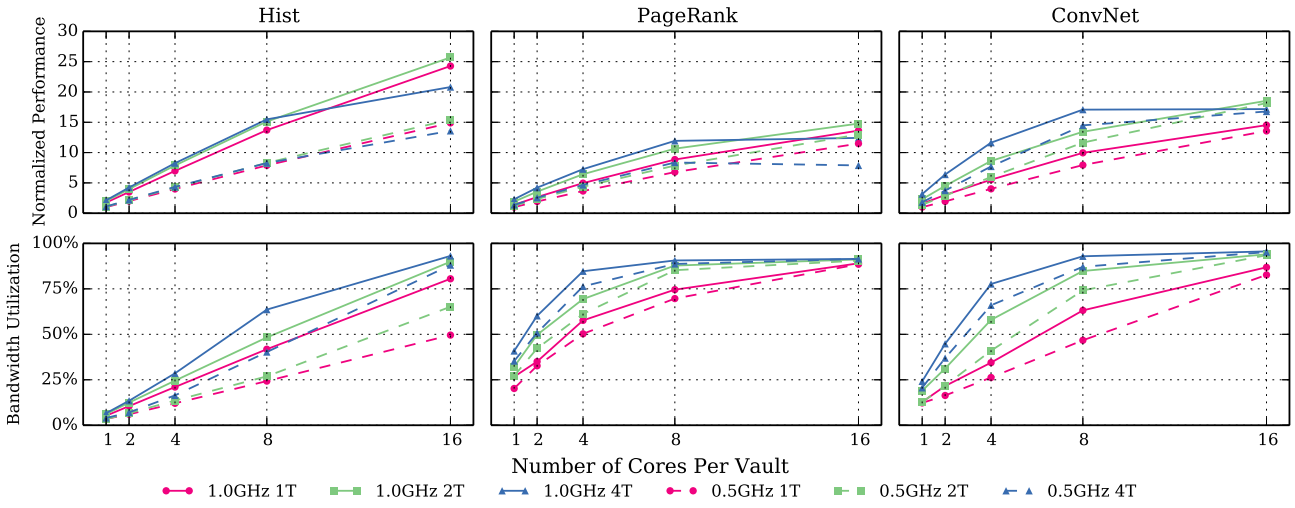

Figure 4. Performance scaling and bandwidth utilization for a single-stack NDP system.

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.

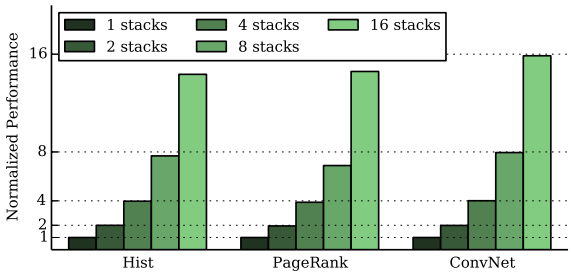

Figure 7. Performance as a function of the number of stacks.

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

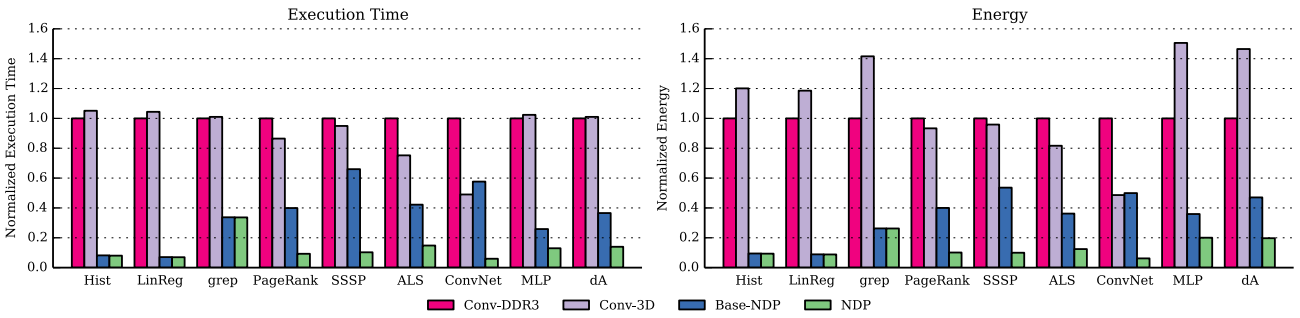

Figure 8. Performance and energy comparison between Conv-DDR3, Conv-3D, Base-NDP and NDP systems.

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

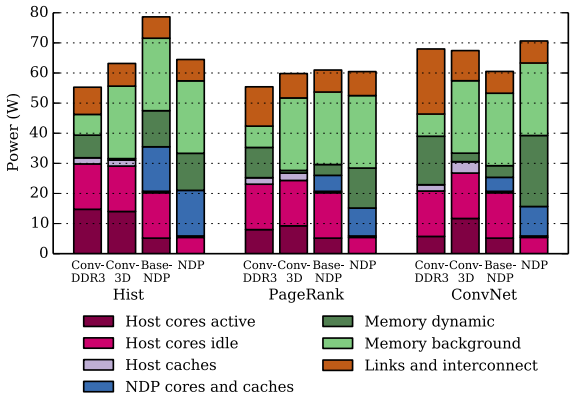

Figure 9. System power breakdown (host processor + 8 stacks).

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.

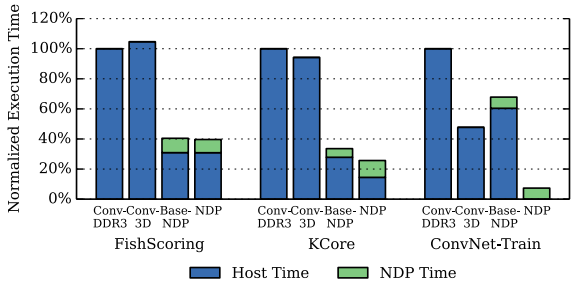

Figure 10. Hybrid workload performance comparison.

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.