An Architecture for Near-Data Processing Systems

Erik Vermij
IBM Research
the Netherlands
erik.vermij@nl.ibm.com

Christoph Hagleitner
IBM Research – Zurich
Switzerland
hle@zurich.ibm.com

Leandro Fiorin
IBM Research
the Netherlands
leandro.fiorin@nl.ibm.com

Rik Jongerius
IBM Research
the Netherlands
r.jongerius@nl.ibm.com

Jan van Lunteren
IBM Research – Zurich
Switzerland
jvl@zurich.ibm.com

Koen Bertels
Delft University of Technology
the Netherlands
k.l.m.bertels@tudelft.nl

ABSTRACT

Near-data processing is a promising paradigm to address the bandwidth, latency, and energy limitations in today's computer systems. In this work, we introduce an architecture that enhances a contemporary multi-core CPU with new features for supporting a seamless integration of near-data processing capabilities. Crucial aspects such as coherency, data placement, communication, address translation, and the programming model are discussed. The essential components, as well as a system simulator, are realized in hardware and software. Results for the important Graph500 benchmark show a 1.5x speedup when using the proposed architecture.

1. INTRODUCTION

The world is more connected than ever before, by means of a wide variety of social media, and modern voice-, text- or image-based communication methods. Besides personal data, companies and institutes are piling up seismic data, atmospheric data, all kinds or traffic related data, etc., and with strong movement towards 'the internet of things', more and more data will be generated and stored. Creating value from huge amounts of data is becoming an ever more important task for computer systems. Unfortunately, not only are the tasks becoming harder, traditional computer systems are also becoming relatively worse at handling them. There is a growing gap between processing performance of CPUs, GPUs, and other computing devices, and the bandwidth and latency of those devices towards the data they require.

This work is 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, the Netherlands.

CF'16 May 16-19, 2016, Como, Italy

ACM ISBN 978-1-4503-4128-8/16/05.

DOI: http://dx.doi.org/10.1145/2903150.2903478

In this work, we present an architectural extension to a contemporary server, specifically aimed at bridging the gap between what today's data-intensive workloads demand, and what existing computer architectures can offer. To achieve this, we make use of the near-data processing paradigm, which has received renewed attention in the last years [3], especially for data-intensive workloads like graph-processing [7], on top of a memory system organization as found in the IBM POWER8 CPU [11]. This CPU has two levels of memory controllers (MCs): at the CPU we have up to eight memory-technology agnostic MCs, while the memory-technology specific MCs are located on separate 'memory buffer' chips, tightly coupled with the main memory, and are connected to the CPU by means of CPU- and memory-agnostic high-speed links ('*DMI*' links).

We propose the addition of processing capabilities to such a 'memory-side chip', bringing processing very close to the main memory. This is shown in Figure 1, where near-data processors (NDPs) are directly attached to the memory-side MCs. The focus of this paper is to discuss the architectural implications on both the CPU and the NDP side, the required software, and workload partitioning. Because of space limitations, we leave to future work a detailed discussion about the NDP implementation and optimization for different memory technology. We consider the NDP-specific part of the memory-side chip to be either several general-purpose cores, or an embedded reconfigurable area, allowing for workload-optimized cores through reconfiguration.

2. RELATED WORK

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, near-data processing has been recently re-discovered. In [5], it has been investigated for accelerating big-data workloads with poor locality in SSDs, while in [10, 8, 2] accelerator architectures embedded in the logic layer of 3D-stacked DRAM devices, such as the Micron's Hybrid Memory Cube (HMC), have been proposed and studied. From a software perspective, near-data processing related work was presented by Trancoso [12], focusing on application partitioning, and Chu [4], focusing on a high-level programming paradigm. Work in [1] studies the execution of separate instructions at the CPU or directly at the memory, based on hardware-managed locality profiling. In [6], the authors propose the execution of kernels on coarse-grain

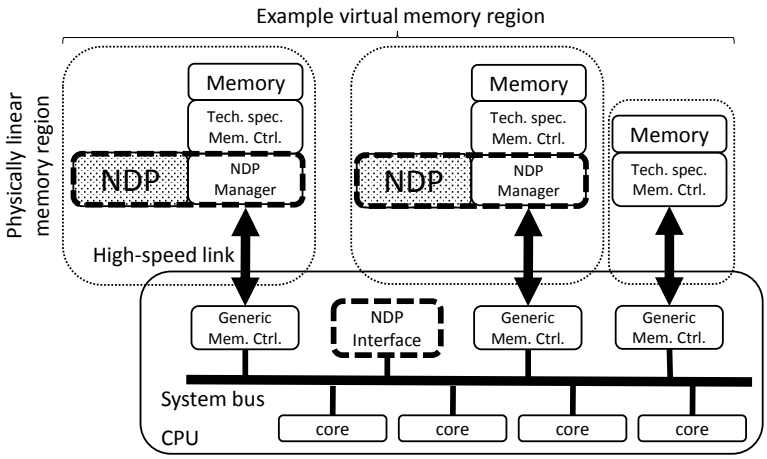

Figure 1: A CPU environment enhanced with near-data processing capabilities.

reconfigurable architectures (CGRAs) on top of DRAM devices. Our work complements the above mentioned work by introducing the system-level architecture that supports arbitrary NPD kernels in a CPU environment.

3. ARCHITECTURE OF AN NDP-ENHANCED CPU

Figure 1 shows the proposed architecture, where the CPU is enhanced with NDPs. The CPU has several high-speed links, and for illustrative purposes, one is connected to a 'default' memory-side chip, while two are connected to NDPs, although fully NDP-equipped systems are more likely. NDPs are tightly coupled with their respective memory controller, and have their own contiguous physical address range. However, an NDP can be invoked by a process that has its virtual address space distributed across multiple physical regions (see Figure 1), possibly even in a multi-node SMP domain. An NDP with its local memory can therefore be seen as a NUMA domain in an asymmetrical multi-processor system, being a sub-node of the CPUs NUMA domain, with different access characteristics to local and global memory.

The three functional requirements of our system are:

- to enable near-data processing capabilities, while being minimally-invasive, in an existing CPU architecture;
- to implement arbitrary near-data processing functionality, ranging from workload-optimized cores, to general-purpose multi-core processors;
- to dereference all virtual pointers of the host process on the NDP, coherently with the CPUs view of the memory.

3.1 Required architectural features

The architectural features needed to implement the above described functional requirements are shown in Figure 1 and described below:

- The MC functionality must be split in two: a generic MC at the CPU, and a technology-specific MC at the memory-side chip.
- An NDP-Manager (NDPM) is introduced between the high-speed link and the memory-side MC. The NDPM is the interface component towards the NDP, offering clean memory and communication ports. It furthermore provides all support-functionality to get arbitrary workloads up and running, like coherency, address translation, and accessing global data.
- The NDP-Interface (NDPI) is attached to the system bus. This component (i) handles the communication between the core (OS and/or user application) and the NDPMs, (ii) represents the access point to the global coherent memory space from the NDPs perspective, and (iii) provides address translation capabilities for the NDPMs by having its own memory-management unit, capable of walking the page tables. A single NDPI manages all the NDPMs, because adding ports to the system bus is very costly, and designing a component working at full system-bus load does not represent a challenge.

3.2 Communication

To support the communication between the NDPM and the NDPI, we generalized the traffic protocol managing the high-speed serial link. Instead of being load/store-only, it was extended to be able to also carry the communication between the the NDPI and NDPM. A single identification byte per every 128 byte frame (a cacheline) was added, introducing a neglectable overhead for the original traffic. Both at the CPU and the NDPM side original data traffic and messages can easily be separated and forwarded to the appropriate data or communication path.

Software can send messages to the NDPI by means of (existing) architecture specific instructions, which force a cache line directly on the system bus to be snooped by the NDPI. This gives us the possibility to dispatch messages with very little latency and overhead, in contrast to memory-mapped solutions. Messages from the NDPI to the software use an interrupt-based mechanism, to again have a low latency solution.

3.3 Coherency

Coherency is a crucial issue for near-data processing, for which several approaches have been proposed in academic

work. They range from positioning the NDP in a separate address space (thus not requiring coherency), to manually enforced coherency, as well as proposals requiring changes to the CPU's coherency methods. We believe hardware-managed coherency will be necessary for any successful device, as doing this manually is very difficult, if not impossible. Furthermore, as stated in the functional requirements, we believe changes to the CPU should be kept to a minimum.

In our work, to enforce coherency between the NDP and the CPU, we use the basic MSI (modified-shared-invalid) coherency protocol, implemented between the NDPM and NDPI. Devices (CPU and NDPs) can therefore send messages like *getS*, *getM*, *Upg* etc. towards the NDPM. This approach does not require a change in the coherency protocol on the CPU, because the MSI state is a subset of the much more complex coherency protocols found in contemporary CPUs. When the NDP wants ownership of a cache line, an invalidate signal is sent to the NDPI. If the cache line was modified in the CPU cache, it is first written back to main memory. If the CPU needs ownership of a cache line, it invalidates it at the NDPM and possibly reads the cache line from main memory.

The state of the memory is managed at the NDPM, and every line has either the state *CPU owned (CPU)*, *NDP owned (NDP)*, or *Shared*. This makes possible to share data between the CPU and the NDPs, without having to invalidate the copy on the original device. Furthermore, since cache lines stay in a *device-owned* state even after eviction, devices can access their datasets as *fast local memory*, as no coherency state changes are required. The state is stored in a directory data structure, kept in DRAM and accessed via a hardware-managed cache.

3.4 Data placement and scaling

Typically, CPUs stripe accesses to subsequent cache lines across different memory channels but also supports contiguous memory regions per memory channel. The latter configuration is used in this work for the memory channels that feature NDPs. The memory management in the operating system needs to be made aware of this, and it needs to be able to allocate data within a certain physical range.

This setup scales well to multiple NDPs for applications that can be parallelized on today's distributed memory clusters (e.g., map-reduce, MPI, OpenSHMEM). The system bus, even for heavy inter-NDP traffic, will not represent a bottleneck, as it is designed to be able to saturate the memory channels. For some applications, the proposed memory setup can limit the bandwidth realizable by the CPU. If this becomes a problem, the default, striped, memory organization can be used in combinations with NDPs as long as the application has parallelism at the level of single cache lines.

3.5 Address translation

Address translation is managed by the NDPM and the NDPI. The NDPM contains a translation lookaside buffer (TLB), holding 64 translations. As we are foremost interested in workloads with large datasets, the small size of the TLB is not a limitation. Modern versions of the OS memory management will always try to satisfy an allocation with 'huge' pages, in a transparent way, to reduce the TLB pressure. If a TLB miss occurs, the NDPM ask the NDPI for the correct translation, which uses its memory-management unit to walk the page tables. This mechanism for handling misses has a significant latency, even though this is not necessarily the penalty incurred by the NDP, as other cores or load/store streams that are not affected by the miss can still continue to execute.

3.6 Accessing global data

Accessing global data results typically in an address translation miss at the NDPM, and after receiving the correct translation, the NDPM recognizes it as being non-local. The access is forwarded to the NDPI, to be issued there. In case of a *write*, the NDPI claims the associated cache line and puts it in its cache, followed by the actual write. In case of a *load*, the NDPI gets a shared copy of the cache line, does the read, and replies the result back to the NDPM.

3.7 NDPM memory model

The memory interface offered to the NDP is as wide as the interface of the MCs, and we expect the NDP to have its design optimized for this width. Therefore, the NDPM does not implement a coalescing-like mechanism. However, the NDPM does implements a weakly ordered memory model to be able to hide long-latency events like address-translation misses, and accessing global data. A strict model would halt the entire NDPM and thereby the NDP, resulting in an unnecessary drop in performance. When in-order commitment or synchronization is required, a *barrier* command can be issued.

3.8 NDP programming paradigm

One process can make use of an NDP at a given time, but the memory behind the NDP is not exclusive to that process. From the NDP's perspective, accessing data located behind a different MC is much slower than accessing data stored in its own memory. We therefore embrace the local/global data concept as used, for example, in the PGAS (Partitioned Global Address Space) paradigm to make, for the user, an explicit distinction between which data is stored in an NDP's own memory, and which data is (or can be) stored somewhere else. This is conceptually not different from NUMA optimizations done on today's multi-CPU systems. During runtime, the user requests NDP access and, when granted, *local* allocations are done in its memory and communication is sent to the respective NDPM. Since the NDP supports arbitrary functionality, multiple methods of operation exist, as described below.

3.8.1 Application specific hardware functions

When using predefined or user-defined hardware functionality, typically stored in an image store, the user makes explicit calls to *set* and *start* methods. A user-level message interface, acting as a tunnel between NDP and host application, makes possible to send commands/information to the NDP and retrieve information during runtime, for example when the NDP is in stand-by all the time, waiting to receive work items. Automatically generated and executed hardware functions are envisioned as well [9], and the clean separation between NDPM and NDP makes this possible in our architecture.

3.8.2 General-purpose cores

When using general-purpose cores, annotated kernels need to be compiled for the target instruction set. The compiler

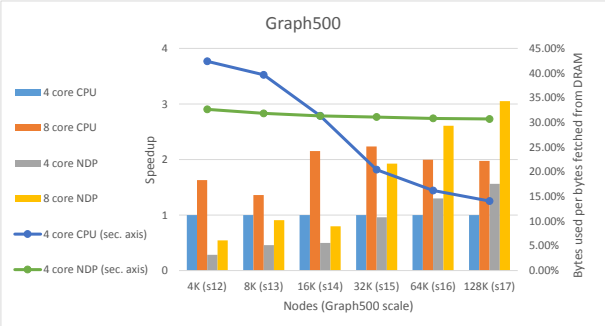

Figure 2: Graph500 performance comparison between CPU and NDP, and DRAM access efficiency.

will insert the necessary commands in the host instruction stream, and the described user-level communication interface can be interpreted as a inter-processor interrupt channel.

4. RESULTS

To validate the architecture, we developed the NDPM in VHDL, and a software-based system simulator was developed around it. At the simulated CPU we can execute arbitrary code, and at the NDPM NDP-interfaces we can connect both workload-optimized cores developed in a hardware description language, as well as general-purpose multi-core processors simulated in software. The NDP-interface offers 32 GB/s bandwidth at 32 B granularity, and a 30ns latency, while the CPU can use half of that bandwidth, at 128 B granularity, with 80ns latency. We evaluated several applications, but, for space limitation, we discuss here only the results obtained for the well-known Graph500 benchmark. In Figure 2, we show the results for running Graph500 on four or eight CPU cores and four or eight general-purpose NDP cores, all with a single memory channel. The CPU cores are four times faster the NDP cores, and the NDP cores have only 512 B caches. For small problem sizes, the fast CPU cores can work from their close-by caches, and perform much better than the NDP cores. When the essential data structures outgrow the CPU's cache size, the NDP starts to benefit from its lower latency and smaller access granularity to main memory. As the NDPs spend less cycles waiting for data, they achieve up to 1.5x speedup with respect to the CPU.

Figure 2 shows also the ratio between the amounts of bytes fetched from DRAM, and the amount of bytes actually used. When the data structures that are accessed in a scattered way do not fit in the cache anymore, this ratio worsen dramatically for the CPU, given its 128 B access granularity. The NDP on the contrary shows only a very modest decline, since the NDP caches are so small we are not exploiting a lot of locality anyway. The NDPs are able to access to the DRAM at least twice as efficiently as the CPU.

5. CONCLUSION

In this work, we presented the architecture of a system that combines NDPs and a modern CPU. The proposed architecture introduces the NDP-Interface and NDP-Manager components for handling essential features like coherency, communication and address translation, for a seamless integration with a standard CPU. An implementation of crucial components is made, and an implementation of the relevant Graph500 benchmark shows a significant 1.5x speedup with respect to the CPU implementation.

6. REFERENCES

[1] J. Ahn and et al. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Annual International Symposium on Computer Architecture*, (ISCA), pages 336–348, New York, NY, USA, 2015. ACM.

[2] E. Azarkhish and et al. High Performance AXI-4.0 Based Interconnect for Extensible Smart Memory Cubes. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, (DATE), pages 1317–1322, San Jose, CA, USA, 2015. EDA Consortium.

[3] R. Balasubramonian and et al. Near-Data Processing: Insights from a MICRO-46 Workshop. *Micro, IEEE*, 34(4):36–42, July 2014.

[4] M. Chu and et al. High-level Programming Model Abstractions for Processing in Memory. In *1st Workshop on Near-Data Processing in conjunction with the International Symposium on Microarchitecture*, (WoNDP), 2013.

[5] A. De and et al. Minerva: Accelerating Data Analysis in Next-Generation SSDs. In *IEEE International Symposium on Field-Programmable Custom Computing Machines, (FCCM) , Seattle, WA, USA*, pages 9–16, 2013.

[6] A. Farmahini-Farahani and et al. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *High Performance Computer Architecture, 2015*, (HPCA), pages 283–295, Feb 2015.

[7] Z. Guz and et al. Real-Time Analytics as the Killer Application for Processing-In-Memory. *2nd Workshop on Near-Data Processing (WoNDP)*, 2014.

[8] R. Nair and et al. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3):17:1–17:14, March 2015.

[9] E. Panainte, K. Bertels, and S. Vassiliadis. Compiling for the Molen Programming Paradigm. In P. Y. K. Cheung and G. Constantinides, editors, *Field Programmable Logic and Application*, volume 2778 of *Lecture Notes in Computer Science*, pages 900–910. Springer Berlin Heidelberg, 2003.

[10] S. Pugsley and et al. Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads. *Micro, IEEE*, 34(4):44–52, July 2014.

[11] W. Starke and et al. The cache and memory subsystems of the IBM POWER8 processor. *IBM Journal of Research and Development*, 59(1):3:1–3:13, Jan 2015.

[12] P. Trancoso. Moving to Memoryland: In-memory Computation for Existing Applications. In *ACM International Conference on Computing Frontiers*, (CF), pages 32:1–32:6, New York, NY, USA, 2015. ACM.