Sorting big data on heterogeneous near-data processing systems

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

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

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

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

ABSTRACT

Big data workloads assumed recently a relevant importance in many business and scientific applications. Sorting elements efficiently in big data workloads is a key operation. In this work, we analyze the implementation of the mergesort algorithm on heterogeneous systems composed of CPUs and near-data processors located on the system memory channels. For configurations with equal number of active CPU cores and near-data processors, our experiments show a performance speedup of up to 2.5, as well as up to 2.5× energy-per-solution reduction.

1. INTRODUCTION

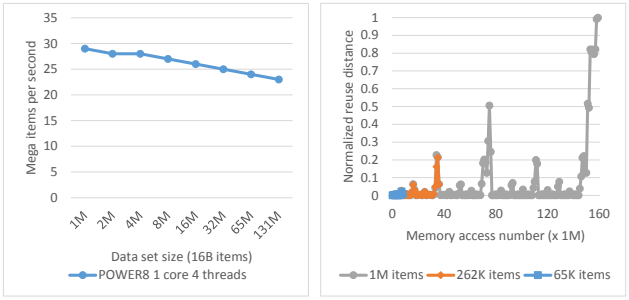
The efficient analysis of big data workloads represents a key element in many businesses and scientific and engineering applications [12]. A significant amount of data, often stored in unprocessed form, need to be extensively searched and analyzed, creating substantial challenges to all the components of the computing system [13].

Sorting elements is often one of the key operation to be performed on big data workloads: For example, *TeraSort* is a sorting application, included in the Apache Hadoop distribution, which is widely used by many big data vendors to benchmark and stress test computing clusters [12].

This work analyzes the sorting of big data workloads on near-data processing architectures [8]. This computing paradigm, recently rediscovered, allows to alleviate the classical "memory wall problem" by moving the computation closer to the memory. For certain types of workload which typically do not benefit from the availability of a complex hierarchy of caches, it represents a clear advantage in terms of latency reduction, available bandwidth to memory, and energy efficiency [19].

Our work focuses on sorting big data on a heterogeneous system composed of a CPU and near-data processors (NDPs), in which NDPs are implemented as workload-optimized processors on FPGA. Moreover, we present a dynamic workload balancing mechanism that allows to optimize the utilization of CPU and NDPs and increase the overall sorting performance when working on large data sets, by using near-data processors in a heterogeneous way to execute the phases of the application with little data locality.

The remainder of this paper is organized as follows: Section 2 presents background information and related work. Section 3 discusses the reference architecture and its system simulator used in our experiments. Section 4 discusses the optimized implementation of the sort algorithm on the heterogeneous architecture. Section 5 presents an analysis of the experimental results, while Section 6 concludes the paper.



(a) (b)

Figure 1: (a) Mergesort single-core performance, while varying the size of the data set. (b) Reuse distance when when sorting three different lengths of sequences with mergesort.

2. MOTIVATION AND RELATED WORK

In order to evaluate the performance of modern processors while sorting big data workloads, we run an implementation of *mergesort* on a 2-socket IBM POWER8 machine. Each socket has 10 cores running at 4.2 GHz, with each core provided with a 64 KB L1 cache, a 512 KB L2 cache, and a 8 MB semi-shared L3 cache. Without loss of generality, we focus on single-core performance. Mergesort is a well known $O(nlog(n))$ sorting algorithm which conceptually first divides a list of n elements in n sublists on 1 element, and then repeatedly merges two sublists at the time to produce new sorted sublists, until only one sorted list remains.

Figure 1 shows the results of our preliminary evaluation for the case of 4 threads per core, in which we operate on

CF'17, May 15-17, 2017, Siena, Italy

DOI: http://dx.doi.org/10.1145/3075564.3078885

16 B items, consisting of an 8 B index (to be sorted), and an 8 B pointer. As Figure 1a shows, the performance decreases with the size of the data set. This behavior can be explained by analyzing Figure 1b, which shows the *reuse distance* when sorting three different lengths of sequences with mergesort. The reuse distance is calculated as the number of memory accesses to unique addresses performed in between two memory accesses to the same address, and it is a measure of temporal data locality. In case of a large reuse distance, the availability of local memories does not provide any advantage, and memory accesses are performed on a higher-level storage, e.g., DRAM for a typical CPU, resulting in lower bandwidths, higher latencies, and lower performance. Figure 1b highlights that bigger data set will cause more phases with a high reuse distance. These phases will run slower compared to the regions with a small reuse distance, causing the performance reduction observed in Figure 1a.

In this work, we therefore propose a heterogeneous approach in which the mergesort phases with good data locality are executed at the CPU, while the phases little data locality are executed by the near-data processor. This follows the fundamental properties of a memory hierarchy and near-data processing. A cache at the CPU offers high bandwidths and low energy costs per access, while main memory offers mediocre bandwidths and high energy costs per access. When an access pattern can be satisfied from the caches, that is the preferred method. When an access pattern can only be satisfied from main memory, it is best to execute it on a near-data processor, to limit data movement.

2.1 Overview of related work

Sorting unstructured items represents an important task for big data applications [12]. In general, several platform designs and optimizations have been proposed to deal with big data applications. As big data workloads fundamentally differ from workloads usually run on more traditional data-warehousing systems, a tailored system-level optimization is needed [12]. A custom system and board design is proposed in [4]. The system, designed around Flash and DRAM memories, targets the energy-efficient execution of big data workloads. Hardware acceleration of big data frameworks such as Hadoop MapReduce has been proposed, delegating the *Map()* and *Reduce()* procedures to a many-core processor [13], or a cluster of FPGAs [7]. An overall performance improvement of up to 6.6× and 4.3× has been reported for the many-core processor and the FPGA implementation, respectively. Acceleration of MapReduce with GPUs is discussed in [5], which reports speedups over single CPU core execution ranging from 3.25 to 28.68.

In this paper, we focus on near-data processing architectures, which has been recently re-discovered thanks to technology advances on 3D stacking, and due to the availability of big data workloads with high degrees of parallelism and programming models for distributed big data applications [3]. Most of the proposed near-data architectures extend the logic layer found in 3D-stacked DRAM devices, such as the Micron's Hybrid Memory Cube (HMC) [11], by adding small general purpose cores or dedicated accelerators to it [16, 15, 2, 1, 9], and exploit the finer data access granularity and the higher memory bandwidth available.

A near-data architectures for big data graph processing has been proposed in [1], while the work in [9] discusses an architecture which targets the execution of big data analytic frameworks near the memory. In [22], a sorting accelerator is implemented as part of a 3D-stacked memory system.

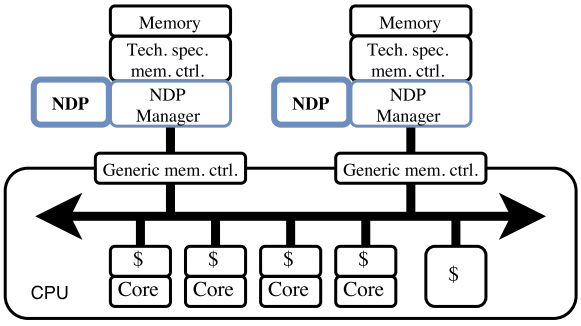


Figure 2: High-level view of the system organization. Near-data processors are tightly integrated with the memory controllers.

Depending on the application and architecture implementation, typical reported performance speedups are from 3× to 16×, with significant energy reduction over CPU-based implementations.

Our experiments relies on the work presented in [20], where an NDP extension to a POWER8-based server environment is described. Differently to related work, the coprocessor is tightly integrated with the memory controllers and supports coherence between CPU and NDPs. Moreover, while the presented approaches to integrate the NDPs into the POWER8 system are generic enough, the work in [20] focuses on a workload-optimized NDPs implemented on FPGA, able to run specific workloads very efficiently. This paper also shows, for the first time, how exploiting the characteristics of a heterogeneity system composed of a CPU and NDPs can improve significantly the performance of sorting big data sets, while reducing the overall power consumption.

3. NEAR-DATA ARCHITECTURE

A high-level view of the reference system architecture is shown in Figure 2 [20]. The architecture relies on two-level memory controllers: Memory-technology agnostic memory controllers are implemented at the CPU, while controllers specific for the adopted memory technology are tightly coupled to the main memory. An example of such a setup is the memory system that can be found on the IBM POWER8 CPU [17], which has eight memory-technology agnostic memory channels each connecting to a 'memory buffer' chip, holding four DDR3/4 memory controllers. Another example can be found on CPUs connected to 3D-stacked stacked memory devices such as the Hybrid Memory Cube [11], in which technology-specific memory controllers (called *vault controllers*) are implemented in the logic layer of the device.

As shown in Figure 2, near-data processing capabilities are added in the technology-specific memory controllers. Each NDP relies on an hardware component, called NDP-Manager (NDP-M) [20], which provides the NDP with support for interfacing the system, including virtual memory, coherence, and communication, in way conceptually similar to what implemented by the CAPI interface in POWER8 CPUs for interfacing external coprocessors [18]. The NDP-M allows to interface any type of NDP, such as for instance general purpose processors or workload-optimized cores. In this work, we focus in particular on using workload-optimized NDPs implemented on a reconfigurable fabric, such as an FPGA.

Table 1 shows specification for the reference system considered in this work. The system template we use in the

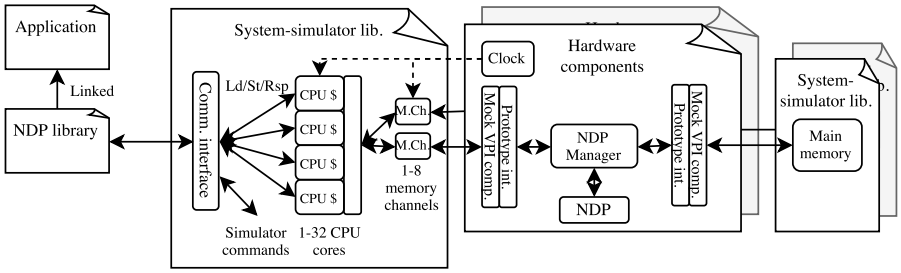


Figure 3: The simulator environment used in our experiments.

Table 1: System specifications.

Simulated CPU	
Cores	4.2 Ghz
Access granularity	128 byte
Data caches	128 KB / core
Memory channels	20:10 GB/s Up:Down
Main memory x4	
Technology and bandwidth	4 x DDR4-2400 @ 60% : 48 GB/s
Access granularity	32 byte
NDP	
Frequency	500 MHz

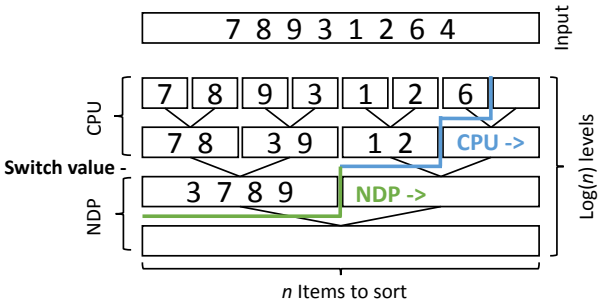


Figure 4: Implementation of the heterogeneous (CPU + NDP) mergesort.

remainder of this work has one to four memory channels and one or two CPU cores per memory channel. For practical simulation reasons, the used cache size shown in Table 1 is smaller than the one of a real CPU. However, we scaled appropriately simulation results to take into account for this limitation, by evaluating data set sizes and system propriety with respect to the ratio between the size of the actual cache and the size of the one simulated.

To evaluate the performance of applications running on the NDP architecture, a simulator was implemented. The tool simulates the interaction between the CPU (and in particular its memory hierarchy) and the NDPs. Figure 3 shows a high-level view of the simulator. It is implemented as a mix of C++ and hardware-description language (HDL) components, communicating through the Verilog Procedural Interface (VPI), which allows behavioral Verilog code to call C functions, and C functions to invoke standard Verilog system tasks.

Applications run native on the host, and all load and store instructions, including their data fields, as well as special instructions such as synchronization, are provided as input to the simulator. The caches and main memory modeled by the simulator hold therefore the actual application values, and in this way it is possible to verify the correct implementation of all aspects of the NDP hardware and the architecture, like synchronization and barriers. In fact, as the application running on the host works with the data values it retrieves from the simulator, an exact implementation of the simulator is essential to produce meaningful results. Memory allocations done on the host are mirrored in the simulator by using the same virtual and physical addresses and page layout, meaning that the translations supplied by the NDP software-library to the NDP-M are based on page table scans on the host.

The NDP-M is implemented in hardware and provides communication mechanisms between the NDP and software, and implements a TLB such that the NDP can work with virtual pointers. The NDP-software and the OS changes are implemented in a user-level software library. This library provides all the functionalities to allocate data structures, handle the address translation requirements for the NDP-M, communicate with the NDPs, provide synchronization between the CPU and the NDPs, etc. Since the memory in the simulator mirrors the memory on the host, NDP workloads can be implemented by using the host's virtual pointers. All actions done by the software side of the simulator are triggered or synchronized by the hardware clock, which makes the simulator cycle-accurate.

The simulator supports the availability on the host CPU of several cores, each with a private cache, a system bus, and a different number of memory channels and NDPs.

While the framework is general enough to simulate NDPs implemented either as general-purpose cores or workload-optimized cores, in this paper we focus on NDPs implemented in FPGA, interfaced to the host with the functionalities provided by the NDP-M.

4. SORT IMPLEMENTATION

We implemented a multi-threaded version of the mergesort algorithm, which operates on 16 B items, consisting of an 8 B index (to be sorted), and an 8 B pointer. The implemented mergesort is optimized for making use of all the computing resources of our platform, i.e., the multiple threads available on the CPU and the multiple NDPs.

Mergesort sorts a data set by recursively merging sorted subsets into a larger sorted subset, as shown in Figure 4. The algorithm starts merging two single items of an unsorted list into a sublist (single item list can be considered as already sorted). Then, it repeatedly merge the produced sublists to create a new sorted sublist. This continues until only one list of sorted elements exists.

As shown in Figure 4, the first iterations of the algorithm offer plenty of small merges that can be easily parallelized

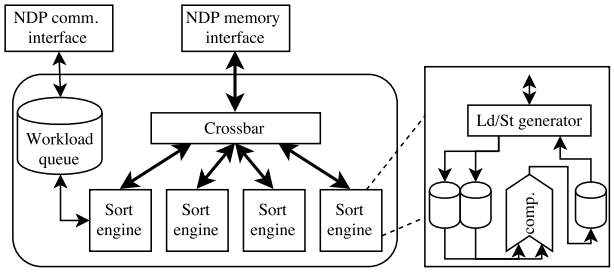


Figure 5: Block diagram of the workload-optimized merge core implemented in the NDP.

over multiple processing elements. However, in the latest iterations, the amount of straightforward parallelism is reduced. Therefore, for the lastest levels we implemented a parallel version of the merge step which is based on a divide and conquer method [6]. This method finds two items in the data sets as close as possible in value to each other, and as close as possible to the center of the data set. From these items the data sets can be split in two, and then merged in parallel. This obviously comes at a cost, and therefore the method is only used for the latest levels.

We developed a workload-optimized merge core in VHDL, able to do a single partial merge. Figure 5 shows the block diagram of the merge core implemented in the NDP. Since we are targeting future reconfigurable fabric, we assume the core to operate at a frequency of 500 MHz. The merge core does a single comparison every clock cycle, and therefore needs 32 B (16 B load and 16 B store) every cycle, utilizing 16 GB/s of main memory bandwidth. To be able to use all the available bandwidth, we used four merge cores, and connected them together by using a crossbar. The VHDL design also includes a controller which is in charge of receiving the workload messages from the CPU, distributing the workload to the four merge cores, and sending the results back to the CPU.

As explained in Section 3, we modeled the interaction between the CPU and the NDPs by using the load/store operations generated by the part of the merge program running on the CPU. Besides the load/store operations, applications can issue *stall* operations to the simulator, to be able to adjust the overall performance. In fact, by using this approach, we can achieve performance error of less than 5% with respect to measurements taken on a real POWER8 system running at 4.2 GHz. In particular, without loss of generality, we considered the single-core performance obtained when running the application with four threads, out of the available eight, as during our simulation campaign it allowed to achieve the highest performance on the POWER8. Table 1 summarizes the system specifications.

Our mergesort implementation adopts a heterogeneous approach: the small partial merges, having lots of data locality, are done on the CPU, while large partial merges are done by the NDP. The approach is represented in Figure 4, showing how the CPU and the NDPs can be work in parallel for solving the problem. We call *switch value* the minimum size of the sublists processed on the NDPs. This value is dynamically adjusted to balance the load between the CPU cores and the NDPs. The switch value is initialized at 128 items, and it increases or decreases after a certain time that the NDP work queues are full or almost empty, respectively.

The maximum value for the switch value is set to 2048, which corresponds to the sublist size for which the two sublists to be merged fully occupy the available cache. This maximum switch value corresponds to the size of the caches used in our simulator. A larger switch value would mean that a CPU core is working with data that can, by definition, not be in its cache, and thus has to come from main memory. Since the NDPs are already designed to make full use of the available memory bandwidth, the CPU will not improve performance, but only reduce the performance of the NDPs, which are much more efficient at sorting directly from main memory.

The workloads arrive at the NDP by means of the message interface discussed in Section 3, where every message contains the *virtual* pointers and item counts needed for a partial merge. A separate thread on the software side manages the sending and receiving of messages to and from the NDPs.

In case we use multiple NDPs to sort a single data set, every NDP gets an equal portion of the data to be sorted independently. Once the NDPs have completed their tasks, the CPU performs the last merges to create the final result.

5. ANALYSIS AND RESULTS

5.1 Speedup analysis

In this subsection, we analyze the theoretical speedup achievable by using a heterogeneous system including CPUs and NDPs. As example, we focus in particular on a system using a single CPU core and a single NDP. In the analysis, it is possible to distinguish between two cases. In the first case, the CPU is the bottleneck in the calculation, and the NDP is partially underutilized as waiting for the CPU results to complete. In this situation, the smallest and highest speedups are achieved when using the largest and the smallest switch value, respectively. As the switch value is adjusted dynamically, the resulting speedup will be in between these two limit values and it can be described by the following formula:

$$\frac{log_2(n)}{log_2(Max\_switch)} \leq Speedup \leq \frac{log_2(n)}{log_2(Min\_switch)} \quad (1)$$

in which n is the data set size. For a data set size of one mega items and the parameters in Table 1, the estimated speedup is therefore between 1.8 and 2.5, while for a data set of four mega items, the speedup is between 2 and 2.75. The potential speedup increases because with larger workloads more merge levels are implemented on the NDP, which is therefore more utilized.

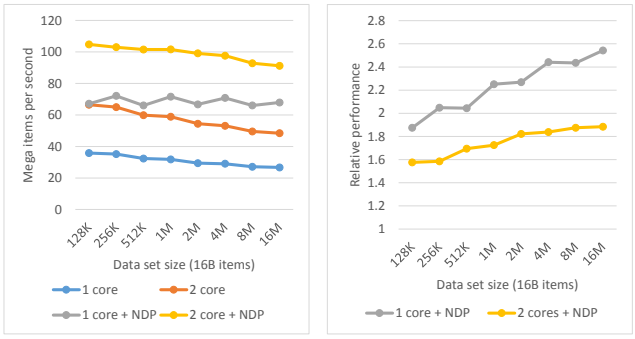
As the switch value has a maximum defined by the cache size, the work distribution becomes again unbalanced for very large data sets. In these cases, the NDP becomes the bottleneck in the implementation, and the overall performance will be bounded by the NDP performance, and in particular by its memory bandwidth. Without NDP, and for very large data sets, the performance would be limited by the CPUs memory channel bandwidth. In this situation, the speedup when using NDP can be described as:

$$\lim_{n \to \infty} Speedup = BW_{NDP}/BW_{Memory\_channel} \quad (2)$$

When using the values shown in Table 1, the speedup is approximately equal to 1.5.

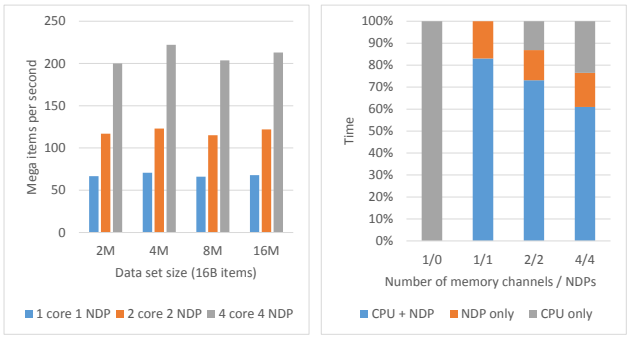
5.2 Results

In our experiments, we evaluate the performance of heterogeneous systems with different combinations of CPU cores



(a) (b)

Figure 6: (a) Absolute mergesort performance for four different systems configurations. (b) Relative performance with respect to configuration with no NDPs.



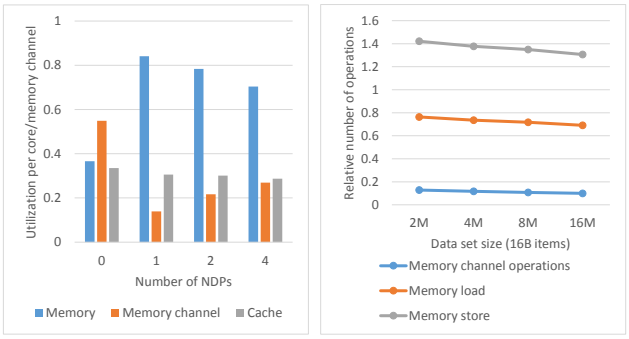
(a) (b)

Figure 7: (a) Performance for systems containing more memory channels and NDPs. (b) Time distribution of the heterogeneous usage of both the CPU and the NDP.

and NDPs. Figure 6a compares the absolute performance of four different systems configurations: one CPU core with no NDP, two CPU cores with no NDP, one CPU core with one NDP, and two CPU cores with one NDP. Figure 6b shows the relative performance with respect to the configurations with no NDPs.

When adding an NDP to the configuration with one CPU core, the performance increases and stays steady towards larger data sets, where the saw-tooth pattern is due to the discrete nature of the workload balancing mechanism. This reflects in an increasingly higher speedup, as shown in Figure 6b, up to a factor 2.5 for the largest data set, as estimated in Section 5.1. The measured speedup is, in fact, within 5% difference from the average speedup calculated considering the smallest and highest speedup values estimated in equation 1. For the configuration having two CPU cores and a single NDP, the trends are different. As in this configuration the NDP becomes the computation bottleneck, the performance drops for larger data sets, while the speedup flattens. For the largest data set, the switch value is at its maximum value for 42% of the time, clearly indicating an imbalanced workload distribution, resulting in a declining performance, and the speedup reaching its upper bound.

In Figure 7a, we show the performance results when using more NDPs, by keeping one NDP per core. As discussed, the partial results created by the NDPs, each one working on



(a) (b)

Figure 8: (a) The component utilization for various number of memory channels / NDPs, when sorting 16 mega items. (b) Relative number of operations generated by the single CPU core and single NDP configuration, with respect to a single CPU core configuration.

its own memory channel, are merged together by the CPU to create the final result. As a result of the parallelization process, the scaling is not perfect. We can observe a scaling factor of 1.75 when moving from one to two NDPs, and a scaling factor of 3.1 when moving from one to four NDPs. Since the performance is stable for the various data set sizes, it makes no differences whether we look at *strong* or *weak* scaling.

In Figure 7b, we show the portion of time in which both components (CPU + NDP) are working, and the time in which only one of them is active. It is clear that, when using more NDPs, a larger portion of time is spend in the final merge steps done by the CPU, which directly results in the above mentioned not perfect scaling.

In Figure 8a, we show the components' bandwidth utilization when using various numbers of NDPs. As it is possible to notice, when using NDPs, we can observe a lower memory channel utilization and a higher memory utilization. Figure 8b shows the relative number of read/write DRAM requests and the relative number of memory channel transactions generated by the single CPU core and single NDP configuration, with respect to a single CPU configuration. An important observation that can be made is that, by using the heterogeneous approach, the amount of data going through the memory channels is 10× lower, clearly indicating that the proposed approach reduces significantly the amount of data moved within the overall system, and, potentially, the amount of energy associated with it. The 30% increase in the number of memory stores, with respect to the single CPU system, is due to the extra write-back pass that it is introduced when switching from the CPU to NDP processing. The reduction in the number of loads is instead due to the fact that the NDP does not have a hardware managed cache, and can issue a 32 B store directly to the memory system, without first having to load the data from memory, as it is the case for a cacheline-based architecture as the CPU.

Figure 9 shows the size of the workload queue at the NDPs as well as the switch value, over time, using two CPU cores and a single memory NDP, sorting a data set of four mega items. Every time the queue size reaches its maximum (set for this experiments to 32), the switch value is doubled, until the maximum of 2048 is reached. If the queue is almost empty, the switch value is divided in half. The occupation

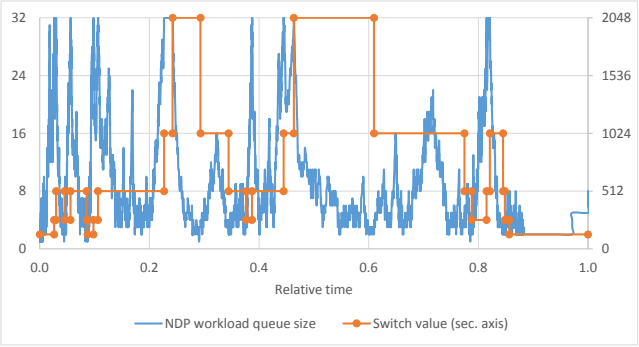


Figure 9: NDP work queue size and switch value over time, when sorting four mega items using two CPU cores and a single NDP.

of the queue is evaluated every 32 workload submissions to the NDP, to let the previous switch take effect and avoid increasing or decreasing the switch value too fast.

5.3 Power analysis

We synthesized our NDP hardware for a (medium sized) Xilinx UltraScale XCKU060 device, resulting in a design utilizing 3500 LUTs, 3250 registers, and 30 BRAMs. To estimate power consumption, we used the *Xilinx Power Estimator* [21] and set the switching activity to 100%, resulting in 0.9 W of dynamic power and 0.6 W of static power. The NDP-M, handling the integration of the NDP with the system, is estimated to use 4 W, giving the entire memory side (memory-side chip + DRAMs) a 20% increase in power [14]. This results in a power budget of our entire proposal, using four memory channels, of 22 W. A bare, eight core, four memory channels, POWER8 system uses 400 W, where the CPU contributes for about 200 W [14] [10]. In such a system, our proposal would adds 6% power at system level. Depending on the data set size and configuration, the performance increase of up to 1.8 or 2.6 times over a similar system without NDPs would therefore results in a factor of up to 1.7 and 2.5 of energy-to-solution saving, respectively.

6. CONCLUSION

Sorting big data represents an important problem in many application fields. New architectures and solutions have been recently investigated, to support efficiently this operation on large data set. In this work, we explored a heterogeneous approach to implement mergesort on an architecture composed on CPUs and near-data processors. We showed how with a careful scheduling of the tasks on our architecture it is possible to achieve up to 2.5 of performance speedup, and up to 2.5× energy reduction with respect to an only-CPU based system.

7. REFERENCES

[1] J. Ahn, S. Hong, S. Yoo, et al. A Scalable Processing-in-memory Accelerator for Parallel Graph Processing. In *Proceedings of ISCA '15*, June 2015.

[2] E. Azarkhish, D. Rossi, I. Loi, et al. High performance AXI-4.0 based interconnect for extensible smart memory cubes. In *Proceedings of DATE '15*, Mar. 2015.

[3] R. Balasubramonian, J. Chang, T. Manning, et al. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro*, 34(4), July 2014.

[4] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: An Improved Architecture for Data-Intensive Applications. *IEEE Micro*, 30(1), Jan. 2010.

[5] L. Chen, X. Huo, and G. Agrawal. Accelerating MapReduce on a Coupled CPU-GPU Architecture. In *Proceedings of SC '12*, Oct. 2012.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, et al. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[7] D. Diamantopoulos and C. Kachris. High-level synthesizable dataflow MapReduce accelerator for FPGA-coupled data centers. In *Proceedings of SAMOS '15*, July 2015.

[8] B. Falsafi, M. Stan, K. Skadron, et al. Near-Memory Data Services. *IEEE Micro*, 36(1), 2016.

[9] M. Gao, G. Ayers, and C. Kozyrakis. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *Proceedings of PACT '15*, Oct. 2015.

[10] H. Giefers, R. Polig, and C. Hagleitner. Accelerating arithmetic kernels with coherent attached FPGA coprocessors. In *Proceedings of DATE '15*, Mar. 2015.

[11] HMC Consortium. "http://www.hybridmemorycube.org/", 2017.

[12] H. P. Hofstee, G. C. Chen, F. H. Gebara, et al. Understanding system design for Big Data workloads. *IBM Journal of Research and Development*, 57(3/4), May 2013.

[13] T. Honjo and K. Oikawa. Hardware acceleration of Hadoop MapReduce. In *Proceedings of IEEE BigData '16*, Oct. 2013.

[14] IBM. System Energy Estimator. "http://www-912.ibm.com/see/EnergyEstimator", 2017.

[15] R. Nair, S. F. Antao, C. Bertolli, et al. Active Memory Cube: A processing-in-memory architecture for exascale systems. *IBM J. of Research and Development*, 59(2/3), Mar. 2015.

[16] S. H. Pugsley, J. Jestes, R. Balasubramonian, et al. Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads. *IEEE Micro*, 34(4), July 2014.

[17] W. J. Starke, J. Stuecheli, D. M. Daly, et al. The cache and memory subsystems of the IBM POWER8 processor. *IBM J. of Research and Development*, 59(1), Jan. 2015.

[18] J. Stuecheli, B. Blaner, C. R. Johns, et al. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development*, 59(1), Jan. 2015.

[19] P. Trancoso. Moving to Memoryland: In-memory Computation for Existing Applications. In *Proceedings of CF '15*, May 2015.

[20] E. Vermij, C. Hagleitner, L. Fiorin, et al. An Architecture for Near-data Processing Systems. In *Proceedings of CF '16*, May 2016.

[21] Xilinx. Xilinx Power Estimator (XPE). "http://www.xilinx.com/products/technology/power/xpe.html", 2016.

[22] S. F. Yitbarek, T. Yang, R. Das, et al. Exploring specialized near-memory processing for data intensive operations. In *Proceedings of DATE '16*, Mar. 2016.