Moving workloads to a better place

Optimizing computer architectures for data-intensive applications

Vermij, Erik

DOI
10.4233/uuid:9976d272-a596-4ad0-ab9e-de230cd0aba3

Publication date
2017

Document Version
Final published version

Citation (APA)
Vermij, E. (2017). *Moving workloads to a better place: Optimizing computer architectures for data-intensive applications*. [Dissertation (TU Delft), Delft University of Technology].
https://doi.org/10.4233/uuid:9976d272-a596-4ad0-ab9e-de230cd0aba3

MOVING WORKLOADS TO A BETTER PLACE

OPTIMIZING COMPUTER ARCHITECTURES FOR DATA-INTENSIVE APPLICATIONS

MOVING WORKLOADS TO A BETTER PLACE

OPTIMIZING COMPUTER ARCHITECTURES FOR DATA-INTENSIVE APPLICATIONS

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op dinsdag 4 juli 2017 om 15:00 uur

door

Erik Paul VERMIJ

Master of Science in Embedded Systems,
Technische Universiteit Delft, Delft, the Netherlands,
geboren te Gouda, Nederland.

Dit proefschrift is goedgekeurd door de

promotor: Prof. dr. K.L.M. Bertels
copromotor: dr. C. Hagleitner

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof. dr. K.L.M. Bertels	Technische Universiteit Delft, promotor
dr. C. Hagleitner	IBM Research – Zurich, copromotor
Onafhankelijke leden:	
Prof. dr. ir. C. Vuik	Technische Universiteit Delft
Prof. dr. H. Corporaal	Technische Universiteit Eindhoven
Prof. dr. ir. L. Eeckhout	Universiteit Gent
Prof. dr. ir. O. Mutlu	ETH Zurich
Prof. dr. ir. A.-J. van der Veen	Technische Universiteit Delft, reserve lid
Overige leden:	
Prof. dr. H.P. Hofstee	Technische Universiteit Delft

Keywords:	Square Kilometre Array, computer architecture, near-data processing, high-performance computing
Printed by:	Gildeprint
Front & Back:	Île Sainte-Marie, Madagascar

ISBN 978-94-6186-821-3

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

CONTENTS

SUMMARY

The performance of supercomputers is not growing anymore at the rate it once used to. Several years ago a break with historical trends appeared. First the break appeared at the lower end of worldwide supercomputer installations, but now it affects a significant number of systems with average performance. Power consumption is becoming the most significant problem in computer system design. The traditional power reduction trends do not apply any more for the current semiconductor technology, and the performance of general-purpose devices is limited by their power consumption. Server and system design is in turn limited by their allowable power consumption, which is bounded for reasons of cost and practical cooling methods. To further increase performance, the use of specialized devices, in specialized server designs, optimized for a certain class of workloads, is gaining momentum. Data movement has been demonstrated to be a significant drain of energy, and is furthermore a performance bottleneck when data is moved over an interconnect with limited bandwidth. With data becoming an increasingly important asset for governments, companies, and individuals, the development of systems optimized on a device and server level for data-intensive workloads, is necessary. In this work, we explore some of the fundamentals required for such a system, as well as key use-cases.

To highlight the relevance of the work for a real-world project, we analyze the feasibility of realizing a next-generation radio-telescope, the Square Kilometre Array (SKA). We analyze the compute, bandwidth and storage requirements of the instrument, and the behavior of various important algorithms on existing products. The SKA can be considered to be the ultimate big-data challenge, and its requirements and characteristics do not fit current products. By putting the SKA requirements next to historical trends, we show that the realization of the instrument at its full capacity will not be achievable without a significant effort in the development of optimized systems.

In order to make steps towards the successful realization of the SKA, we develop a custom hardware architecture for the Central Signal Processor (CSP) subsystem of the SKA. The CSP is dominated by high input and output bandwidths, large local memories, and significant compute requirements. By means of a custom developed ASIC, connected to novel high-bandwidth memory, the proposed solution has a projected power-efficiency of 208 GFlOPS/W, while supporting all CSP kernels in a flexible way. This is an example of how optimized systems can drive down the energy consumption of workloads, and thereby aid the realization of projects with non-conventional requirements.

To enable improving the efficiency of a variety of workloads, we developed a hardware architecture supporting arbitrary processing capabilities close to the main-memory of a CPU. This follows the theme of 'near-data processing', offering foremost high bandwidths and reduced data movement. The effort is driven by the two main observations that 1) processing capabilities should be workload-optimized, and 2) a focus on data and memory is necessary for modern workloads. The architectural description includes

data allocation and placement, coherence between the CPU and the near-data processors (NDPs), virtual memory management, and the accessing of remote data. All data management related aspects are implemented with existing OS level NUMA functionality, and require only changes in the firmware of the system. The other three aspects are realized by means of a novel component in the memory system (NDP-Manager, NDP-M) and a novel component attached to the CPU system bus (NDP Access Point, NDP-AP). The NDP-M realizes coherence between CPU and NDP by means of a fine- and coarse-grained directory mechanism, while the NDP-AP filters unnecessary coherence traffic and prevents it from being send to the NDPs. Address translation is implemented by the NDP-M, where the Translation Lookaside Buffer (TLB) is filled and synchronized via a connection with the NDP-AP. The NDP-AP is furthermore the point where remote data accesses from the NDPs enter the global coherent address space. Several benchmarks, including a graph-traversal workload, show the feasibility of the proposed methods.

The evaluation of the architecture as well as the evaluation of various types of NDPs required the development of a novel system-simulator. The developed simulator allows the evaluation of NDPs developed in a hardware description language placed in a simulated memory system. Arbitrary applications making use of the simulator feed the simulated memory system with loads and stores, and can control the NDPs. It is also possible to evaluate general-purpose NDPs running software threads. The complex system level interactions concerning coherence and remote data accesses are modeled in detail and provide valuable insights.

Two relevant benchmarks for both high-performance computing and data-intensive workloads are the High-Performance Conjugate Gradient (HPCG) benchmark, and the Graph500 benchmark. They implement a distributed multi-grid conjugate gradient solver and a distributed graph breadth-first search, respectively. Both benchmarks are implemented on the proposed architecture containing four NDPs, consisting of very small and power-efficient cores. By exploring both parameters of the architecture, as well as various software optimizations, we boost the performance of both benchmarks with a factor 3x compared to a CPU. A key feature is the high-bandwidth and low-latency interconnect between the NDPs, by means of the NDP-AP. The cacheability of remote data at the NDP-AP enables the fast access of shared data and is an important aspect for Graph500 performance. The use of user-enhanced coherence boosts performance in two ways. First, guiding the coarse-grained coherence mechanism at the NDP-M eliminates much of the required coherence directory lookups. Second, allowing remote data to be cached in NDP hardware-managed caches, improves data locality and performance, at the expense of more programming effort to manually maintain coherence.

A typical operation in big-data workloads is the sorting of data sets. Sorting data has, by nature, phases with a lot of data locality, and phases with little data locality. This opens up the intriguing possibility of heterogeneous CPU and NDP usage, where the two types of devices sort the high-locality, and low-locality phases, respectively. The CPU makes optimal use of its caches, while the NDP make optimal use of the high bandwidth to main memory. This is evaluated when considering a workload-optimized merge-sort NDP, and we obtain up to a factor 2.6x speedup compared to a CPU-only implementations. Given the very low power of the workload-optimized NDP, the overall energy-to-solution improvement is up to 2.5x.

1

INTRODUCTION

1

1.1. BACKGROUND

We live in exciting times. Computers help doctors with the correct diagnoses of cancer patients, and with the creation of optimal treatment plans for these patients, increasing the quality of life for many people around the globe [1]. Grandparents can connect with their grandchildren by means of a variety of mobile applications and social media. Climate research, as well as research into novel energy sources, drives our understanding of how to keep our planet inhabitable for the upcoming generations. These opportunities are driven by the ever growing capabilities of computer systems, enabled by continued research into the field of semiconductor technology and computer architecture. These growing capabilities of computer systems are typically explained as being a consequence of 'Moore's law'. This famous law, described in a 1965 paper [2], states that the number of transistors we can put on a chip doubles every year, and that we can furthermore realize this for the same cost. This turned out to be true for some decades following 1965. Moore's law does not imply anything about the performance (the amount of operations per second) we are able to realize with computers build with these transistors, but historical trends have shown that the performance developments follow the same trend of doubling every year. However, things have changed.

Computer performance development does not follow the historical trends anymore. In Figure 1.1 we show the achieved performance of the 500 fastest supercomputers in the world, the TOP500 [3]. Shown is the cumulative performance, the performance of the number one system, and the performance of the number 500 system. When looking at the slowest system of the list, we see a clear break with historical trends around the year 2008. From that year onwards, the performance grows at a constant slower rate. The same holds for the cumulative performance, but from the year 2013 onwards. Although supercomputers alone do not capture the full extent of the computer systems business, technologies often get introduced in, and developed for, such systems.

We have entered an era where the performance of computers, on the device as well as the system level, is bounded by power consumption. Making a device faster will make it generate too much heat to be able to cool it down with practical cooling methods. At this moment, a CPU already has a significantly higher power density (Watts / cm^2) than a hot plate [4]. Making a complete system faster will result in unreasonably high power bills. The estimated accumulated annual electricity bill of all datacenters in the USA in the year 2020 is an impressive 13 billion dollars [5]. Moving data around in a computer system is the main drain of energy. It is estimated that moving two operands from external main memory to a functional unit is a factor 100x up to a factor 1000x more expensive than doing an operation (e.g. multiplication) on that data [6] [7]. The performance of computers is thus, when put in a simple statement, limited by data movement.

All three examples in the introduction have something in common: they are driven by data. Intelligent healthcare systems can ingest all available literature on a topic and process this into meaningful conclusions. Social media collect our communications, our online browsing behavior, our whereabouts and many more things, typically aimed at the creation of personalized advertisements. Research into climate change can only be performed when large amounts of (historical) measurements are available.

In Figure 1.2 we show a generic overview of data growth in the last decade. The exponential growth is clearly visible. In Figure 1.3 we show the growth in available data

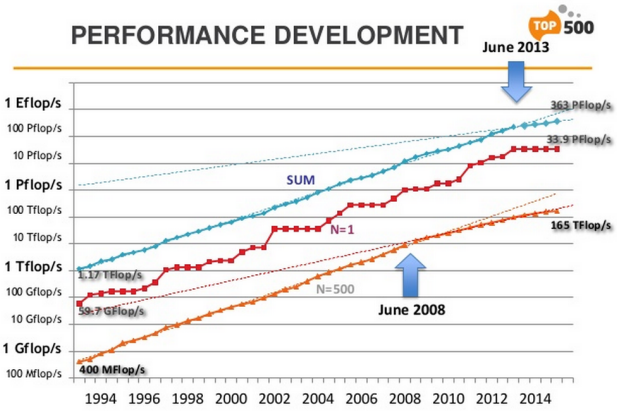


Figure 1.1: TOP500 historical performance including trend lines. Clearly visible are the two breaks with historical trends in 2008 and 2013 for the slowest system and the cumulative performance of all the systems in the list, respectively. Image courtesy: TOP500

for the specific field of bio-informatics. This field is a clear example of where the data volumes grow much faster than any other trend we have seen in the history of computer architecture. Every seven months the amount of available DNA data doubles [8], which is about twice as fast as the (historical) initial growth rate of Moore's law.

Bio-informatics is a clear example of a field being overflown with data. Another example of a field experiencing an 'astronomical data deluge', is radio-astronomy. Existing radio telescopes create enormous amounts of data, requiring novel real-time compute, back-end compute, and storage solutions, to create scientific insights [9]. The Square Kilometre Array (SKA) [10] will produce orders or magnitude more data, and is a clear example of an instrument that will push the boundaries of computer system development [11].

In the remainder of this chapter we will explain the drivers and mechanisms behind the declining trends in computer performance, as well as the trends that counter those effects. The explanation will be started at the device level, reasoning about the reasons single devices can not become much faster. From there the step towards a system-level analysis is made. By including various types of processing elements in a single system, the overall performance of the systems kept increasing for workloads fitting the new types of processing elements. By an extensive analysis of supercomputers, we however show that the solutions being pursued here are not the ones benefiting modern, data intensive, workloads. Novel, data-oriented, and well-integrated solutions, are likely to push computer system performance forward in the next decade.

1.2. DEVICE LEVEL ANALYSIS

To understand the performance limitations in modern computer systems, we need to start at the device level.

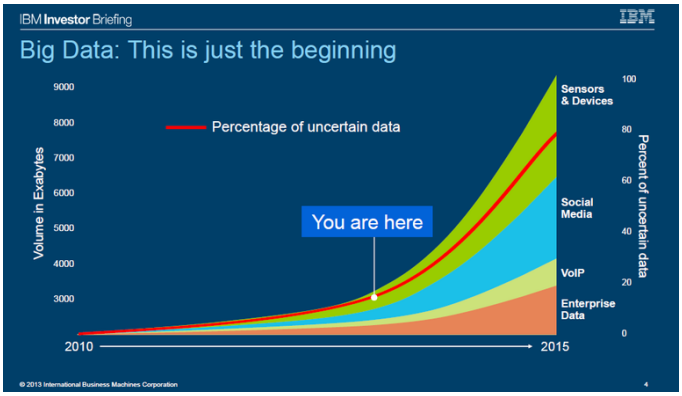


Figure 1.2: A view of the exponential growth in data volumes for a variety of fields (source: [12]).

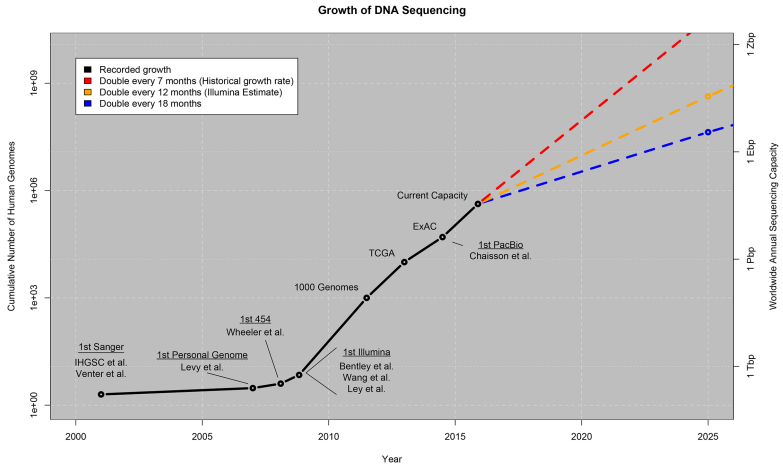


Figure 1.3: Illustrating the amount of available genomic data, growing much faster than Moore's law (source: [8]).

1.2.1. ENDING OF MOORE'S LAW

The initial growth rate of Moore's law has held true for many years, but has slowed down in recent decades. Although it depends on the exact type of technology we are looking at, in the 80s a drops towards a 3x increase per four years became apparent [13]. Around the year 2000, this slowed down to a 2x per two years, and at the moment we are experiencing a 2x per three years. Another aspect of Moore's law is the cost per transistor. Due to the push-back (if not cancellation for the foreseeable future) of developing 450 mm wafer technology [14], the cost per transistors will also not go down with the same rate as before.

1.2.2. ENDING OF DENNARD SCALING, THE POWER WALL, AND DARK SILICON

A transistor becomes a factor $S = 1.4$ smaller with every process generation, and since we are talking about a 2D structure, we can put $S^2 \approx 2x$ more transistors on a chip. This does not imply anything about performance, but trends have shown that performance increases with the same rate as the number of transistors on a chip for a long time. The reason is that, when transistors become smaller, they can be operated at a higher frequency, and thus realize a higher performance. The power used by such a transistor would not increase with respect to previous generations, as a smaller transistor has a smaller capacitance and can operate at a lower voltage. This effect is known as 'Dennard scaling' [15], and is the drive behind ever increasing frequency of processors, until we hit the 'power wall' early 20th century. With increasing leakage currents due to thinner insulation layers, and voltages hitting the lower bound at which a transistor still functions, the power per transistor no long dropped at the same rate as before. The consequence of the increase of power is popularized as 'dark silicon' [16]. In this work it is shown that the power per transistor only drops with a factor S per generation nowadays (and this is uncertain for the future), and since we get S^2 transistors, the power increase for the chip is S, at constant frequencies. As the peak power of a chip is limited by practical cooling methods, this results in a new reality in which for every process generation, we can switch on a factor S fewer transistors compared with the previous generation, resulting in a increasing fraction of mandatory 'dark', or switched-off, silicon.

Figure 1.4 illustrates that, from the year 2000 onwards, the power per processor is capped between a 100 and 200 watts, and the frequency is capped at two to four GHz. This effect is complemented by the 'dark silicon' effect, illustrated in Figure 1.5, showing how, for a process shrinkage of $S = 1.4$, we get double the transistors or cores, but we have to switch off half of them to stay within the power budget.

1.2.3. SOLUTIONS AT THE DEVICE LEVEL

To still realize performance improvements, although not at the same rate as before, industry and academia have explored various directions.

PARALLELISM AND BIGGER CACHES

Although the practical operating frequency of transistors no longer improves from one generation of semiconductor technology to the next, we still get more transistors with every generation. To make use of all the available transistors, CPUs started to get more

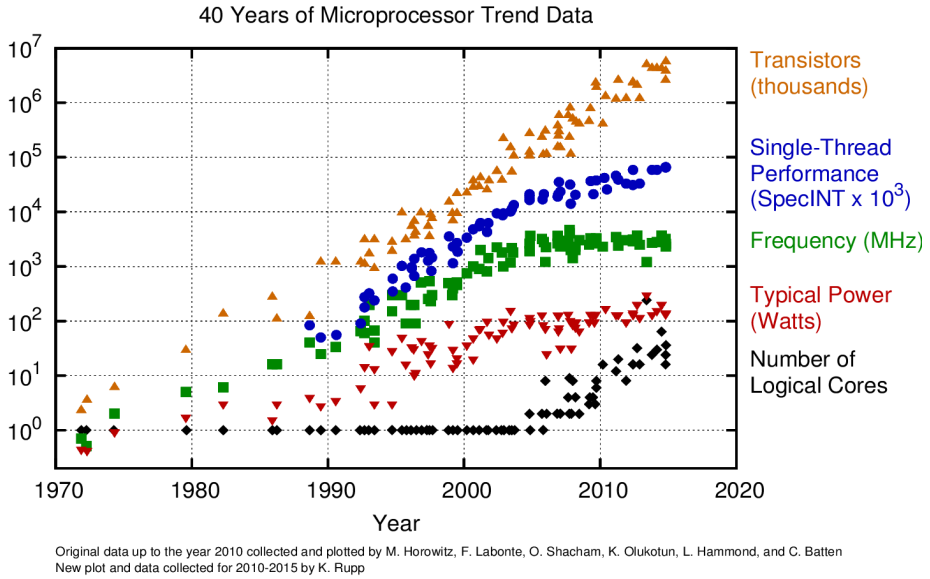


Figure 1.4: The effects of the 'power wall': processors have hit the limit of power usage, and the frequency cannot increase further because of that. (Image source: [17])

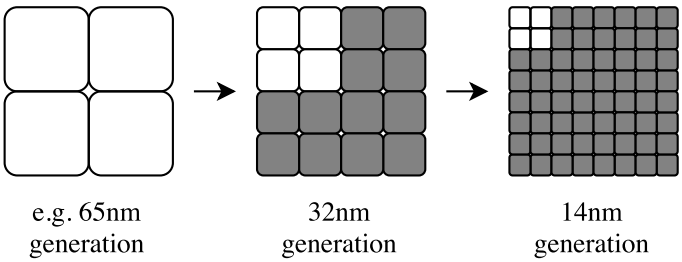


Figure 1.5: Effect of the 'power wall': for every process generation, we have to switch-off a larger fraction of the transistors to stay within the power budget, even at constant frequencies. This is popularized as 'dark silicon' [16]. (image based on: [18])

cores from 2003 onwards. This is illustrated by the trend lines in Figure 1.4. Using more cores increased the performance of the device, for flat frequencies. However, due to the dark silicon problem sketched above, between generations, the frequency still needed to decline, or become variable based on *turbo* and *sleep* modes, to keep the power usage within the budget [19]. Another observable trend is to use the available area for more and larger caches. Although caches use power as well, their footprint is smaller than a processing core, thus easing the problem while making good use of the area.

The so-called many-core architectures, like GPUs and the Xeon Phi, take this concept to the extreme. By deliberately using many weak cores, parallelism is favored over single core performance. Although these products are sometimes regarded as being revolutionary, their working is not fundamentally different from CPUs.

SPECIALIZATION WITHIN A DEVICE

Instead of using all transistors for fully programmable general-purpose cores, sacrificing area for specialized pieces of logic can improve performance and power efficiency. In case they are not used, the power budget can be used by the general-purpose cores, running at a high frequency. When an application can make use of the specialized logic, it will execute much faster. An example of this concept from academia is the 'conservation cores' [20], but many variations, including reconfigurable ones like 'VEAL' [21], exist. The complexity of implementing and using ideas like this have limited their realization.

Industry examples of the same concept, but at a coarser level, is the *POWER edge of network chip* [22]. This chip contains several types of specialized accelerators targeting various task in the area of network traffic processing, like packet inspection. Also in CPUs, specialized logic is being used to accelerated industry standard functionality that will not change for the foreseeable future [23] [24]. Examples of this are encryption (e.g. AES, random number generation) and compression (e.g. gzip). For these algorithms the gains are considered to outweigh the complexity and investment of implementing them. The IBM Cell processor [25] was the first processor to implement two types of general-purpose cores on the same chip, making optimal use of the available area to accelerate foremost multi-media applications. Another example of this is the big.LITTLE technology from ARM [26], implementing several big cores as well as many small cores on the same processor.

True application-level specialization can come from the use of FPGAs. These devices allow complete reconfiguration and can thus be optimized for the task at hand. They can however not easily be used in a stand-alone fashion, and must either be integrated in a system-on-chip [27], or as a separate device connected to a CPU, as will be discussed later in this chapter.

1.3. NODE-LEVEL ANALYSIS

With the processing capabilities of devices continuing to increase, other problems, at the node and the system level, become apparent. In this section we will first discussed two trends regarding memory bandwidths and the handling of data, followed by an analysis of supercomputers.

1.3.1. THE MEMORY WALL

In 1995 it was already established that computer performance is growing exponentially faster than bandwidth, resulting in the so called 'memory wall' [28]. This means we cannot get data fast enough into the device to 'feed' the compute elements, reducing the utilization. As an example, for NVIDIA's® Tesla® product line, the bandwidth-to-compute ratio has worsened by a factor 2.15 between 2008 and 2013 (C870 to K40) [29]. Similar trends will be visible for basically every other type of device. Furthermore, the memory latency has been shown to lag bandwidth quadratically [30], meaning we also have to, relatively, wait longer for the data to arrive.

One of the origins of the memory wall is a consequence of scaling ratio between the area and perimeter of the chip. Multicore performance scales with the area, whereas I/O generally has to escape a chip or package at its perimeter. Furthermore, the increase in pin count does not keep up with the increase in transistor count [31]. While many of the interfaces in a computer changed from parallel to serial in the last decades, the de-facto standard DRAM memory technology still uses a parallel interface. With increasing throughput and frequencies, this results in complex routing and timing issues [32]. It is however not just a technological challenge. Economics and product placement also play a role in the features manufacturers focus on. The CPUs from IBM typically have a much better bandwidth-to-compute ratio as well as a higher absolute bandwidth compared to Intel CPUs, which tend to focus more on compute performance [33] [34]. Novel serial memory technologies like the Micron *Hybrid Memory Cube* (HMC) [35], or 2.5D stacked DRAM technologies like *High Bandwidth Memory* [36], offer much higher bandwidths than current mainstream technologies. First products utilizing these technologies are available already [37], creating a step function in absolute memory bandwidth. The decade old trend that created the memory wall is however not turned around that easy, and we see only a stabilization of the compute-to-bandwidth ratio for this generation of products. Furthermore, these new memory technologies are not yet found in CPUs, but are foremost used in GPUs and some FPGA solutions.

1.3.2. DATA-LOCALITY AND DATA-MOVEMENT PROBLEMS

Besides raw bandwidth, difficulties arise regarding the data-movement management in a system. Data movement is necessary to bring operands to functional units doing operations. This is a fundamental property we cannot do much about. To limit data movement, *data-compute affinity*, or *data-locality*, is essential. This means we should do our computations as close as possible to the data, or the other way around, store the data as close as possible to the place we are doing the computation. When looking at a simple operation, e.g. an addition, it is clear we need to combine two operands together. These two operands can however very well come from different data sets, stored in different physical memories, frustrating the data-compute affinity. When looking at parallel compute kernels, many data sets can be accessed, of which some are private to each kernel execution, while others are shared by all kernels in parallel, that are possibly running on various distinct compute elements (e.g. multiple CPU cores, multiple CPUs). Optimizing data-locality in such a situation becomes increasingly hard. Data sets can be allocated and initialized at various points in an application, some data sets are used only for one compute kernel, while others live throughout the entire applications life. Opti-

mizing data-locality is essential to limit data movement, but also a practical as well as fundamental problem due to how computers and applications work.

With ever more complex multi-socket hardware-managed memory hierarchies, it is increasingly hard to understand where the data resides [38]. In an opposing approach, with the introduction of different types of processing devices in a node (discussed later), separate memory spaces were added, needing explicit data management from the programmer. With memory bandwidth and interconnect bandwidths not keeping pace with the processing capabilities, this data locality management (or the lack of the ability thereof), becomes increasingly problematic. With the recent introduction of the IBM and NVidia NVLink technology [39], and the recent announcement of the IBM POWER9 interface technology [40], extra steps are taken to increase the bandwidth between devices, and to ease the management of data locality. Besides the problems associated with handling data, moving data costs a lot of power. Moving data from one side of a chip to the other side is already an order of magnitude more expensive than doing a computation on that data. This factor is estimated to increase towards a factor between a 100x and a 1000x more expensive when considering loading the value from external DRAM [6] [7]. With the already discussed problems regarding power usage, data movement should thus be limited whenever possible.

1.3.3. HETEROGENEITY

Device-level specialization is a method of increasing the performance of a device beyond the traditional trends. This can not only be exploited within a device, but also at the node level. Last decade, this concept was popularized by adding specialized coprocessors, often called *accelerators*, to a node, to boost performance. These coprocessors are typically highly parallel, sacrificing single thread performance for running many threads. They often offer a high floating point compute performance, and are thus meant to run the compute intensive part of an application. This heterogeneous approach is often celebrated for its performance and power efficiency.

Adding a different type of compute element to a system, thus making it heterogeneous, is not without difficulties. First, coprocessors typically have a different programming paradigm and applications must be explicitly optimized to make use of them. With the availability of various types of coprocessors, and the release of new product generations every couple or years, this is a serious burden on programmers [41]. Especially for large code bases developed over many years, with the intent of being portable and maintainable for many more years, explicit optimizations for one kind of coprocessor are unpopular. Although, e.g., 'general purpose GPU' programming has been around for over a decade, only now hardware-agnostics parallel programming paradigms like OpenMP (version 4+) [42] are supporting pragma-based usage of coprocessors in a meaningful way [43] [44]. Second, coprocessors are typically connected to a CPU by means of a relatively slow bus, intensifying the data transportation problem mentioned in Section 1.3.2. Furthermore, the level of system-integration is only improving slowly. Generic hardware-managed mechanisms to allow sharing the same view on data between CPU and coprocessor are still not in place today (but are under development [45]).

1.4. WORKLOAD-OPTIMIZED SYSTEMS

At the device level, performance improvement due to frequency scaling has run out of steam, and last decade we have seen an increase in the core count to improve performance. Taking this concept to the extreme, by introducing heterogeneity through adding highly-parallel coprocessors such as GPUs to a node, boosted the performance well into this decade. However, the use of standardized coprocessors has its limits and the approach does not necessarily fit every application of today. To further improve the performance of relevant applications, we need to optimize a computer system for the task at hand. This analysis is illustrated in Figure 1.6, clearly showing the steps from device scaling, to core scaling, towards heterogeneous workload-optimized systems. Workload optimization can be achieved by using reconfigurable fabric like FPGAs, as already mentioned in Section 1.2.3, attaching them to a CPU by means of fast interconnects. One step further is the use of application-specific integrated circuits (ASICs), to create another step function in performance and power-efficiency, at the expense of flexibility. The customization of nodes and the resulting performance increase, will also improve the power-efficiency of large supercomputers. By using fewer, but stronger, nodes, overheads in e.g. the power supply [46] get smaller, and less inter-node data transport is required.

There are several industry examples of this trend. Microsoft is using reconfigurable logic in their data-centers to push the performance and power-efficiency of their search-related infrastructure [47]. Google recently announced a (partly) custom motherboard to design a complete compute node specific for their needs [48]. To boost their performance and power-efficiency in the field of deep-learning, Google furthermore announced an in-house developed ASIC [49], delivering a 15x-30x speedup compared to commodity hardware [50]. With the introduction of the CAPI interface [51], and the announcement of OpenCAPI [45], the customization of computer systems has taken flight. Custom developed coprocessors can now coherently be attached to a high-end CPU, and be used directly from a user-level application, and the already mentioned upcoming bandwidth technologies in Section 1.3.2 will tackle bandwidth bottlenecks between various devices. By utilizing these concepts, systems will be tailored to the workload at hand, and will excel in both performance and power-efficiency as compared to general-purpose solutions.

1.5. SUPERCOMPUTER ANALYSIS

Figure 1.7 shows the measured performance of the number one systems of the TOP500 for the past decade. Note we only include new number one systems: including the number one from every list would make the figure less clear, while not changing the result. It can be seen that on a logarithmic scale, this dataset fits a straight line, which goes back all the way back to 1993, as also shown in Figure 1.1. The performance of the number one systems increases with roughly a factor of two every year.

1.5.1. POWER EFFICIENCY LAGS BEHIND PEAK PERFORMANCE

Figure 1.7 shows, in flops per Watt, the power efficiency of the system. Also in this case, we can observe a straight line. When looking at the performance growth of the number

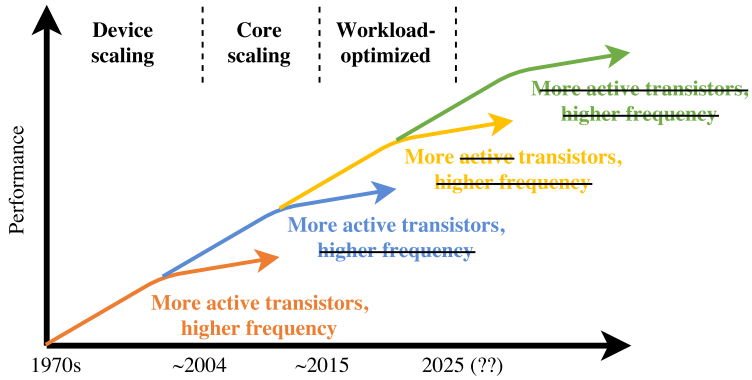


Figure 1.6: Illustrating the steps taken in this chapter, from device scaling, to core scaling, towards heterogeneous workload-optimized systems. (Based on images from H.P. Hofstee and M.L. Schmatz)

one supercomputer in the world, we observe a factor 1.9x in the last 25 year, very close to the *initial* rate of Moore's law. The slope of the power efficiency is lower than that of the peak performance, meaning that peak performance is growing exponentially faster than power efficiency. This means that new systems use exponentially more absolute power than their predecessors. This directly follows the arguments put forward in Section 1.2.2 and Section 1.2.1 about the end of Moore's law and end of power-efficiency ('Dennard') scaling. The performance increases with a factor 1.9x a year, while the amount of transistors per chip does not increase anymore with that rate, and neither does the increase in power-efficiency per used transistor. The performance increase can thus only be realized by using more devices, which use more power. Although the shown numbers are obtained when running the LINPACK benchmark, it has been shown they can be generalized for general-purpose workloads [52], meaning power bills are becoming higher and cooling will become more impractical. As device-level power consumption is limited at around 200 Watt (considering air-cooling), as discussed in Section 1.2.2, the total power usage of a supercomputer is limited as well, although the bound is less strict. A typical limit used is 20 to 30 mega Watt [53], comparable to a small town. When the practical absolute power usage limit is reached in the not so distant future, the growth in peak performance will have to slow down to follow the trend in power efficiency.

A version of the TOP500 list aiming at understanding the developments in power-efficiency is the Green500 [54], which shows the same systems, but now sorted on power-efficiency. The interpretation of this list is however difficult, as the top 10 contains both small and very large systems. Number one on this list achieves 9462 GFlops/W at a total power budget of 349 KW. The first very large system, the *Sunway TaihuLight*, reaches 'only' 6051 GFlops/W at place four of the list, but has a total power budget of 15.3 MW, making it a very different beast compared to the number one system. Nonetheless, to some extent breaking with historical trends, we see very large, both homogeneous as well as heterogeneous systems, in the top 10, indicating that industry is capable of realizing large systems with a good power-efficiency, when running LINPACK. As on the device level and the node level, data transport is a major drain of energy in a supercomputer.

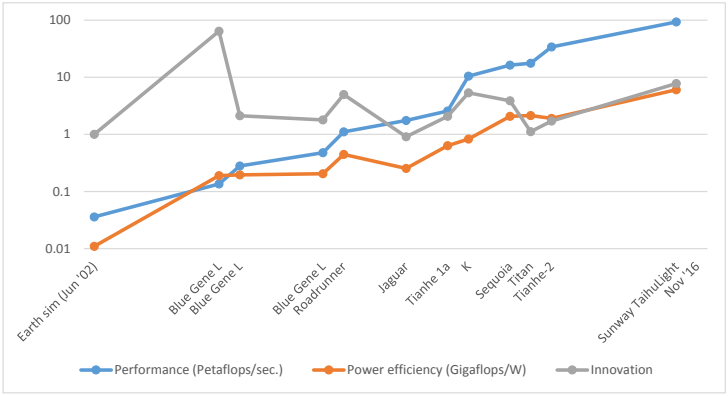


Figure 1.7: Trends for the new number one systems in the TOP500. *Innovation* is explained in Section 1.5.

An analysis presented in [46] shows that for an average set of workloads, around 15% of the total power budget is spent in the network between nodes.

1.5.2. INNOVATION IN NEW SUPERCOMPUTERS

Every new number one supercomputer is faster than its predecessor, and most of the time also more power efficient. We are interested in the level of innovation that every new generation brings, i.e., how much closer it brings us to high-performance and power-efficient computing. To investigate this, we create the *Innovation* metric, shown in Equation 1.1. *Perf*, *Eff* and n are the measured peak performance, power efficiency and system index, respectively. The innovation score is the product of the relative increase in performance with the relative increase in power efficiency. The power-efficiency of supercomputers is not strictly increasing, and therefore the *Max* operator is introduced, to evaluate the power efficiency of the system under study against the best power efficiency to date. The results are shown in Figure 1.7.

$$Innovation_n = \frac{Perf_n}{Perf_{n-1}} \times \frac{Eff_n}{Max(Eff_0 : Eff_{n-1})} \tag{1.1}$$

Over the past decade, an interesting observation is that, out of all the systems, the best scoring ones are all based on custom-designed hardware (Blue Gene/L, PowerXCell, K, Blue Gene/Q). The lower scoring systems are all based on commercial off-the-shelf (COTS) products, or are extensions of existing systems. This analysis shows us that although the use of COTS products is convenient for many reasons, they apparently do not give us the big steps forward we need. Research, out-of-the-box thinking, and novel approaches are needed to realize the next supercomputer which is not only faster but also more power efficient.

1.5.3. UTILIZATION AND LIMITATIONS FOR MODERN DATA-INTENSIVE WORKLOADS

The performance numbers for supercomputers shown so far are based on the high-performance LINPACK benchmark [55]. LINPACK factors and solves a large dense system of linear equations using Gaussian Elimination [56]. The dominant calculations in this algorithm are dense matrix-matrix multiplications and related kernels. This benchmark has been very relevant for a long period of time, but last decade the community started to realize the used kernels no longer capture the behavior of modern workloads. This gave rise to (at least) two additional commonly used 'industry-standard' benchmarks.

In the field of high-performance computing, methods based on differential equations have become the standard [56], and the typical characteristics of these methods needed to be captured in a benchmark. This became the HPCG (high-performance conjugate gradient) benchmark [57]. HPCG solves a sparse system of differential equations by means of a conjugate gradient method. The benchmark shows a variety of memory access patterns as well as inter-node communication patterns deemed relevant today.

Next to high-performance computing, the field of 'data analytics' made its entry. This field tries to create (economic) value from the growing amount of generated and stored data. To capture the typical behavior in data-analytic workloads, the Graph500 benchmark [58] was introduced. This benchmark does a breadth-first search in an undirected graph, and captures the memory access patterns and inter-node communication patterns essential for knowledge discovery based on unstructured data.

Where LINPACK performance foremost depends on the compute capabilities of a system, the other two benchmarks highly depend on the memory access characteristics as well, and can be considered to be 'data-intensive' workloads. In Figure 1.8 we show the utilization (fraction of compute capabilities used) for the top 10 supercomputers running LINPACK as well as HPCG. It can be observed that LINPACK reaches a very high utilization, between 75% and 95%. HPCG on the other hand, shows a very low utilization, with the *K* system scoring the highest with a 5.3% utilization, and the *Sunway TaihuLight* scoring the lowest with a 0.3% utilization. This is a clear indication that the systems we build today are very good at running a benchmark deemed less relevant, and are seriously under-utilized when running a more representative benchmark.

In Figure 1.9 we show the performance of the top 10 supercomputers running LINPACK, HPCG, and Graph500, normalized against the number one system (*Sunway TaihuLight*). For LINPACK, the results obviously decrease towards the number 10 in the list, as this is the index the list is sorted on. For HPCG we see very different results. Several systems score better than the number one, and until we reach number eight in the list, there is no decreasing performance trend at all, while for LINPACK, these systems already run 10-20x slower than the number one. For Graph500, unfortunately not all systems have an entry, but the analysis is very similar to HPCG. Also shown is the introduction date of the system. Striking are the *K* system and the *Sequoia* system, both performing equal or better than the *Sunway TaihuLight* system, while being five and four years older, respectively. This is a clear indication that, when looking at modern and relevant workloads, little progress has been made in the last years to improve performance. Apparently, the metric we focus on when developing new supercomputers, is

1

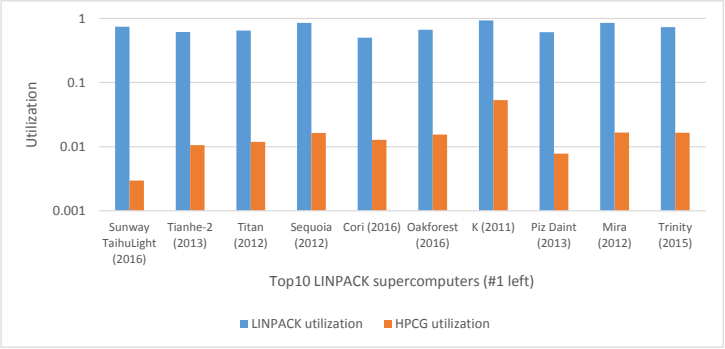


Figure 1.8: The utilization for the top 10 supercomputers, running the LINPACK and the HPCG benchmark.

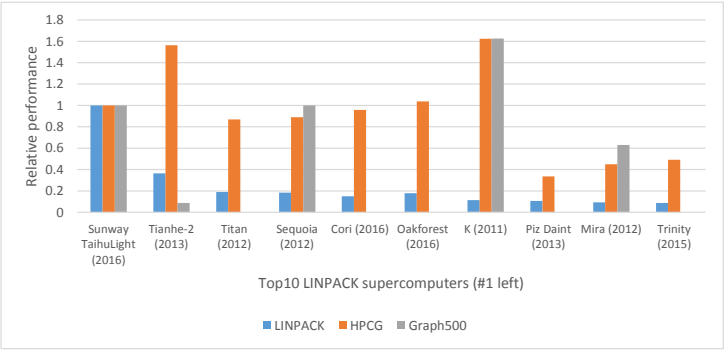


Figure 1.9: The relative performance of the top 10 supercomputers running the LINPACK, HPCG, and the Graph500 benchmark.

not necessarily the metric most relevant today.

1.5.4. HETEROGENEOUS SUPERCOMPUTERS

Figure 1.10 shows the fraction of the TOP500 supercomputers using some form of coprocessors. The first one (the 'Clearspeed' coprocessor) was introduced in 2005, followed by the now very popular GPUs shortly after. Note how these dates correspond with the trends shown in Figure 1.4. Although coprocessors (foremost GPUs) are popularized and used more and more often (e.g. for deep learning), their penetration into the TOP500 is slow. The share of systems using coprocessors never exceeded 20%, and the performance share never exceeded 35%. In fact, in recent years we see a decline in the usage of coprocessors, in both system share as well as performance share.

To understand this better we look at the usage of coprocessors of the top 20 systems of the HPCG benchmark list. In Figure 1.11 we show the performance of these systems, indicating which ones are homogeneous, and which ones are heterogeneous. This is complemented by Figure 1.12, showing the utilization of the same systems. There does not seem to be a clear relation between the heterogeneity of a system and either the performance or the utilization of these systems, for this important benchmark. From

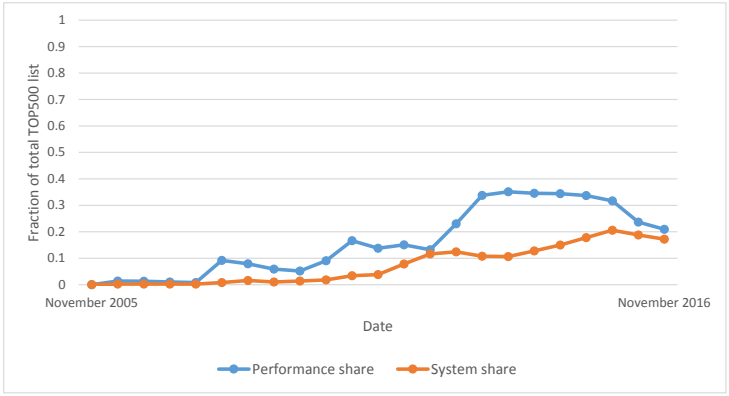


Figure 1.10: Penetration of coprocessors in the TOP500 list of supercomputers.

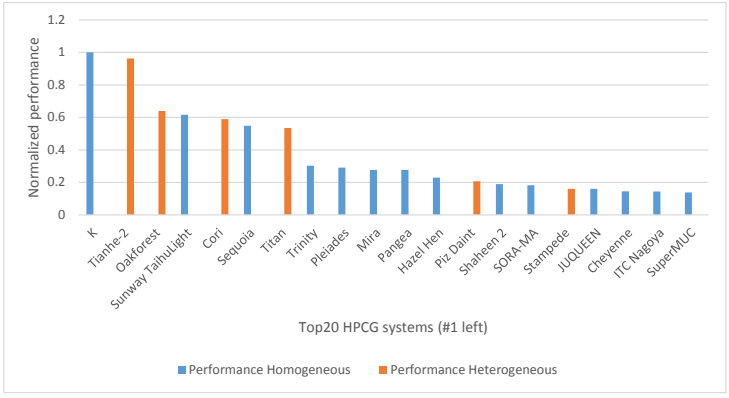


Figure 1.11: The relative performance of the top 20 system running the HPCG benchmark, indicating which ones are homogeneous, and which ones are heterogeneous.

this we can conclude that adding the current type of floating-point performance focused coprocessors so to a system, does not necessarily makes them better at the tasks at hand, despite common beliefs [59].

Heterogeneity is one of the few ways forward left to increase (super)computer performance, but care must be taken on the aspects to focus on. The coprocessors developed over the last decade, and the heterogeneous systems they gave rise to, are focused on peak floating-point performance, while modern applications like the ones represented by HPCG and Graph500, being more data-intensive, do not benefit from this metric. The slow progress on system-level integration and programming support, still limits the penetration of heterogeneous computing.

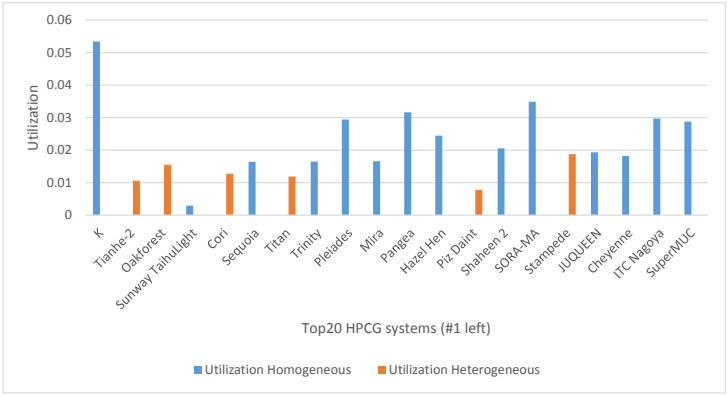


Figure 1.12: The utilization of the top 20 system running the HPCG benchmark, indicating which ones are homogeneous, and which ones are heterogeneous.

1.6. NEAR-DATA PROCESSING

As indicated at the beginning of this chapter, we are experiencing a digital data deluge. Data is becoming an increasingly important asset, and computer systems are needed to create value from this asset. We discussed the problems associated with handling data and transporting data in a computer system, as well as trends regarding customization and heterogeneity. We furthermore showed that supercomputers do not reach their potential when executing data-intensive workloads, and we showed that the classic heterogeneous approaches focusing on compute capabilities do not help. From this we can argue that an effort to create workload-optimized systems, targeting data-intensive workloads, is needed. This overall theme is popularized in recent years as 'near-data processing'. By moving the compute closer to the data, in general, the following benefits are claimed:

- Higher bandwidths, since there is no (or less of a) memory-channel bottleneck to cross;
- Lower latency, since the compute is done physically close to the memory;
- Less energy usage, since expensive data movement is avoided;
- Depending on the architecture and memory technology, a smaller access granularity.

The typical characteristics of applications benefiting from this paradigm are:

- Little temporal data-locality, meaning that data is not reused soon, and local memories (e.g. caches) are thus not helping;
- Low operational intensity, meaning only few operations per amount of data loaded from memory;

- Short dependency chains, meaning the next load depends on the previous load, frustrating efficient prefetching;

- Little spatial data-locality, meaning wide, cache line sized, accesses waste bandwidth.

1.6.1. HISTORY OF NEAR-DATA PROCESSING

The first occurrence of the near-data processing paradigm dates back to 1970 [60]. Already at that moment it was realized that the inherent parallelism inside memory allows for great performance increases when processing elements were added inside DRAM memory. Two decades later, in 1992, this concept was put in a more concrete form and even prototyped, targeting foremost digital signal processing algorithms [61]. In this work it is already recognized that, for some algorithms, execution time is constant for increasing problem sizes, since the number of of processing elements grows with the amount of memory. The EXECUBE work [62] restates the available bandwidth within the memory, but also proposes a system design based on in-memory processors. A more complex variant of the same concept is presented in [63], putting programmable SIMD cores in the memory. A somewhat different approach is presented in [64] [65], putting a SIMD processor on the same die together with large amounts of DRAM, replacing the caches to a large extent.

In 1994, the 'memory wall' was first described [28] (Section 1.3.1), which aligns with the growing amount of publications around the topic of near-data processing [66]. Later in the nineties more in-memory processing proposals were published, and several of them begin to have more concrete system-level proposals, and more detailed implementations [67] [68] [69] [70]. A more detailed programming paradigm appears in [71].

From this moment also the focus shifts from purely in-memory, to also include near-memory. A clear example of this, from 2005, is a heterogeneous approach presented in [72], adding a vector and streaming processor close to the main memory, and including a distinct 'near-memory processor interface' component to create an abstraction layer between memory and the near-memory processor. The 00's has been a quiet decade for near-data processing. Previously discussed research typically did not materialize, or did not make it past a prototype stage. The technical challenges and cost associated with the physical integration of processing elements in memory, as well as the lack of suitable programming models, have been major reasons for this [73].

Since the start of the current decade we can observe a steep increase in near-data processing related research. This is not limited to the 'traditional' in-memory processing, but also includes processing at storage class memory, putting specialized processors close to the memory, and using 3D stacking [73]. As stated in [73], the renewed interested is a combination between the need for data-oriented architectures, technological advances, and a novel class of algorithms. Each is discussed extensively in this chapter. We will discuss recent near-data processing work in detail in the subsequent sections.

1.6.2. NEAR-DATA PROCESSING THROUGHOUT THE MEMORY HIERARCHY

Many interpretations of the 'near-data processing' theme exist, and in Figure 1.13 we show a taxonomy of the near-data processing options throughout the memory hierarchy

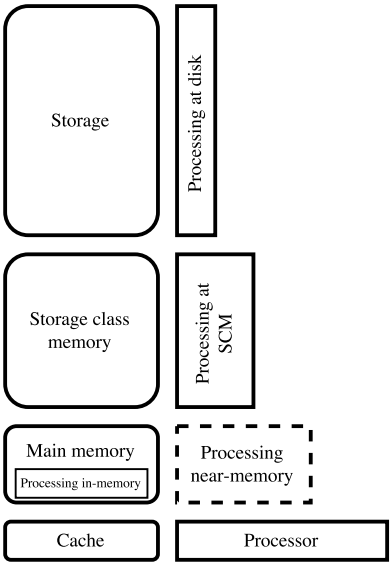


Figure 1.13: Taxonomy of the near-data processing landscape. The dashed lines indicate the part this work is focused on. Image based on [38].

[73]. Data is stored in the entire memory hierarchy of a computer, four levels in the case of Figure 1.13, and near-data processing can be performed at each of those levels.

When looking at the top level, processing at disk, an industry example is IBM Pure-Data [74], built on Netezza technology [75]. These systems offer the execution of massively parallel data analytics based on FPGAs close to the storage. The CPUs are used to orchestrate the system and keep the FPGAs busy, while not having to care about the massively parallel disk IO.

Working directly on storage-class memory is presented in [76]. The work proposes a custom system and board design around Flash and DRAM, targeting big-data workloads. By doing operations close to the storage-class memory, this work improves performance and energy-efficiency significantly. Another example of working directly on storage-class memory is [77], in which the authors implement query capabilities in the firmware of an SSD, improving performance and energy-efficiency. In [78] an embedded GPU is added to an SSD to allow highly parallel processing in a MapReduce [79] context.

The third level, the main memory of the CPU, is divided in two: processing in-memory, and processing near-memory. Processing in-memory integrates the processing elements physically within the main memory of a CPU. Processing near-memory is, to some extent, independent of the memory technology, as the processing elements are physically separated from the memory. Both will be discussed extensively in the next section.

The last level, the processor, is often not considered in near-data processing. However, the accumulated core - L2 cache bandwidth for a modern CPU is 1.5 TB/s [34]. As long as there is enough data locality, CPUs are very capable of running data intensive workloads.

1.6.3. A TAXONOMY OF WORKING IN AND NEAR MAIN-MEMORY

In this work we focus on the third level of the vertical taxonomy shown in Figure 1.13: processing at the main memory of a CPU. This is the most researched interpretation of near-data processing, and therefore an extra breakdown is necessary. In Figure 1.14 a taxonomy of processing near and in the main memory of a CPU is shown. The described workload categories are roughly based on work presented in [80], but in this work a more explicit distinction between workloads and near-data processor (NDP) designs is made. Although believed to be a correct distinction, this way not all combinations with the NDP designs are possible. While a fully programmable NDP can run a simple *copy* workload, a fixed function NDP can obviously not run an *arbitrary application kernel* workload. Categories added for this work are 'memory integration' and 'system integration'.

WORKLOADS

Atomic operations are already supported by memory controllers, realizing a form of near-data processing [40]. The offloading of operations bound to a single cache line to the memory system is proposed in [81].

For compound operations two categories are recognized. First, simple operations like data-reordering [82], and searching for the largest value [83]. A detailed study on the feasibility and performance of copying data and bitwise operations in DRAM is presented in [84] [85] [86] [87]. The second category of compound operations are complex industry-standard operations like encryption and compression. All mentioned concepts can benefit from the advertised higher bandwidth of near-data processing, but the industry-standard operations are challenged by the trend of integrating accelerators with industry-standard APIs in the nest of the CPU, as discussed in Section 1.2.3.

Focusing on entire kernels to offload large parts of applications to near-data processors is promising, as researched in for example [88] and [89], focusing on big-data analytics and scientific kernels, respectively.

NDP IMPLEMENTATIONS

When looking at NDP implementation types, fixed function NDPs can be used to implement bound-operand or compound workloads [80]. Being optimized for a specific task, the energy-efficiency of these processing elements can be very good, of course at the expense of offering limited functionality.

In between fixed-function and programmable we find reconfigurable logic. Reconfigurable NDPs have the great advantage of supporting optimized processing pipelines for every application, and more importantly, support custom ways of handling loads and stores. Applications doing streaming accesses can make use of deep prefetchers and buffer sizes matching the characteristics of the memory system, while applications with scattered and unpredictable memory accesses (e.g. graph processing) can make use of many parallel and independent elements each holding several outstanding loads. An optimized architecture leveraging both reconfigurable logic as well as coarse-grained arrays in the context of near-data processing is presented in [90].

The other end of the spectrum is filled by programmable NDPs, running an instruction set, to execute kernels or entire applications close to the memory [72] [91]. To avoid the energy inefficiency of the superscalar, speculative, out-of-order cores found

in a CPU, but still have enough loads and stores outstanding to saturate all the available bandwidth, several solutions have been proposed. For example the use of many in-order cores, each running multiple threads [88], having a decoupled architecture able to efficiently prefetch data ahead of time [91], or using in-order cores able to process critical loads out of order [92]. An interesting approach in this category is [93], proposing little programmable cores within the memory controllers of the CPU, to accelerate, e.g., double-indexed lookups. By executing two dependent demand cache misses at the memory controller, some latency is cut.

MEMORY INTEGRATION

The category of memory integration is divided in four groups, ranging from physically integrating the processing in the memory, to having the memory as separate chips. Processing in-memory is, as discussed, already a decades old concept, and various ways of integrating processing elements in the (DRAM) memory are proposed [64] [70], but is still an active field of research today [84] [83]. With the rise of novel storage and device technologies, new in-memory processing concepts are explored, of which a recent overview is presented in [94]. The work in [95] proposes the use of memristor device technology to enable true in-memory operations, and evaluates and architecture for simple matrix operations as well as complex bioinformatics workloads. Work in [96] proposes a reconfigurable architecture based on memristors, to improve the useability of this technology. Although promising, the realization of memristor based computers is still in its research phase. Different from memristors, and already available, the Micron Automata Processor [97] can implement non-deterministic finite automata in hardware and can be used to implement, among other things, complex regular expressions.

Advances in stacking technology are sometimes named as a key enabler for near-data processing [98], by adding processing capabilities to either the logic layer of the HMC, or on top of traditional (2D) DRAM [99]. One of the earliest proposals for using the 3D stacking is [100], foremost focusing on technology, but also showing power and performance simulations for isolated kernels. A more general-purpose example of utilizing the logic layer in 3D stacked memory is [101], adding GPU like processing capabilities to realize a throughput oriented near-data processor, targeting HPC workloads. Work in [102] focuses on data-locality and data-placement optimizations within 3D stacked memory. Regarding the HMC product, advantages of this integration technique are, with respect to accessing the memory from outside the device, a smaller access granularity at the full memory bandwidth, a slightly lower latency, and a better power efficiency due to less data movement [35]. Examples of disadvantages are the lack of flexibility, e.g. varying processing capabilities versus memory capacity, and the need for strong coordination across industry, matching product roadmaps, supply chains, and intellectual properties.

2.5D stacked (direct attached) high-bandwidth memory (HBM) has found its way into the high-end GPU market, and offers a dramatic increase in bandwidth with respect to the GDDR alternative [37]. This integration technology has the advantage of being more flexible than 3D stacking, at the expense of the processor sitting further away from the data.

The last category is the most traditional and the most flexible one: having the memory as separate devices. This, among others, includes DRAM memory, the HMC as exter-

nal memory and HBM over a serial link [32]. The best option for memory integration depends on the requirements for bandwidth and capacity, the access characteristics of the key workloads, and the budget, where, e.g., stacked memories can deliver much higher bandwidths, but do not have the cost-effectiveness of DRAM.

NDP INTEGRATION

Regarding NDP integration, the NDPs can be attached to the CPU as a separate device with its own address space. This makes it possible to create fully custom solutions offering great performance for a range of applications [98]. In also implies that data movement, coherence etc. is left to the device driver and user. With the rise of high-speed and coherent links like the already discussed NVLink and CAPI, hard address-space boundaries begin to blur, with some integration aspects being taken over by hardware mechanisms.

The majority of NDP proposals integrate processing capabilities with the CPUs main memory, and are integrated by means of both hardware mechanisms and a device driver / runtime system. This solution offers the highest possible bandwidth to the CPUs main memory, with the lowest possible latency, but implies necessary overhead to manage, for example, coherence and virtual memory. Somewhat of an outlier is [103], proposing the use of near-data processing in the logic layer of stacked memory, attached to a GPU. This is a very novel approach, but the integration aspects are not fundamentally different from integrating with a CPU.

Last, we can integrate the NDP as a full CPU peer in the memory system. In this case the NDP becomes a node in the SMP domain, with the same access rights and capabilities as the CPU, and the NDP can run threads belonging to the OS.

1.7. PROBLEM STATEMENT

The rise of the field of data-analytics, changes in the field of high-performance computing, and the realization of novel large-scale scientific instruments, has resulted in the increased usage of data-intensive applications. Due to the more important role of data, a redesign of computer systems is necessary to increase efficiency by enabling arbitrary processing capabilities close to the main memory of a CPU.

We identify the following problems:

- The next-generation radio telescope will generate a huge amount of data. It is unclear how current technology can handle this, and it is unclear whether the instrument is feasible at all without significant technological advances.
- The addition and integration of novel components in a computer system is a non-trivial task. It is unclear how we can add coherent and virtualized processing capabilities to the main memory of a CPU, or even perform basic operations such as communicating with a component in the memory system.
- By having multiple heterogeneous devices in an asymmetric shared-memory space, the ability to control data locality is essential for performance. Mechanisms to handle data locality for NDPs and CPUs without changing the OS are not yet discussed.

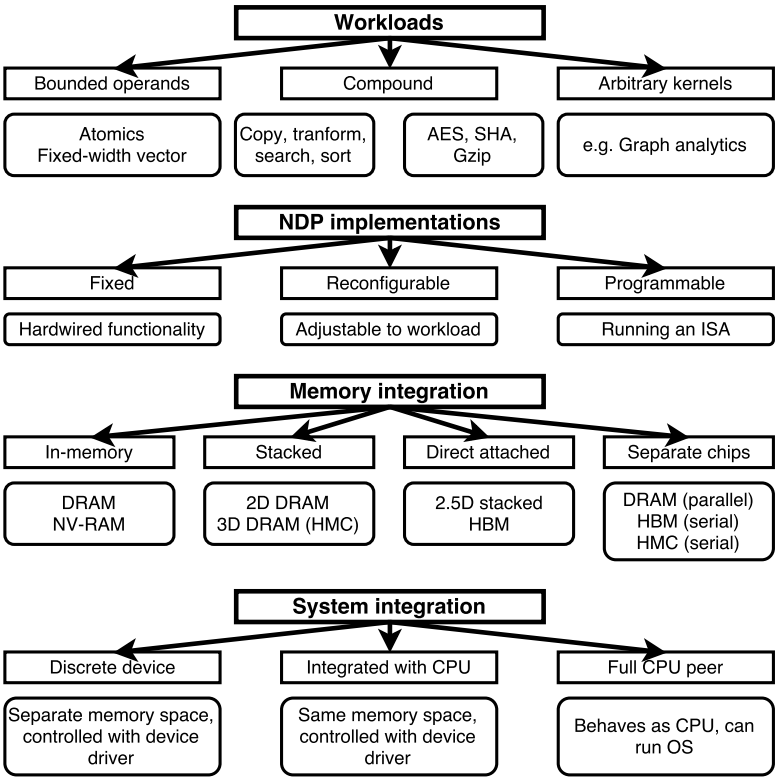


Figure 1.14: Taxonomy of near-data processing in and near main-memory.

- Although the near-data processing concept is evaluated in many forms, clear comparisons with existing hardware when using industry-standard benchmarks is lacking. This makes it hard to evaluate many proposals on an absolute scale, especially when taking in to account common optimizations deployed in the field.

1.8. CONTRIBUTIONS AND THESIS OUTLINE

The contributions of this thesis can be described as follows.

- **An analysis of the (high-performance) computing environment and the Square Kilometre Array radio-telescope.** We reason about the trends and performance of devices and supercomputers for several benchmarks to understand the current bottlenecks. We show how custom and innovative, workload-optimized, data-centric designs are needed to boost both the performance as well as the power efficiency of large supercomputers for relevant workloads. We argue that this is especially true for the SKA radio-telescope, which has unorthodox requirements in terms of bandwidths, data storage, and power efficiency. The SKA can not be realized at its full potential without a significant effort in optimized-system design. This is discussed in Chapter 1 and Chapter 2.

- **An optimized architecture for the Central Signal Processor (CSP) subsystem of the SKA.** We propose an optimized ASIC connected to novel high-bandwidth memory to improve the power-efficiency of the CSP subsystem. By doing so a first step in aiding the realization of the SKA is made. This is discussed in Chapter 3.

- **An architecture to enable arbitrary near-data processing capabilities close to the main memory of a CPU.** The architecture offers all integration features relevant to integrate a near-data processor in a virtualized, coherent and consistent way, without negatively affecting CPU performance. Key aspects are a mixed fine- and coarse-grained coherence scheme, selective extension of CPU snoop and virtual memory operations, and inter-NDP traffic optimizations through a CPU-based cache. Also presented is a method for data-locality management based on existing OS level functionality. This is discussed in Chapter 4.

- **A system simulator simulating a CPU and its memory system, including the new proposed components.** The simulator allows the simulation of workload-optimized near-data processors as well as software running on general-purpose near-data processors. It allows for the evaluation of near-data processing at the application level, as well as heterogeneous CPU and near-data processor usage. This is discussed in Chapter 5.

- **Evaluation and optimization of the HPCG and Graph500 benchmark on the proposed architecture.** Key aspects are a study of access granularities and software prefetching, a study of inter-NDP traffic and its optimization, the cacheability of remote data, executing atomic at the CPU, and various software optimizations. Performance improvements with respect to server-class CPUs are in the order of a factor 3x, for both benchmarks. This is discussed in Chapter 6.

- **The heterogeneous CPU and NDP usage to sort data sets.** Key aspects are a dynamic workload balancing scheme to keep both the CPU and the NDPs busy, and the usage of workload-optimized NDPs. By doing so we showcase the most fundamental property and advantage of near-data processing. When working together, the CPU and the NDP make use of all the bandwidth in the memory hierarchy, and a performance increase with respect to a server-class CPU of a factor 2.5x. This is discussed in Chapter 7.

- **A generalization of the near-data processing concept.** We show, complementing the preceding chapters and looking further in to the future, a different design point for near-data processors. By making them equal to a CPU, we can create a generic, memory-access parallelism focused, data-centric, compute platform. This is discussed in Chapter 8.

2

DATA CHALLENGES IN RADIO ASTRONOMY

The introduction chapter discussed various trends and observations regarding (super)-computers, running industry standard benchmarks. These benchmarks are useful to compare systems between each other, and to show progress over time, but have obviously no real-life meaning. The solving of real-life problems is however the goal, and in this chapter we describe one of these problems.

The never ending pursuit of scientific knowledge by mankind has led to the realization of some very powerful and incredibly complex instruments. A well known example of this in the large hadron collider, build by the European Organization for Nuclear Research (CERN). Another example is the Hubble space telescope, build by NASA. In the field of radio-astronomy, the community is currently designing the next generation radio-telescope, called the Square Kilometre Array, or the SKA. This instrument is supposed to enable significant advances in our understanding of the galaxy.

Already before the design started in any serious way, it became clear that the SKA would generate an astonishing amount of data, and would need an equally large amount of compute power to process the data. With that observation, it perfectly fits the 'digital data deluge' sketched in Section 1.1. The compute requirements, especially when taking the power budgets into account, would upon first sight almost certainly not fit in the already struggling industry trends regarding big and power-efficient supercomputers. In this chapter we analyze the requirements of the SKA, inspect the used algorithms, and argue about the feasibility of its realization.

The content of this chapter is based on the following two papers:

Exascale radio-astronomy: can we ride the technology wave?
E. Vermij, L. Fiorin, C. Hagleitner and K. Bertels
International Supercomputing Conference (ISC), 2014
Best paper award

Challenges in exascale radio-astronomy: can we ride the technology wave?
E. Vermij, L. Fiorin, R. Jongerius, C. Hagleitner and K. Bertels
The International Journal of High Performance Computing Applications
Invited paper

ABSTRACT

The Square Kilometre Array (SKA) will be the most sensitive radio telescope in the world. This unprecedented sensitivity will be achieved by combining and analyzing signals from 262,144 antennas and 350 dishes at a raw datarate of petabits per second. The processing pipeline to create useful astronomical data will require hundreds of peta-operations per second, at a very limited power budget. We analyze the compute, memory and bandwidth requirements for the key algorithms used in the SKA. By studying their implementation on existing platforms, we show that most algorithms have properties that map inefficiently on current hardware, such as a low compute-bandwidth ratio and complex arithmetic. In addition, we estimate the power breakdown on CPUs and GPUs, analyze the cache behavior on CPUs, and discuss possible improvements. This work is complemented with an analysis of supercomputer trends, which demonstrates that current efforts to use commercial off-the-shelf accelerators results in a two to three times smaller improvement in compute capabilities and power efficiency than custom built machines. We conclude that waiting for new technology to arrive will not give us the instruments currently planned in 2018: one or two orders of magnitude better power efficiency and compute capabilities are required. Novel hardware and system architectures, to match the needs and features of this unique project, must be developed.

2.1. INTRODUCTION

The Square Kilometre Array (SKA) [10] will be the largest radio telescope in the world, and it will have an unprecedented sensitivity, angular resolution and survey speed. Most specifications are ten to a 100 times better than any existing telescope. Because of the size of the project, its construction has been divided into two phases: SKA-1 and its extension SKA-2. SKA-1 is currently being designed, and construction will start in 2018. In the same year, the design of SKA-2 will start. This paper will only look at the SKA-1, because the specifications for SKA-2 have yet to be finalized. Therefore, we will from now on refer to SKA-1 as 'SKA'. The SKA will deploy 262,144 antennas and 350 dishes in remote areas in South-Africa and Australia, together producing several petabits of data per second. Realizing the SKA will face many challenges in diverse fields like data transport, algorithms, data storage, and system design. In this paper we will look at the computational challenges of the project: the absolute performance and power efficiency required. Power efficiency has special attention, since several subsystems of the SKA will be located far away from any human infrastructure.

The main contributions of this paper are as follows. We present a detailed computational profile of SKA and its main algorithms, analyze the algorithms on existing hardware, and discuss points for improvement. We show relevant trends in high-performance computing and introduce the innovation metric to compare generations of supercomputers. Finally, we argue about the feasibility of deploying the SKA using commercial off-the-shelf (COTS) hardware.

2.2. SKA PROJECT DESCRIPTION

SKA [104] will consist of three instruments: SKA-low, SKA-mid and SKA-survey.

SKA-low is an aperture array instrument [105] consisting of 1024 stations, each con-

taining 256 dual-polarized antennas, which will receive signals between 50 and 350 MHz. The antenna signals are summed per station into a single beam, which is transported to a central signal-processing facility. The stations will be 35 m in diameter, and placed at most 40 km apart. This instrument will be very much like a big version of LOFAR, the low-frequency aperture array built in the Netherlands [9].

SKA-mid will use 254 single-pixel feed dishes capable of receiving signals between 350 MHz and 13.8 GHz. From this frequency range, a 2.5 GHz band can be selected for measurements. The dishes will be placed at most 90 km apart.

The SKA-survey instrument will use 96 dishes, each containing phased array receivers. Every receiver will have 94 antennas and can point 36 beams onto the sky. In this way, a single dish has a huge field of view, compared with the SKA-mid dishes. The frequency ranges between 250 MHz and 4 GHz, with an instantaneous bandwidth of 500 MHz. The antennas will be spaced at most 25 km apart.

2.2.1. SCIENCE CASES FOR THE SKA

In the early stages of the SKA(1) project, two major science cases where identified, citing the SKA organization [106]:

- Understanding the history and role of neutral Hydrogen in the Universe from the dark ages to the present-day; and
- Detecting and timing binary pulsars and spin-stable millisecond pulsars in order to test theories of gravity (including general relativity and quantum gravity), to discover gravitational waves from cosmological sources, and to determine the equation of state of nuclear matter.

Besides these two main science cases, there are about 10 others, like searching for exo-planets and studying cosmic magnetism. From these science cases, various use-cases have been defined, which give hints towards instrument and processing requirements of what is needed to produce relevant science results. Two main categories can be identified: imaging and non-imaging use-cases. In the imaging mode we create images of the sky, while for the non-imaging mode, we are interested in, for example, time-series. Besides the very first processing steps, these two modes have not much in common.

The first science case stated above (also known as Epoch of Reionization), has an imaging use-case, and some requirements are listed below:

- Use the SKA-low instrument, in a frequency range from 50 to 300 MHz.
- 208,333 image frequency channels, about 1.2 KHz each.
- Image resolution between seven arcseconds and one arcminute, for respectively calibration and actual imaging.
- Image dynamic range larger than 2.5×10^6.

From these requirements we can, to some extent, dimension the telescope and processing pipeline. For example, the image frequency channel requirement translate to the

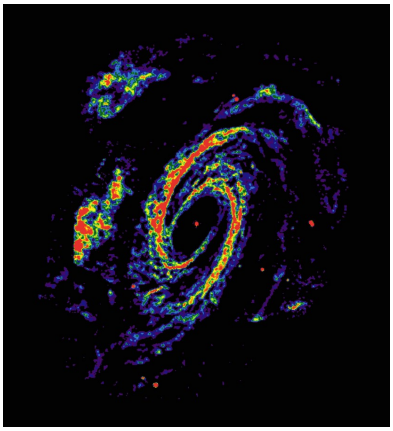


Figure 2.1: Cold atomic hydrogen gas in the M81 galaxy, measured at 1420.4 MHz. As a reference, human-visible red light has a frequency of 430 THz. Image courtesy of NRAO, D. S. Adler, D. J. Westpfahl.

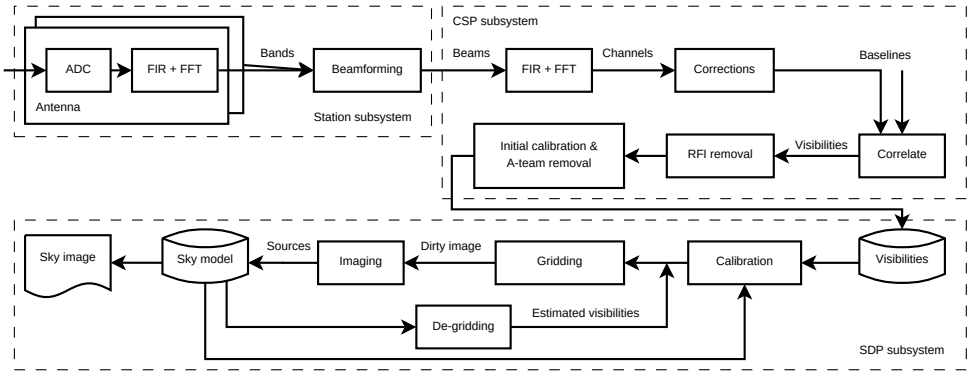


Figure 2.2: Overview of the processing steps for the SKA-low, using imaging pipeline. For SKA-mid and SKA-survey, the antenna/station subsystem will be replaced by a dish.

Fast Fourier Transform (FFT) size needed in the correlator (as explained in the next section). The dynamic range requirement translates to the quality of the entire calibration and imaging pipeline.

As we cannot evaluate all science cases in this work, we will focus on an overall use case: creating sky images for the entire frequency range of the instrument, and no channel integration. Based on experience from existing telescope, this is expected to be the most compute intensive workload. In Figure 2.1 we show an example of a sky image (for a single frequency channel). In this image we see the distribution of hydrogen in the M81 galaxy, which shows a far more extended structure than images made in the human-visible light spectrum.

2.2.2. PROCESSING PIPELINE AND APPLICATIONS

In Figure 2.2, we show the simplified processing flow for the SKA-low, using the continuum imaging science case. For the SKA-mid and SKA-survey, the antenna/station subsystem will be replaced by a single dish.

STATION PROCESSING FOR SKA-LOW

Because the SKA-low uses aperture arrays instead of dishes, extra processing is required to synthesize a dish. The digitized antenna signal from the analog-digital converter is sent to a polyphase channelizer consisting of several finite impulse response (FIR) filter banks and an FFT, creating a number of frequency bands. These are fed into the beamformer, in which every band is multiplied with a complex phase shift to delay the signal, and added to the corresponding band from the other antennas. By delaying signals between antennas and summing them, the instrument focuses its sensitivity into a specific direction, creating a so-called *beam*. The channelization before beamforming is needed because the beamforming concept only works on small frequency bandwidths [107].

CENTRAL SIGNAL PROCESSING (CSP)

The beam from every station or dish undergoes a second channelization, generating finer frequency channels, after which the beams are aligned in time and phase and, in case of the SKA-low, undergo a gain correction to offset filter artifacts from the station processing. The beam is correlated (multiplied) [108] with the data from all other stations or dishes, and integrated over a small period of time (the dump time). By correlating, the signal-to-noise ratio of the data improves. A pair of stations/dishes is called a *baseline*. The result is a visibility, which is a sample of the Fourier-transformed sky. The visibilities are processed by removing RFI signals and by performing a calibration step, to correct for known system inequalities. Furthermore, a set of well-known very bright sources (the A-team) is demixed from the dataset. After these steps, the data is often integrated again in time and frequency, depending on the frequency smearing [109] and/or other science requirements.

SCIENCE DATA PROCESSOR (SDP)

From the visibilities, a sky image can be constructed. The calibration works on a station/dish basis, and accounts for both direction-independent effect (gains, crosstalks) and direction-dependent effects, such as ionospheric distortion. The corrected visibilities together with the calibration solution are passed to the imaging pipeline. From a telescope model and the calibration parameters, we can create a small map representing the complex gain function for a beam, called A-projection (analog to the lens behavior in an optical camera) [110]. Two maps of a single baseline are multiplied together, and then multiplied with a W-term to account for the non-coplanar baselines effect [111] (the earth is not flat), and scaled up. The resulting map, or convolution matrix, is multiplied with a visibility and added (gridded) onto a Fourier grid. This gridding process happens in various time-steps, called snapshots [112]. All snapshots are later refitted into a single grid. A Fourier transform of the grid results in a dirty image, and the CLEAN deconvolution algorithm [113] is used to extract bright sky sources. After a certain threshold has been reached, the extracted sources are converted back into visibilities (de-gridding), which are subtracted from the original dataset. This gives us a visibilities dataset with

only weak sources, and the gridding process starts over until only noise is left. All extracted sources are kept in a sky model. This sky model is used to make better estimations about certain calibration parameters, resulting in another feedback loop back into the calibration step. After a sufficient amount of iterations, the sky model can be converted into a sky image, ready for research.

2.3. SKA COMPUTATIONAL PROFILE

In this section we analyze the compute requirements of various applications in the SKA processing chain. The key element of this research are the scaling rules, or how the compute for an application relates to telescope-parameters. The analysis shown is based on work performed by Jongerius [114], internal project documents, and own research. We would like to point out that the numbers in this section are estimations. Furthermore, some algorithms, like calibration, are missing, since they are at this moment not well defined enough to include.

2.3.1. STATION PROCESSING APPLICATION SET

CHANNELIZATION

The channelization consists of a fixed-size bank of FIR filters and a fixed-size (real to complex) FFT. The compute requirements per second per single-polarized antenna are given in Equation 2.1. N_{taps} is the number of filter taps and N_{bands} is the number of frequency bands. $Samples_{sec}$ is the incoming sample rate.

$$Ops_{channelization} = Samples_{sec} \times (2N_{taps} + 5 \times 0.5 log_2(2N_{bands})) \tag{2.1}$$

BEAMFORMING

The beamforming step multiplies every dual-polarized antenna sample with 2×2 matrix holding complex weights. Thus every sample undergoes 14 operations, eight multiplications and six additions.

2.3.2. CSP APPLICATION SET

CHANNELIZATION

The compute requirements per second per beam for this step is a variation on Equation 2.1. In this case the inputs are complex numbers, and we generate frequency channels instead of frequency bands.

CORRELATION

The correlator multiplies two signals together and adds the result to a sum. The compute for the correlator per second per channel is given in Equation 2.2. Here, N_{stat} is the number of stations or dishes.

$$Ops_{correlator} = 8 \times Samples_{sec} \times 0.5 \times N_{stat}^2 \tag{2.2}$$

An implementation challenge lies in the fact that data arrives per station, containing all the frequency channels of that station, whereas the algorithm wants the data per fre-

quency channel, containing all the stations. This data rearrangement is often called the 'corner turn', and frustrates practical implementations of the correlator.

RFI REMOVAL

Based on experiences from LOFAR, we see that good RFI removal costs 268 operations per input sample [115]. This is not shown in an equation.

2.3.3. IMAGING APPLICATION SET

CONVOLUTION MATRIX GENERATION

Creating the convolution matrices as described in Section 2.2.2 involves several 2D FFTs and point-wise matrix multiplications. The compute is however dominated by a single 2D FFT, often not a power-of-two in size. The compute requirements per second per channel and baseline are given by Equation 2.3. W is the average W-matrix size, and O is a scaling factor. S_{sec} indicates how many seconds the projection matrices are valid because of time dependent effects. C_{chan} is the channel compression, indicating how many channels can be served by the same W-A combination. N_{iter} is the number of iterations following the feedback loops described in Section 2.2.2.

$$Ops_{Conv-matr-gen} = 5 \times \frac{N_{iter}}{S_{sec} C_{chan}} \times W^2 O^2 log_2(W^2 O^2) \tag{2.3}$$

The up-scaling introduces a form of interpolation in the gridder. This is necessary because the location of the visibilities is of much higher precision than the Fourier plane gridpoints.

W-SNAPSHOTS GRIDDING

As de-gridding and gridding are very much the inverse of each other with the same kernels and properties, we will only focus on gridding in this section. Based on the location of the visibility with respect to the grid, a 1/64th subset of the convolution matrix is selected. The visibility is multiplied with this matrix and added to the Fourier plane. The compute per second per channel is given in Equation 2.4. T_{dump} is the correlator dump time, and N_{bl} the number of baselines.

$$Ops_{gridding} = (6+2) \times \frac{N_{iter} N_{bl}}{T_{dump}} \times W^2 \tag{2.4}$$

A practical aspect of this algorithm is that additional parallelism in the baselines exists. However, exploiting that can be difficult because of the addition to a final grid, which has to happen in an atomic way.

W-SNAPSHOTS RE-PROJECTION AND 2D FFT

Refitting a gridding snapshot consists of a coordinate transformation, and a 2D FFT. The re-projection is estimated to be 50 operations per pixel per channel and is not shown in an equation. The compute requirements per channel for the FFT follow Equation 2.5, where R is the amount of pixels in one dimension of the image. $Snapshots_{sec}$ indicates the amount of seconds a snapshot is valid.

$$Ops_{snapshot-refitting} = 5 \times \frac{N_{iter}}{Snapshots_{sec}} \times R^2 log_2(R^2) \tag{2.5}$$

DECONVOLUTION

For this application we assume Cotton-Schwab CLEAN [116], together with the W-snapshots described earlier. An estimate for the compute requirements per channel for this algorithm are shown in Equation 2.6. N_{mc} is the number of minor cycles, N_{pp} is the amount of pixels along one axis of the dirty-beam patch, N_{pi} is the amount of pixels along one axis of the image, N_{al} is the number of pixels in the active-list, and $Measurement_{sec}$ is the measurement time. A detailed explanation of these parameters is beyond the scope of this work, but can be found in [117].

$$Ops_{deconvolution} = 3 \times N_{mc} \times \frac{N_{iter}}{Measurement_{sec}} \times (\frac{N_{pp}}{N_{pi}})^2 \times N_{al} \tag{2.6}$$

2.3.4. COMPUTE REQUIREMENTS

With the analysis performed in the previous Sections, and the specific parameters of all the instruments, we can calculate the compute requirements. The result is shown in Figure 2.3. From this Figure it is clear that several applications have very high compute requirements, up to several hundreds of peta-ops per second. To make this work as relevant as possible, we focus to four very compute intensive kernels, namely:

- FIR + 1D FFT (channelization), used in the SKA-low stations and the CSP.
- Correlation.
- 2D FFT, used in creating the convolution matrices, the W-snapshots re-projection, and several places not explicitly mentioned in this work.
- Gridding.

In the remainder of this work we will (foremost) talk about these four kernels.

PARALLELISM AND DATATYPES IN THE APPLICATIONS

The compute requirements for SKA are high, but there are also trivial parallelizations possible. Already recognizable in the previous section, most SDP applications can be parallelized over frequency channels. These channels do not interact with each other, and they can therefore be processed completely independent of each other. Another parallelization possibility are the beams, which is especially relevant for SKA-Survey.

Another interesting point of our application set is that the datatypes are not necessarily the default 32 or 64 bit floating-point. For example, the input for the correlator is defined as 8 bit integers. The applications in the SDP do not have requirements like this, but it is clear that some steps need more precision and/or dynamic range than others. For example, the multiplication of a visibility with a convolution matrix can be done with a 'small' datatype, compared to the full Fourier plane.

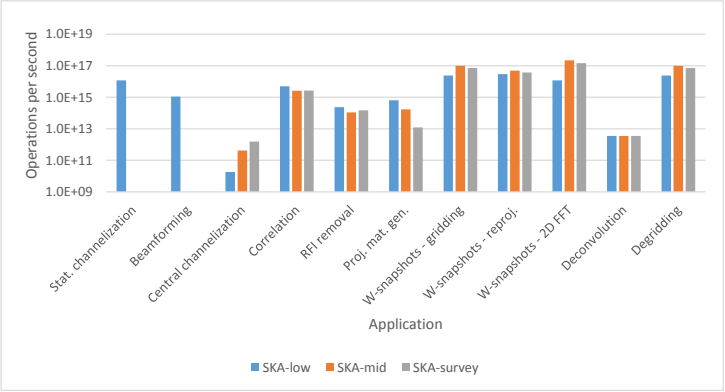


Figure 2.3: Compute requirement estimates for the various applications for the three SKA instruments.

2.3.5. VISIBILITY BUFFER AND DATAFLOWS

The visiblity stream out of the correlator must be stored in a visibility buffer (or UV buffer). The various major and calibration cycles can then be executed on this dataset. The visibilities need to be stored in a ping-pong fashion, one buffer for the incoming data, and one buffer for the dataset currently being processed. Here we assume that the telescope will always be 'on'. In Figure 2.4a we show the amount of storage needed for the visibilities, for the three instruments of the SKA. It can be observed that the buffer requirements are very high, over a 100 petabytes.

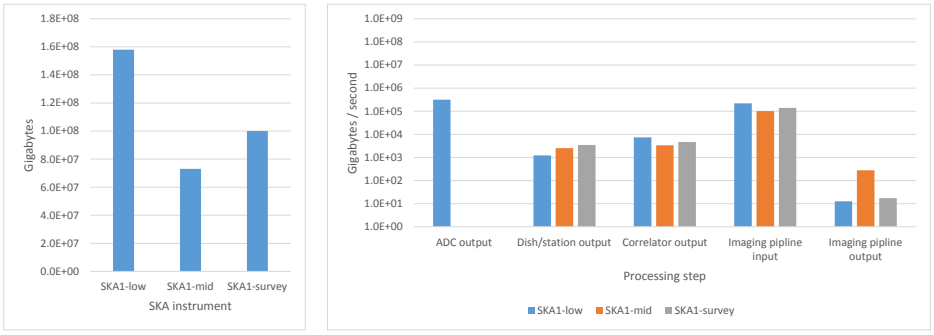
The input and output bandwidths for several steps for the three SKA instruments are shown in Figure 2.4b. For the SKA-low we see a huge bandwidth reduction from the AD-converters to the station output, due to the beamforming, where we sum all the antennas together. For all the instruments, we see an increase in data rates in the correlator. This is due to the large number of stations/dishes we have in SKA. Most existing, smaller, telescopes reduce the data rates in the correlator, due to the summation happening there. The input data rate for the imaging pipeline is again much higher than the correlator output because of the various calibration and major loops that need to happen here. Finally, the output data rates of the imaging process (a set of images), is much smaller, as a large stream of visiblities is converted into an image.

2.4. THE SKA ON THE TECHNOLOGY OF TODAY

As shown in the preceding sections, SKA will require significant computational power. In this section, we will be analyzing state-of-the-art implementations of the key algorithms, and take a look at how we could optimize current technology for SKA.

2.4.1. CORE TECHNOLOGIES

A decade ago, the first *multi-core* CPU was introduced, which sacrificed single-core peak performance for parallelism. With *manycore* architectures, this concept is taken to the extreme. Besides using more cores, *specialized* logic designed for a specific task can be added: using additional area is traded off for better power efficiency or through-



(a) (b)

Figure 2.4: (a) UV buffer requirements for the SKA and (b) bandwidths throughout the SKA instruments.

put/latency [20]. Performance and/or power efficiency can be further improved by using *heterogeneity*. This can be done on a node level by using multiple types of devices (CPUs and GPUs for example), or within a single device (ARM® big.LITTLE [26] for example). Another kind of heterogeneity comes in the form of attached FPGAs, which are *reconfigurable*. Examples of this are the Molen polymorphic processor [118], and the systems of Convey Computer [119].

Most advances in power efficiency and throughput we see today are based on these technologies. For example, GPUs employ the manycore paradigm, have thousands of small cores, and are often used in a heterogeneous setup. CPUs become faster by adding wider and more specialized instructions [120] or small accelerators [24].

2.4.2. THE SKA KERNELS ON EXISTING PRODUCTS

CHANNELIZATION

The work performed by Shahbahrami et al. [121] shows that FIR filters using real numbers are well suited for SIMD parallelization. For complex numbers additional data-shuffling hardware is required. Jongerius et al. [122] show that a modern CPU can only run a real FIR filter at 10-15% of its peak performance, because of the low number of operations per byte of I/O. Romein [123] shows that this also holds true for complex FIR filters on GPUs.

The FFT algorithm features an irregular data access pattern with low computational intensity, which make it hard to run this algorithm efficiently on almost any architecture. Jongerius et al. [122] show that the FFTW library [124] reaches 17% of the peak performance of a modern CPU. The research by Xu et al. [125] shows in detail how the FFT can be optimized for SIMD processing on a modern CPU. Romein [123] shows that FFTs on GPUs are heavily IO bound, and achieve around 20% of the peak performance, comparable with CPUs. Research by Lobeiras et al. [126] and nVidia's own CUFFT library confirm this.

LOFAR shows us that using FPGAs on the Uniboard [127] is a good match for the fixed-sized FIR and FFT. The multiple bitwidth modes LOFAR can use are all supported

on a single FPGA image.

For the channelization we can conclude that SIMD and SIMT models both work fine, and that the bottlenecks are in the memory bandwidth and memory access pattern.

CORRELATOR

Nieuwpoort et al. [128] implemented the correlation algorithm on several architectures, namely CPUs, GPUs, the Blue Gene®/P [129], and the Cell processor. They conclude that having sufficient local storage is key to good performance. Both Cell and the Blue Gene® achieve close to peak performance, whereas CPUs and GPUs reach significantly lower numbers (70 and 40% respectively). Work by Clark et al. [130] and Romein et al. [123] shows that for modern GPUs, a peak performance of up to 80% can be achieved for a large number of stations, provided that significant optimization effort is put into register and IO usage.
[1]

Research by Woods [131] and Souza et al. [132] shows that FPGAs are a suitable candidate for running the correlation algorithm, because of its regular and simple structure. Utilizing custom datawidths is a major, and realistic, advantage over other architectures.

In conclusion, the correlator runs well on most architectures, but a big local storage is important. Using custom datatypes gives an advantage.

2D FFT

A white paper published by Intel® [133] reports CPU utilizations ranging from 70% for a 64×64 dataset down to 50% for a 256×256 dataset, using their commercial math packages. For larger datasets the utilization drops, but one dimension is always kept at or below 256, which makes the results less valuable for this analysis. For the utilization on a GPU we run some small benchmarks. Using an nVidia® K20 GPU and CUDA® 6.0, we see rather flat utilizations in the range of 7.5 and 10%, for input sizes from 64×64 to 8192×8192. The datasets for 2D FFTs on full sky images (10,000 - 50,000 pixels across) will not fit on current day discrete accelerator devices, thereby frustrating an efficient implementation.

Despite vendor-optimized codes, we see that (large) 2D FFTs do not run very well on CPUs and GPUs. Their computational intensity is somewhat better than 1D FFTs, but it is still low.

GRIDDING

The naive way to implement this algorithm gives a very low computational intensity [134], and a lot of atomic add operations. This is shown in [135] for CPUs and GPUs. Both platforms achieve around 5% of their peak performance because of bandwidth limitations. An implementation on the Cell [136] achieves around 25% of the peak performance, but only after considerable optimizations in the memory bandwidth usage. GPU research performed in [137] tries to eliminate atomic add instructions as much as possible, and the GPU reaches 25% of its peak performance. Even when there are hardly any atomic add operations left (0.23% of all grid updates), they still take up 26% of the time.

[1] Blue Gene is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product or service names may be trademarks or service marks of IBM or other companies.

For the gridding, we can conclude that the simple kernel will run well on most architectures; bottlenecks are in the memory bandwidth and atomic additions.

2.4.3. HINTS TOWARDS OPTIMIZED ARCHITECTURES

This section presents and analyzes two aspects of the key SKA algorithms: first, the power breakdown on CPUs and GPUs, and second, the cache performance on CPUs. Together with the work presented in Section 2.4.2, these analyses can be seen as a starting point for research into architectures that can run the algorithms at higher throughputs and with improved power efficiency. The CPU tests are run on a model of an Intel® Xeon® E5-2630 implemented by using the Gem5 simulator [138] and McPAT [139], while the GPU tests are run on a generic model of a NVIDIA® GTX480 GPU implemented by using an enhanced version of GPUWattch [140], a microarchitecture-level GPU power simulator based on GPGPU-Sim [141] and McPAT. This GPU architecture is not the most modern one, and already superseded by two architectures. The general organization of GPUs has however not changed, therefore we believe the results shown are still relevant. The CPU performance numbers are obtained by using the Intel® Vtune™ Amplifier and again an Xeon® E5-2630 processor.

For the channelization, we used the algorithm described by Romein [123]. The correlator implementations are based on code presented by Nieuwpoort et. al. [128], and the gridding implementation described by Romein [137]. For the 2D FFT kernel, we used a 128×128 complex matrix as input. This is a realistic size for the convolution matrix generation, but somewhat small for the W-snapshot refitting.

POWER BREAKDOWN

Figure 2.5 shows the results of our power breakdown experiments. As can be seen, the CPU shows very similar power distributions for all the algorithms. Only a very small portion of the energy goes to the actual computations. Around 50% of the energy goes to the three levels of caches, and another 20% to 30% is spent in the memory controllers. Although not shown in the graph, almost all the power for the level two and level three caches is due to the static power component. For the level one cache, the static and dynamic power distribution is about equal. This means that a significant part of a CPUs power usage is due to the presence of these big caches, which consume energy even when they are idle. From the graph it is possible to notice the slightly higher usage of the memory controllers in the channelization, due to the streaming nature of the algorithm. Moreover, the correlator uses more power in the level one cache, which corresponds with its intensive use of a local storage, as indicated Section 2.4.2.

The breakdown for the GPU shows a bigger power percentage for the functional units than the CPU, as one would expect with GPUs being compute oriented platforms. This difference is especially clear when running a compute-bound algorithm like the correlator. For the 2D FFT the difference is much smaller. This shows that compute bound problems run more efficiently on a GPU, and this is of course the reason why they are used in HPC. Another interesting observation is the large amount of energy GPUs spend in the register file (especially compared to CPUs), around 15%. This corresponds with the fact that the register file in a GPU is very big, and can be compared with the level one cache in CPUs. The constant power (around 15%) is mainly caused by processor

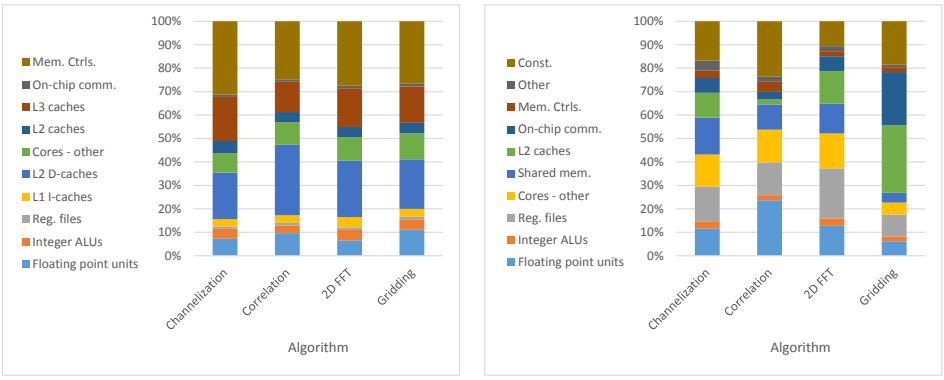


Figure 2.5: Power breakdown on CPUs (left) and GPUs (right) for the four SKA algorithms considered in this work: channelization (Chann.), correlator (Corr.), 2D FFT and Gridding (Grid.).

Table 2.1: CPU cache performance and ALU utilization for the four SKA algorithms considered in this work. Here Chann. is the station channelization.

	Chann.	**Correlator**	**2D FFT**	**Gridding**
Cycles under L1$ miss	29%	32%	46%	1%
Cycles under L1$ miss, L2$ hit	3%	29%	26%	1%
Cycles under L2$ miss, L3$ hit	4%	2%	7%	0%
Cycles under L3$ miss	23%	1%	13%	0%

and memory leakage power and peripheral circuits' power [140]. In the case of GPUs, it can be noticed that we have a high power consumption of the floating point units while running the correlator, as we would expect for a compute bound problem. Furthermore, the 2D FFT consumes a lot of energy in the local storage (register files, shared memory and level two cache), corresponding with the large datasets it has to process in a relative inefficient way.

CPU CACHE PERFORMANCE

The results for the CPU cache performance are shown in Table 2.1, and correspond with what we know about the algorithms. The channelization does not use the level two and level three cache: data is streamed from external RAM into level one, processed and evicted again. For the correlator we know that it appreciates large amounts of local storage, this is reflected in the numbers: level one misses almost always hit in level two. The 2D FFT behaves in the same way as the channelization, but also uses the level two cache, since it uses larger datasets. The gridding exhibits very good cache behavior.

CONSIDERATIONS

From these numbers, we can extract some general guidelines on how to improve the power efficiency and throughput of these four algorithms.

GPUs spend a significant amount of energy in the floating point units, the register file and the rest of the core. We see a similar picture for the corresponding CPU cate-

gories. Adding macro instructions to the core-architecture for executing FFT butterflies (beyond existing shuffle instructions), complex multiplications or even bigger instruction blocks, would result in less register file accesses and more efficient execution units. This could reduce the power consumption and improve the throughput. Furthermore, core-architectures could be optimized to capture the simplicity and regularity of the algorithms. The correlator for example, has basically a single type of instruction (complex multiply-accumulate) inside several loops, but still, when running this algorithm on a CPU, five percent of the power goes to the instruction cache. Exploiting this regularity can also improve the power efficiency and throughout.

Besides the computational part of the chips, the local memories use large amounts of energy, and are often also a performance bottleneck. Making the memories deeper and wider will make performance better, but the energy consumption worse. A possible improvement would be to be able to exploit the regularity and predictability of the data needs. With carefully arranged accesses, specific to our SKA algorithms, we could utilize the available bandwidth to the largest extent. Similarly an improved, more SKA specific, organization of the local memories would increase data locality, reducing the number of accesses to a lower level or the external memory. The numbers presented in Table 2.1 and the power breakdown observations in Section 2.4.3 can serve as a guideline here.

2.5. TECHNOLOGY CHALLENGES FOR THE SKA

In Chapter 1 an overview is given of several industry trends, from the device level up to the supercomputer level. We can revisit the challenges for the future of radio astronomy. For SKA, the power and memory walls can be summarized into a data-locality problem. The instrument generates so much data (Section 2.3.5), that it is impossible to move the data to an appropriate processing device. As a node level example: a modern PCIe-attached device needs hundreds of operations per incoming byte to run at a high utilization. We do not have that amount of operations, as indicated in Section 2.4. Between nodes we see a similar problem: a significant portion of supercomputing power goes to the interconnect, and with the rapidly increasing node peak performance, data communication cannot keep up. Managing where the data is with respect to the compute resources will be the key challenge.

As discussed in the introduction, power efficiency is of special importance to the SKA, since several subsystems will be located in remote areas. Power will have to be transported there or generated on the spot. This gives another dimension to the power efficiency challenge.

2.6. CAN SKA RIDE THE TECHNOLOGY WAVE?

2.6.1. AT NODE LEVEL

The algorithms discussed and analyzed in Sections 2.3 and 2.4 exhibit features that do not map well on existing technologies. In Section 2.4.3, we already briefly discussed some of them: predictable data-access patterns, high bandwidth-to-compute ratio, and complex-number arithmetic. Other exploitable features are narrow and/or flexible datawidths and simple highly repetitive kernels.

Given the challenging requirements of the SKA, we cannot afford to either waste

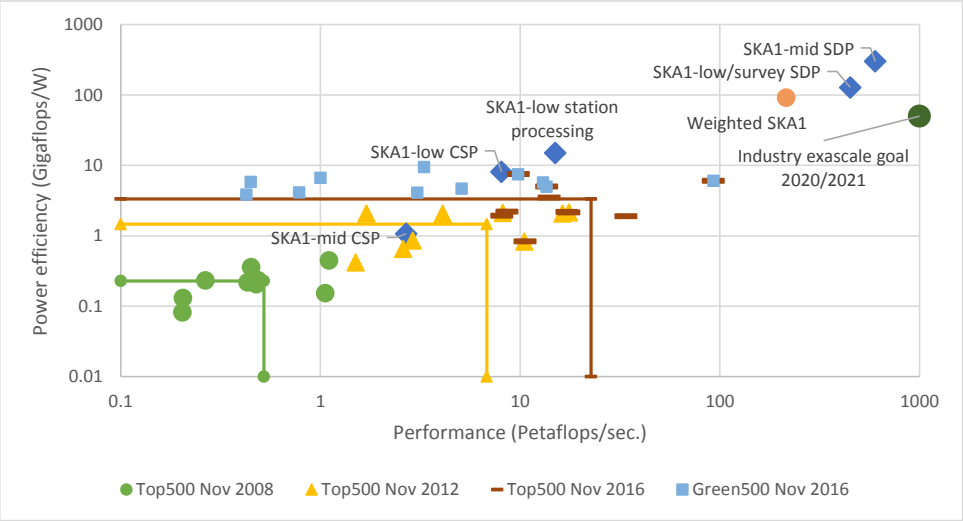


Figure 2.6: The compute and power efficiency points for the SKA subsystems, put next to TOP500 trends.

computational resources or energy or not benefit from some specific features the algorithms offer us. An extra level of specialization, like the industry examples indicated in Section 1.4, will be needed.

2.6.2. AT THE SYSTEM LEVEL

In Figure 2.6, we show the compute versus power efficiency operation points for the several subsystems of SKA. The SKA systems should be built towards the end of this decade. Power requirements are found in the SKA baseline design, and the compute is based on Section 2.3. The graph furthermore shows several point sets and a trend based on several TOP500 top ten systems [3]. We would like to emphasize that our kernels will run on a lower utilization than the Linpack kernel for the TOP500 / Green500 data points. For this figure we do not use the latest TOP500 list, since the number 10 in that list (the only new one), does not have a reported power number.

Two observations can be made. First, the power efficiency of the subsystems are orders of magnitude apart. This means that the power budgets are not balanced very well. As illustration we also included a weighted power efficiency in Figure 2.6. Secondly, it is clear that riding the technology wave and waiting for systems available at the end of the decade will not give us the hardware needed to realize some subsystems (foremost the SDPs) of SKA.

Given the data challenges indicated in Section 2.5, novel data-centric approaches should be considered. A high-level example of this could be to split the system physically into frequency channels (as described in Section 2.3.4). Data can stay at rest within a 'channel-node', while the node does all consecutive processing steps. This would significantly reduce the load on the system interconnect, and thereby reduce the power consumption. With such an approach, we would end up with a true data-centric super-

computer.

2.7. CONCLUSION - REALIZING THE SKA

In this paper, we presented an overview of SKA from a compute perspective, and analyzed whether the instrument can be built using COTS hardware in 2018. Three observations can be made: 1) SKA will require large amounts of computational power at a very high power efficiency. 2) Most SKA algorithms will only run at efficiencies around 10% to 50% on COTS hardware, with clear points for improvement being smarter memory accesses and specialized execution units. 3) The HPC innovation and trend analysis shows that custom-built machines achieve two to three times larger improvement in compute capabilities *and* power efficiency than systems employing COTS hardware.

We conclude that waiting for new technology to arrive will not give us the instruments currently planned at the end of the decade: an order of magnitude better power efficiency and compute capabilities are required. Developing new, workload specific technology, specialized for the tasks at hand must be considered. Solving the data-locality problem will be key: there should always be a suitable compute element near the data, on every level.

3

A CUSTOM COPROCESSOR FOR RADIO ASTRONOMY CORRELATION

The use of customization within a device, as well as the usage of custom systems, is an efficient way to increase the performance and power-efficiency of workloads. The level of customizations, and thus optimizations, that can be applied, depends on the application demands, but also on the longevity of the applications' cause, and the achievable impact. For the SKA radio telescope, unorthodox compute and power efficiency requirements are presented, and workload optimized systems can be a successful way to realize (parts of) the SKA. The first processing stages for the SKA require a significant amount of compute, but their complexity and variability are limited. These stages are therefore an excellent candidate for custom and well optimized computer systems, as described in this chapter.

The content of this chapter is based on the following paper:

An energy-efficient custom architecture for the SKA1-Low central signal processor
L. Fiorin, **E. Vermij**, J. van Lunteren, R. Jongerius and C. Hagleitner
Computing Frontiers, 2015
Best paper award

ABSTRACT

The Square Kilometre Array (SKA) will be the biggest radio telescope ever built, with unprecedented sensitivity, angular resolution, and survey speed. This paper explores the design of a custom architecture for the central signal processor (CSP) of the SKA1-Low, the SKA's aperture-array instrument consisting of 131,072 antennas. The SKA1-Low's antennas receive signals between 50 and 350 MHz. After digitization and preliminary processing, samples are moved to the CSP for further processing. In this work, we describe the challenges in building the CSP, and present a first quantitative study for the implementation of a custom hardware architecture for processing the main CSP algorithms. By taking advantage of emerging 3D-stacked-memory devices and by exploring the design space for a 14-nm implementation, we estimate a power consumption of 14.4 W for processing all channels of a sub-band and an energy efficiency at application level of up to 208 GFLOPS/W for our architecture.

3.1. INTRODUCTION

The Square Kilometre Array (SKA) will be the biggest radio telescope in the world, with unprecedented sensitivity, angular resolution, and survey speed. Collectively, the antennas composing the SKA are expected to gather 14 exabytes and store one petabyte of data every day, requiring exa operations per second for the processing [10]. In this work, we focus on the implementation of the *central signal processor* (CSP) of the SKA1-Low, the SKA's aperture array instrument that will be built in the first phase of the deployment of the SKA and consisting of 512 stations, each containing 256 dual-polarized antennas [10]. We refer in particular to a two-stage signal channelization implementation in which the signal-processing load is distributed between the stations and the CSP (see Figure 3.1 for details) [10]. This configuration is similar to the one implemented in LOFAR, the low-frequency aperture array built in the Netherlands [114].

Figure 3.1 shows an overview of the processing steps for the SKA1-Low, in the case of the *continuum imaging science case* [10]. The processing flow can be divided in three parts: the *station processing*, the *central signal processor*, and the *science data processor* (SDP) [1].

This paper focuses on the study and evaluation of an energy-efficient custom architecture for the CSP. Current proposals for its implementation rely on the use of high-performance homogeneous supercomputers [129] or on GPU-based accelerated architectures [130], and hardware solutions based on FPGAs [132]. We present a first quantitative study for the implementation of a custom hardware architecture for processing the CSP algorithms. By taking advantage of emerging 3D-stacked memory devices and by exploring the design space for a 14-nm implementation, we evaluate that our architecture consumes 14.4 W for processing all channels of a sub-band, achieving an energy efficiency at application level of up to 208 GFLOPS/W.

The remainder of this paper is organized as follows: Section 3.2 describes the CSP and presents an analysis of its main algorithms and of their requirements. Section 3.3 introduces the proposed architecture and discusses the advantages of the new data organization. In Section 3.4, the proposed architecture is evaluated. Related work is pre-

[1] We refer the interested reader to [129, 142] for further details.

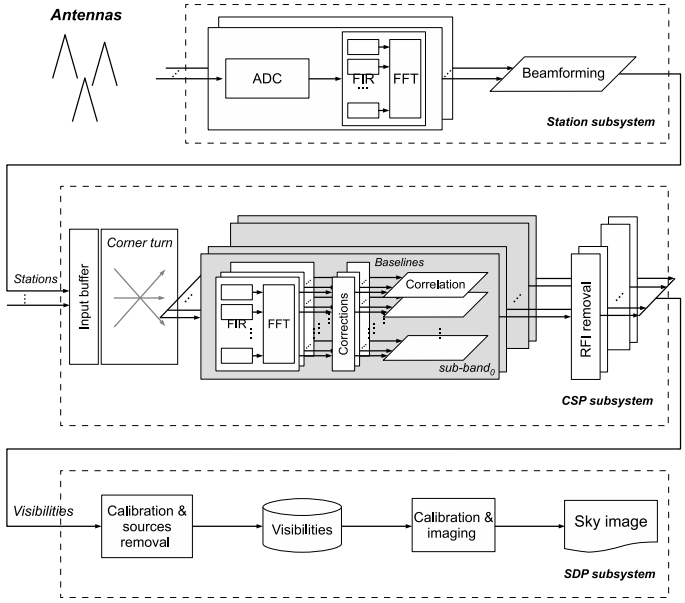


Figure 3.1: Overview of the processing steps for the SKA1-Low, using the continuum imaging science case [142].

sented in Section 3.5, and Section 3.6 concludes the paper.

3.2. CENTRAL SIGNAL PROCESSOR

Figure 3.1 shows an overview of the main processing steps of the SKA1-Low CSP, taking as example the configuration implemented in LOFAR, the low-frequency aperture array built in the Netherlands [114]. At the stations, dual-polarized signals from each antenna are digitized and sent to poly-phase channelizers, which creates a number of frequency sub-bands. Signals of every band are delayed appropriately by multiplying them with a complex phase shift, and added to the corresponding band from the other antennas to create a so-called beam.

At their arrival at the CSP, station data are copied in a memory buffer and a coarse-grained inter-sample time delay is introduced in the beam data to correct for the different relative stations distance and align the beams in time and phase. Station data arrive separated over N_{band} coarse-grained channels or sub-bands, thanks to the first channelization phase performed during station processing.

In the second step, in the stage called *corner turn*, data are redistributed. Data arrive at the CSP *per station*, and the corner turn redistributes them to have data from all stations for a single frequency sub-band in the processing node. This is done because data need to be processed *per sub-band*. After the corner turn, a new channelization step is performed in which signals are split into even narrower frequency channels. After the channelization step, data are multiplied with complex coefficients to correct for artifacts introduced by the filter banks in the station processing.

At this point, samples from every station in the same frequency channel are corre-

Table 3.1: CSP application parameters.

CSP requirements		
Parameter	Description	Value
N_{stat}	# transmitting stations	512
N_{band}	# sub-bands (coarse-grained channels)	512
N_{beam}	# beams per station	1
N_{pol}	# polarizations	2
f_{stat}	Station samples frequency (Mega samples/s)	360
N_{ch}	# channels for frequency sub-band	512
N_{taps}	# FIR filter taps	8
N_{dump}	# samples integrated at the correlation step	1200
Ops_{fir}	# operations per sec. - FIR (Tera ops/s)	11.06
Ops_{FFT}	# operations per sec. - FFT (Tera ops/s)	16.59
Ops_{correc}	# operations per sec. - corrections (Tera ops/s)	5.16
Ops_{corr}	# operations per sec. - correlation (Peta ops/s)	1.51

lated to detect statistic coherence in the received signals - otherwise too weak to be distinguished from the noise - and integrated over a small period of time to reduce the output size. The result data of this step, called *visibilities*, are processed by removing any sample disrupted by radio-frequency interference.

In this work, we focus on the most computing-intensive algorithms of the CSP, i.e., the *channelization*, the *corrections*, and the *correlation* (grey area in Figure 3.1). In the remainder of the paper, we will refer to the parameters and requirements summarized in Table 3.1[2].

3.2.1. ALGORITHMS PROFILE

CHANNELIZATION

In the channelization, every frequency sub-band is split into N_{ch} narrower frequency channels by trading time resolution for frequency resolution. This is performed through a poly-phase filter consisting of N_{ch} N_{taps}-taps finite impulse response sub-filters (FIR) and an N_{ch}-point complex-to-complex Fourier transform (FFT) taking as input the outputs of the sub-filters. As a reference, Figure 3.2 shows a block diagram of the CSP poly-phase filter. Both FIR and FFT algorithms operate on complex samples. Every channel-tap combination in the sub-filters has a different real weight, but all stations and polarizations share the same weight. Given the parameters summarized in Table 3.1 and considering a Radix-2 Cooley-Tukey implementation for the FFT, the aggregate computing requirements of this step for processing all N_{band} sub-bands are 27.6 tera operations per second, with an input bandwidth of approximately 5.9 terabits per second [114].

[2] At the time of the submission, the specification of the SKA1-Low are not yet fully finalized. Values reported in this work represent an educated guess of the worst-case scenario requirements [10].

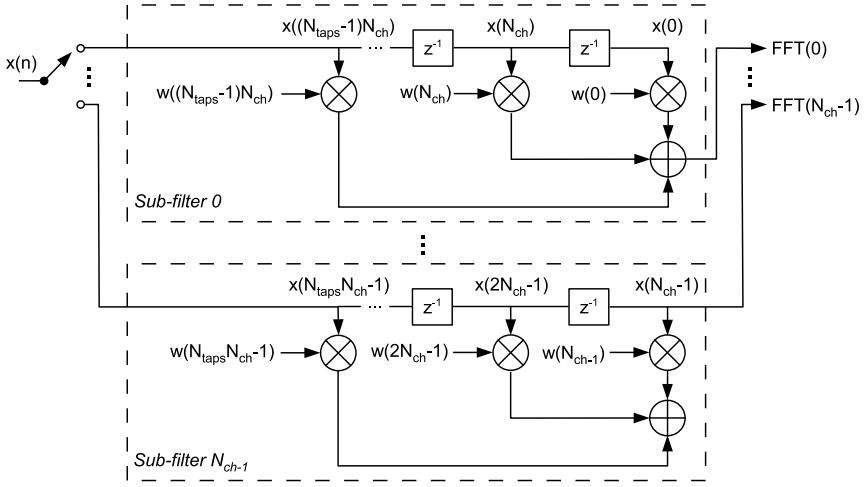


Figure 3.2: Block diagram of the CSP poly-phase filter.

PHASE SHIFT AND BANDPASS CORRECTION

Both phase shift and the bandpass corrections are implemented by multiplying each sample with complex (phase shift) or real (bandpass) coefficients [129] after the channelization. The computing requirements of these two steps are approximately 5.2 tera operations per second. In the case of the phase shift correction, additional computing power may be needed calculating the coefficients, which changes over different time windows. For the sake of simplicity, in the remainder of the paper, we will not discuss the execution of the correction steps, as their computing requirements are negligible with respect to the requirements of the channelization algorithms, and in particular of the correlation algorithm.

CORRELATION

In this step, simultaneous samples of each pair of stations (baselines) are correlated by multiplying a sample with the complex conjugate of the sample of all other stations, and integrated over a period of time (the *dump time*). For each baseline, four visibilities are calculated, i.e., correlating each one of the two baseline's polarizations with itself and the other. This is performed for each one of the $N_{band} \times N_{ch}$ frequency channel in which the signal spectrum has been divided by the channelization phases. The correlation shows a computing requirement of 1.5 peta operations per second (approximately 2 orders of magnitude higher than the one shown by the channelization step), making it the most expensive computing step of the CSP.

An additional difficulty in implementing the correlation step is that on conventional computer architectures, the channelization and corrections steps in general produce data in an order unsuitable for the correlation step [128], which may influence the performance (time-wise) and power efficiency of the entire CSP.

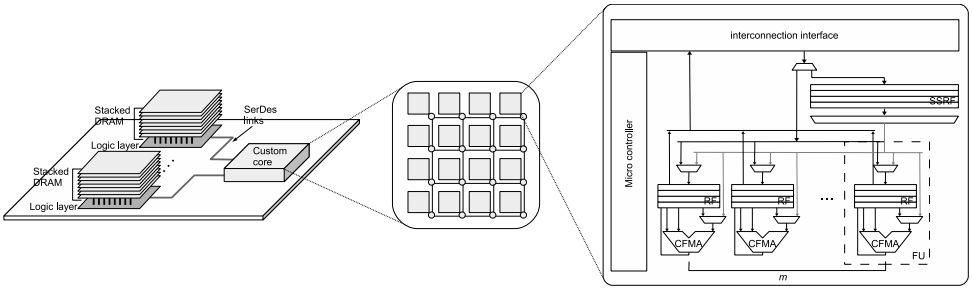


Figure 3.3: Overview of the proposed accelerator architecture. From left to right: socket, chip, core.

3.3. PROPOSED MICRO-ARCHITECTURE

As presented in Section 3.2, the CSP is composed of several algorithms with small repetitive kernels that process data with relatively little locality and with streaming behavior. Although the computing requirements are enormous, the application is embarrassingly parallel. Given the characteristics of the application, we propose a customized Single-Instruction Multiple-Data (SIMD) micro-architecture suitable for satisfying the energy efficiency requirements of the CSP, while still maintaining a good level of flexibility.

Figure 3.3 presents an overview of the proposed custom architecture. Several 3D-stacked memory devices, such as Micron's Hybrid Memory Cube (HMC) [143] [144], are connected to the accelerator die through high-speed Serializer/Deserializer (SerDes) I/O links. The accelerator consist of several homogeneous cores, connected by an on-chip interconnection network. Figure 3.3 shows the CSP custom core micro-architecture template. Each core is composed of m functional units (FUs). Each functional unit is composed of its own private register file (RF) and of an execution unit (represented in the figure by the Complex Fused Multiply-Add - CFMA). Data are transferred from/to the memory exploiting wide path accesses into/from the register files of the functional units. An additional register file (the Shared Scalar Register File - SSRF) is used for storing scalar values shared by all m functional units. As will be explained in Section 3.3.3, all CSP algorithms can be expressed in form of a multiplication of a vector with a scalar value. The SSRF serves as a buffer into which values are written as vector to exploit the wide memory access of the memory, and read as scalar. The core is controlled by a state machine (micro-controller) that manages a predetermined sequence of storage, communication, and computation steps [145]. All functional units work at lock step under the management of the micro-controller, performing the same operation on different data. As also explained in Section 3.3.3, each core operates interdependently on its own set of data.

3.3.1. FUNCTIONAL UNIT

The analysis of the algorithms in Section 3.2 shows that the CSP operations are dominated by the correlation. This function can be implemented, by including also the addition due to the integration over the dump time, as a Fused Multiply Add (FMA) between two incoming complex samples and the temporary result of the integration.

The complex FMA (CFMA) can also serve as functional unit to implement the other algorithms of the CSP. The FIR operation contains N_{taps} real to complex multiplications

and $N_{taps} - 1$ complex additions, corresponding to $4N_{taps} - 2$ real floating-point operations. By using a CFMA, the FIR can be expressed as N_{taps} CFMAs, equivalent to $8N_{taps}$ real floating-point operations, leading to a theoretical utilization of the functional unit of approximately 47% for N_{taps} equal to 8.

The FFT can be similarly expressed in terms of CFMAs [146] [147]. For instance in the case of a Radix-2 Cooley-Tukey implementation, a butterfly between complex operands requires 10 floating-point operations, and it can be implemented as two CFMAs. The theoretical utilization of the functional units in the FFT is therefore equal to 62.5%.

In the case of a phase shift and bandpass correction (if separated from the channelization), the utilization is equivalent to 75% and 25%, respectively, because of the simple multiplication to complex coefficients in the first case and to real ones in the second case.

Even if the theoretical utilization is sub-optimal for FIR, FFT, and corrections, the computational weight of the correlation makes the overall system-level utilization of the functional units close to 100%. Therefore, possible advantages of using functional units customized for the specific algorithms are not justified by the negligible improvement they could bring at system level.

Each CFMA is provided with a private register file, which allows a full throughput in the utilization of the CFMA to be achieved, i.e., a number of interleaved threads equal to the number of pipeline stages of the execution unit is supported. Given the high parallelism of the algorithms, it is possible to schedule threads without being influenced by data dependencies [148]. To achieve a least-energy solution, we assume two different clocks for the CFMA and the register file, trading parallelism versus pipelining in the access to the register file [148] [146].

3.3.2. MICRO-CONTROLLER

The micro-controller controls the entire processing path, i.e., the reading of operand values from memory, transferring these to the functional units, triggering the subsequent processing of these operands, and writing the results back to memory [146] [145]. The micro-controller uses a new type of low-level programming model that allows us to program the memory accesses, data transfer and computation in a more tightly coupled fashion. This allows us to remove a large part of the logic overhead (e.g., buffering) that typically resides between the memory circuits in which the data resides and the functional units that process the data, and which is inherently part of conventional general-purpose processing architectures. By also integrating selected memory controller functions, in particular address mapping and access scheduling, the micro-controller can further improve the overall performance by optimizing the bandwidth utilization and power efficiency of the memory system.

The programming model of the micro-controller is based on two main concepts. The first one is a programmable state machine, called B-FSM [149], which makes it possible to support efficient multi-way branch capabilities in the micro-controller. This allows us to evaluate many conditions in parallel, including loop conditions as part of address generation, and access scheduling related conditions.

The second enabling technology is a programmable mapping scheme that can be used to adapt the mapping of addresses over the memory banks, with the objective to

improve memory bandwidth utilization based on improved access interleaving to reduce bank contention [150].

3.3.3. PROPOSED DATA ORGANIZATION AND ALGORITHMS EXECUTION

To exploit the high available parallelism and the potential of the proposed SIMD architecture, we organize data to better fit the wide line of FUs. As observed, the correlation is the computationally most intensive algorithm, and data should be delivered to the functional units in a way that maximizes their utilization, while minimizing data movement. In this work, we refer to a correlation implementation based on the *input-buffered* architecture [151]. All input samples of the antennas processed by a specific core are first copied to a memory that holds enough samples for the integration over the dump time. The samples of one pair of antennas (baseline) are multiplied and summed to the temporary result until all samples in the dump time have been processed. Then, the result is delivered and a new baseline is processed, continuing in this way until all pairs of antennas have been correlated.

We can assume data to be delivered at the input buffer of the CSP in the form:

$$data\_in[stations][time][polarizations]$$

in which the *time* dimension includes the results of the first channelization, i.e., the "distribution" of the time samples over multiple sub-bands. Similarly to the LOFAR implementation [129], we rely on a corner turn and perform a first parallelization of the computation over the sub-bands (see Figure 3.1). After the corner turn, we assume data organization at the input of each node processing a sub-band in the form:

$$data\_in[time][stations][polarizations]$$

FIR AND FFT EXECUTION

The FIR and the FFT are performed by accessing data in memory per column instead of per row, in a way similar to what is described in [152] for the CELL processor. Each functional unit is therefore responsible for processing samples of one of the polarizations of a single station, both for the FIR and the FFT. Figure 3.4 shows a block diagram of the execution of the FFT over our micro-architecture in the case of an in-place Radix-2 Cooley-Tukey FFT implementation, optimized for CFMAs [147]. Twiddle factors are pre-calculated and organized in memory to optimize their utilization in the successive stages of the FFT calculation. We exploit the parallelism available in the core to perform the algorithms over multiple stations/polarizations at the same time, and use wide accesses to retrieve sample data from the memory. In the figure, $st_{i,j}$ is the $i-th$ station with polarization j; ch_i is the $i-th$ channel generated during the FIR step, and tw_i is the $i-th$ twiddle factor used for calculating the FFT butterfly.

Every functional unit works on its own column of data. As the core's functional units work in parallel and at lock step, all the data in the retrieved memory block/line is processed at the same time, therefore efficiently exploiting the wide line of functional units. Moreover, in the case of the FFT, by accessing data in this way, there is no need for special hardware for shuffling data or for performing any transpose operation, and input data locality can be exploited in the core by implementing a decomposition in smaller FFTs [152] or/and by using higher radix orders.

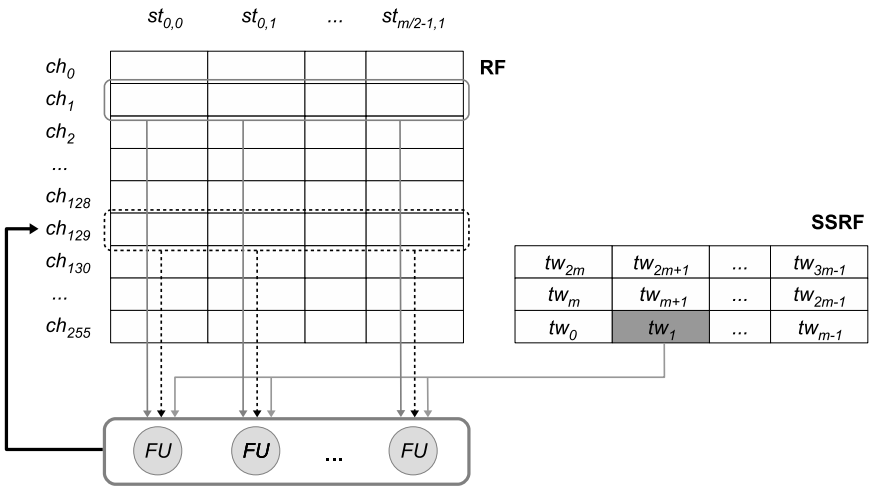


Figure 3.4: Proposed input data organization for the FIR and the FFT algorithm. The picture also sketches how data are processed by the proposed micro-architecture when implementing the FFT.

In the execution of the FIR and the FFT, each basic step implies that each FU multiplies its input sample data with a value common to all FUs (the tap weights in the case of the FIR, and the twiddle factors for the FFT). The SSRF is used to store these values and provide them to the FUs.

CORRELATION EXECUTION

By distinguishing between different channelization batches over the time, data after the channelization are provided to the correlation in the form:

$$data\_in[time][channels][stations][polarization]$$

Figure 3.5 shows the same diagram as in Figure 3.4, but in the case of the correlation. For each time sample t_i, a line of station data is copied on the SSRF, and, one station data at the time, multiplied with the line of station data retrieved for the line of FUs. Temporary results of the integrations are stored in the RFs until all time samples within the dump time for the calculated correlations have been processed and added to the temporary results. This implementation requires the RF to have enough local storage to store the intermediate results of the integrations, i.e., given a line of m FUs and a number of SSRF entries equal to $SSRF_{entries}$, each RF should be big enough to store $m \times SSRF_{entries}$ temporary integration results.

The use of the RF and the SSRF, together with optimized data scheduling, allows us to reduce the number of accesses to the main memory of the accelerator, and therefore the requested data traffic from it. Figure 3.6 shows a simplified example of how data can be scheduled on the proposed architecture when implementing the correlation. In the example, we consider 8 lines of m station data (A to H), i.e., $8 \times m$ stations to correlate (single polarization, in the example). The figure shows the sequence of data loads on the RFs and on the SSRF. For sake of clarity, Figure 3.6 shows only the stations involved in the

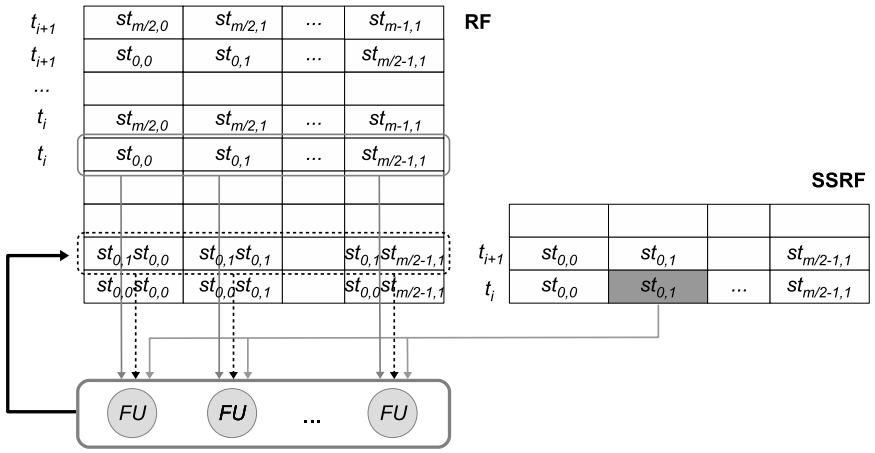


Figure 3.5: Proposed input data organization for the correlation. The picture also sketches how data are processed by the proposed micro-architecture when implementing the correlation (one channel).

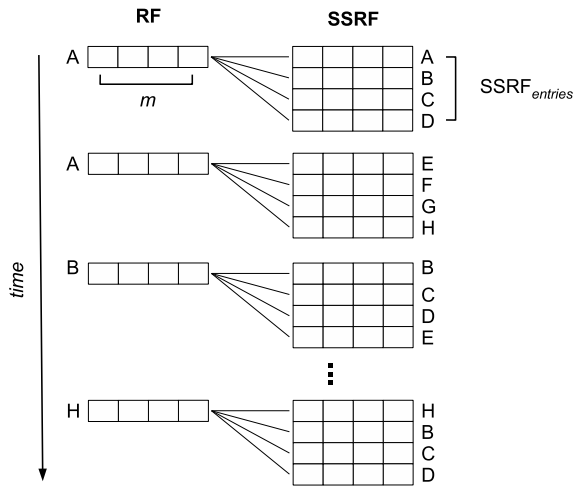


Figure 3.6: Overview of an optimized input data scheduling on the proposed architecture for the correlation algorithm.

correlation, and not each station's time samples. From the figure, it is straightforward to derive that the number of read accesses to the memory and the input bandwidth is proportional, in the case of the SSRF, to:

$$RDs(mem, SSRF) = \frac{stations}{2m}\left(\frac{stations}{m} + 1\right) \tag{3.1}$$

In the case of the RFs, the number of read accesses is proportional to:

$$RDs(mem, RF) = \frac{stations}{2m \times SSRF_{entries}}\left(\frac{stations}{m} + 1\right) \tag{3.2}$$

Note that by reorganizing the data in the way presented before the channelization, there is not need to transpose data before the execution of the correlation, as is currently done in conventional computing architecture [129].

3.4. EVALUATION RESULTS

In this section, we present the results of the design-space exploration we performed on our architecture in order to find the most energy-efficient configuration.

3.4.1. EXPERIMENTAL SETUP

We modeled the CFMA by extracting implementation data for a single-precision floating-point FMA from [148] and scaling them to 14-nm technology, likely to be commercially available in 2018, when the construction of SKA1 will begin. We considered a non-optimized CFMA composed of four FMAs, with twice the number of pipeline stages of the FMA. As our goal is to reduce the total energy, we focused on a low V_{dd} and high V_{th} implementation, i.e., synthesized with a low-power technology library [148]. For modeling the register files and the SSRF, we relied on CACTI-D [153], scaling the results obtained from the tool from 32 nm to 14 nm. On-chip interconnections were modeled on implementation results extracted from [154]. Power and area numbers for the micro-controller are preliminary synthesis results for a 14-nm implementation. Numbers for I/O chip-to-chip data transfers over the backplane were estimated by scaling to 14 nm power and area results obtained for an electrical transceiver implemented in 65 nm and capable of up to 15 Gb/s data rate [155]. For the technology scaling, we referred to the trends reported in [156] and [157] in the case of Low Operating Power technology. For logic, we applied a scaling factor of 0.53x per technology node for the area, and a scaling factor of 0.8x per technology node for dynamic and leakage power. For SRAM devices, from 32 nm to 14 nm, we considered an overall area scaling factor of 0.34x and a power scaling factor of 0.56x. We considered a 30% increase per technology node in the maximum frequency of the design.

We model the 3D-stacked memory modules after a Micron study showing that the energy per bit for access in the first-generation HMC is measured at 10.48 pJ, of which 3.7 pJ/b in the DRAM layers and 6.78 pJ/b in the logic-layer, which includes HMC memory controllers, short-range high-speed Serializer/Deserializer (SerDes) link controllers, a crossbar switch, and additional logic [143] [144]. As the logic layer is implemented in 90 nm, we scaled the logic layer access energy to 2.1 pJ/b to take into account of the most recent technology likely to be used in the second-generation HMC [35]. The energy

Table 3.2: Characteristics of the Pareto point with minimum power (α) and minimum die area (β). Values are reported for correlating all stations for each N_{ch} channels of one sub-band.

Point	Power (W)	Area (mm^2)	GFLOPS/W	$SSRF_{entries}$	RFs_{size}/core (KB)	RF_{conf}
α	11.47	14.54	257	4	268	SRAM, N
β	12.04	9.73	245	2	134	eDRAM, N

access values obtained are in line with numbers reported in other studies [158]. We assume an additional 1.33 W of consumption in the logic layer for components other than the SerDes links, and a background DRAM array power equal to 10% of the DRAM power consumption at maximum bandwidth [144]. An additional 1 pJ/b has been considered for crossing the short-range links between the core accelerator chip and the HMC devices [143]. Each HMC device has a maximum of four full-duplex 16-lane links with an aggregate link bandwidth of 480 GB/s [35].

3.4.2. DESIGN-SPACE EXPLORATION

As the requirements of the correlation dominate the application, we focus on them the system-level design. Performance and costs of executing the channelization algorithms on the architecture will be evaluated afterwards.

CORRELATION

We explored different configurations of the architectural template presented in Section 3.3. While focusing on performing our operations on 64-bit-wide complex operands (32 bits for both the real and the imaginary part), we varied the number m of FUs per core, the number of SSRF entries, and the corresponding size of the RFs. As every FU practically operates on its own set of data independently of the other FUs, we considered each core's RF either as big memories with wide accesses or as m smaller memories with narrow accesses. We explored different memory implementation technologies considering for RFs either SRAM or embedded DRAM (eDRAM).

Each CFMA, implemented as a 6 pipeline stage unit, operates at 433 MHz. Each core is therefore able to deliver a maximum throughput of $m \times 433$ MHz. Evaluations were performed by considering that our accelerator can process all N_{ch} channels of one sub-band. In those configurations in which a single chip was not able to satisfy the design constraints (e.g., the maximum allowed core-memory bandwidth), additional area and power overhead were considered for a multi-chip implementation.

To allow a streaming behavior of the application and overlap execution with memory transfers, we doubled the size of all memory components. Given the requirements in Table 3.1, the minimum amount of memory needed for a streaming input-buffered implementation of the correlation is equal to 9.375 GB. We considered therefore a minimum of three 4-layered 4-GB HMC devices, for a total capacity of 12 GB per accelerator.

Figure 3.7 shows the results of the design-space exploration performed on our custom architecture, by considering the inverse of the number of GFLOPS per Watt and the inverse of the number of GFLOPS per unit of area (mm^2) as design objectives. The power evaluation includes all the components modeled. Area values refer only to the accelerator die. Table 3.2 summarizes the main characteristics of the points in the Pareto

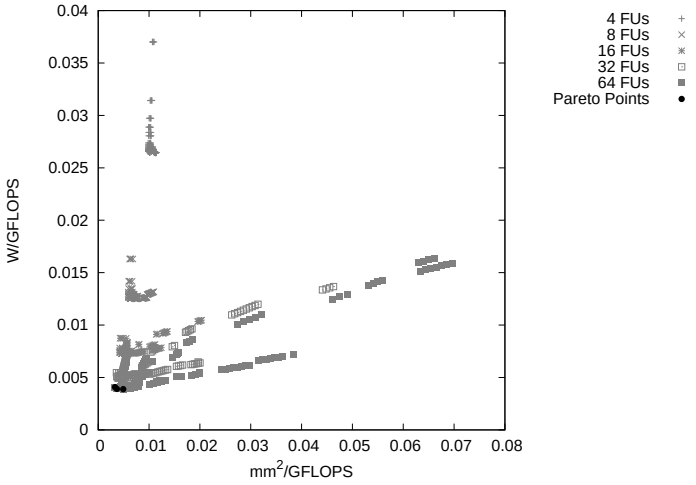


Figure 3.7: Results of the parameters design-space exploration for the proposed architecture.

curve with minimum area and minimum power, α and β, respectively. For both points, the number m of functional units per core is equal to 64 and the number of cores in the chip is 14. The throughput from the HMC for the α and the β point is equal to 68 GB/s and 81 GB/s, respectively. The different throughputs are due to the difference in the $SSRF_{entries}$ parameters and to the influence that $SSRF_{entries}$ has on the number of memory accesses. The throughput to the HMC is in both cases equal to 18 GB/s. In the table, $SSRF_{entries}$ is the number of SSRF entries, and RFs_{size} is the total size of the RFs in the core. RF_{config} specifies the memory implementation technology used for evaluating the memory component and whether it employs narrow (N) or wide (W) accesses. The utilization of the functional units is equal to 95.1%.

As noted, every core works on its dedicated memory space. The amount of main memory needed per core for implementing the correlation is equal to approximately 686 MB. The size of a 4-GB HMC vault, i.e., a vertical memory slice in the stacked DRAM which is managed by a dedicated controller, is equal to 128 MB [35]. By allocating six vaults to each core, the memory access requests can be served in parallel by the controllers without contention penalties.

Figure 3.8 shows the power breakdown in the case of the correlation while varying the number of FUs per core. The histograms are normalized with respect to the configuration using 4 FUs per core. For each core configuration, we chose the point that minimizes the product $power^2 \times area$. As modeled by Equations 3.2 and 3.1, by increasing the number m of FUs per core, the power reduces approximately as $1/m$. This is mainly due to the reduction of the number of micro-controllers needed and to the reduction of the communication between memory and cores, which decreases the memory access power, the HMC links power, and the on-chip communication power. By exploiting the locality of the station data in the correlation using wider cores, the number of accesses to the memory is significantly reduced. Figure 3.8 also shows that the main contribution

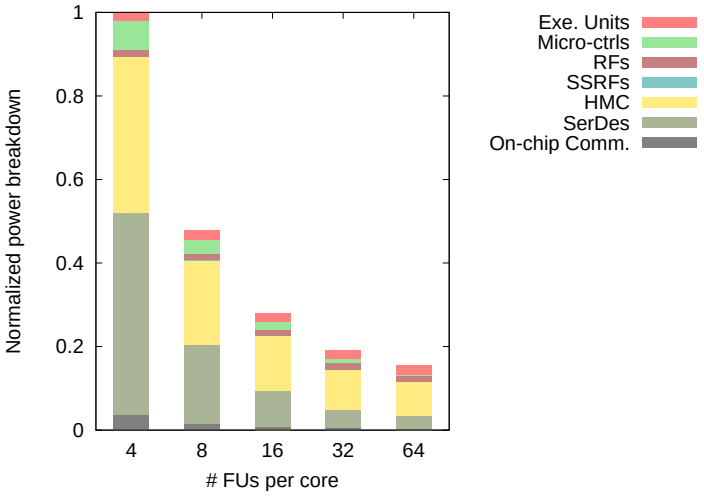


Figure 3.8: Power breakdown for different core configurations.

to the power consumption is due to the accesses to the main memory. Together with the SerDes communication cost, the memory accesses account for around 73% of the power budget for the most energy-efficient configuration. As the computing capabilities needed are the same for each configuration, the power consumption due to RFs and execution units is approximately constant for all configurations. Both power consumptions are dominated by the dynamic components. Given the relatively small size of the SSRF, its contribution to the overall power consumption is negligible. By using 64-wide cores, we are able to save approximately 85% in power with respect to a 4-wide core configuration.

In Figure 3.9, we analyze the variation of the power consumption of the architecture components by varying the size of the SSRF and the RFs. We show results for the configuration with 8 functional units per core that minimizes the $power^2 \times area$ product. The configuration with minimal power (SSRF equal to 32) corresponds to the point at which the reduction in the memory and communication power equals the increase of the power of the RFs. In fact, while the increase of the size of the SSRF decreases the memory-core communication (as shown in Equation 3.2), it also increases the size of the RFs and consequently the cost associated with accessing its elements.

CHANNELIZATION ALGORITHMS

As can be derived from Table 3.1, the channelization step requires 12.2 Giga CFMAs per second for each sub-band. A single core with 64 FUs provides sufficient computational power for implementing the channelization. Therefore, we evaluated the execution of the channelization algorithms by considering an additional core with 256 MB of allocated memory space in the system. By implementing the FIR and FFT as described in Section 3.3.3, each single functional unit, working at 433 MHz, processes eight stations.

To exploit the locality of the input samples, we store several of them in the main

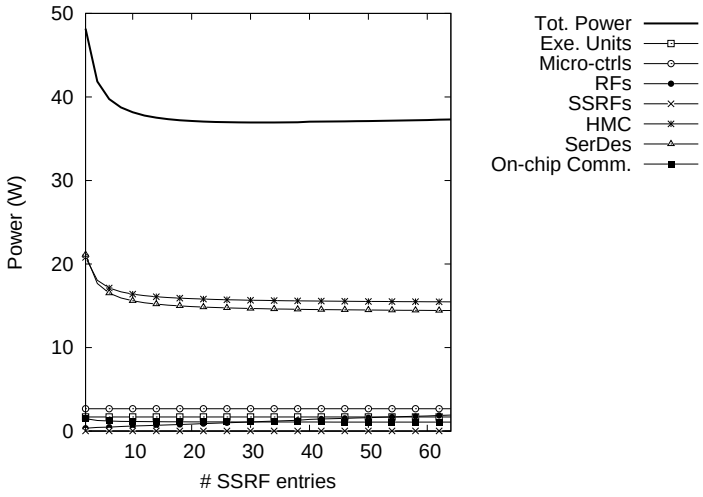


Figure 3.9: Power consumption while varying the size of the SSRF (8 FUs per core).

memory and execute the FIR a sub-filter at the time over those time samples (by taking as reference Figure 3.2, the order of the results would be FFT(0), FFT(N_{ch}), FFT($2N_{ch}$), ..., FFT(kN_{ch}), FFT(1), FFT(1+N_{ch}, FFT(1+$2N_{ch}$), ..., FFT(1+kN_{ch}), etc, where k depends on the number of samples stored). This implementation allows us to reduce the number of input samples read from main memory for each FIR execution to one asymptotically. The FFT step is then performed over all results accumulated in the main memory. A Radix-8 version of the FFT for CFMAs was considered [147]. Once the FFT step has been performed, samples are copied into the memory space of the cores implementing the correlation.

Our implementation of the channelization results in an utilization of the core equal to about 44%. The core requires an additional throughput from the HMC devices to core and from core to the HMC devices of approximately 22 GB/s and 18 GB/s, respectively. The execution of the channelization implies at the node level an additional power consumption of 2.23 W.

By spreading the constant power overhead over the operations performed, the energy of executing an 8-taps FIR is equal to about 2.2 nJ, whereas executing a 512-point FFT in this core configuration consumes 1.38 μJ.

Figure 3.10 compares the normalized power distribution for separate execution of the correlation and the channelization and when considering the two steps together. Accessing the main memory represents the biggest contribution to the power budget. In the case of the channelization, however, this contribution is higher in percentage, because of the relatively small exploitable input data locality of the filtering step, in particular when compared with the execution of the correlation, which presents a more compute-bound behavior [142]. As expected, at application level, the power consumption is dominated by the correlation.

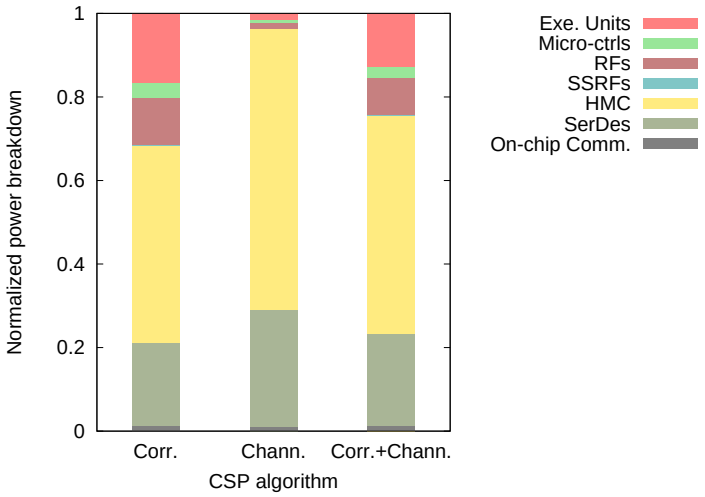


Figure 3.10: Power breakdown for correlation and channelization.

OVERALL SOLUTION

Summarizing the studies presented in the two previous subsections, the CSP can be implemented with 512 nodes (one for each sub-band), each one composed of a 15-core chip with the configuration of point α in Table 3.2, and three 4-GB HMC devices. Each 15-core chip can theoretically deliver around 3.3 TFLOPS. The evaluated CSP algorithms achieve an overall utilization of 92%. The estimated power consumption for the node is 14.4 W, leading to an energy efficiency of 208 GFLOPS/W at application level. The 15-core chip occupies approximately 15.7 mm^2 of die area. The data throughput from and to the HMC devices is equal to 93 GB/s and 37 GB/s, respectively. One 60 GB/s SerDes link for each HMC is therefore sufficient for providing the required bandwidth. The chip input and output data rate are equal to 5.76 GB/s and 2.46 GB/s, respectively. Four transceiver macros are therefore sufficient for providing the needed I/O bandwidth.

3.5. RELATED WORK

The main CSP algorithms have been recently studied and evaluated in several computing platforms, with the goal of optimizing their execution by exploiting the characteristics of the systems targeted. As the channelization algorithms are well known and vastly discussed in the literature, we focus in particular on the correlation implementations.

In [128] and [129], the correlation algorithm was implemented on several architectures, i.e., CPUs, GPUs, and the IBM Blue Gene/P and Cell processors. The authors observed that, given the characteristics of the algorithm, good performance can be achieved when sufficient local storage is available. Both the Cell and the IBM Blue Gene can achieve close to peak performance, whereas CPUs and GPUs reach significantly lower numbers (70% and 40%, respectively), with the Cell the most power-efficient architecture. However by applying a multi-level data-tiling strategy in memory, and by optimiz-

ing the overlapping of streaming data transfer and computation, peak performance of up to 80% can be achieved in modern GPUs for a large number of stations [130].

Research by Souza et al. [132] shows that FPGAs are a suitable candidate for running the correlation algorithm, because of its regular and simple structure. The use of custom data widths allows a potential reduction of the power consumption at the cost of a reduced programmability, flexibility, and algorithm scalability.

With regard to the proposed architecture, many different solutions can be found in the literature for accelerating specific kernels in high-performance computing by exploiting the intrinsic parallelism of the application. While systolic arrays were popular in the 80s [159], more recent works have, for example, studied the acceleration of mathematical kernels, such as general matrix-matrix multiplications (GEMM) [146] [160], or the acceleration of frameworks for the management of big-data workloads, such as MapReduce [144].

While similar in some of the aspects to the related work discussed, our paper presents the first system-level evaluation of the implementation of a custom architecture for accelerating the main algorithms of the SKA1-Low central signal processor. By approaching the problem at the application level, it is possible to optimize the memory utilization by reorganizing data for exploiting wide SIMD cores and the wide access and high bandwidth offered by 3D-stacked memories. This feature is supported in our architecture by the SSRF, which can be used in the execution of all algorithms discussed, and by the micro-controller, which exploits the regularity of the data access pattern for optimizing data movements in the memory layers and the functional units.

3.6. CONCLUSIONS

In this paper, we propose, for the first time, the implementation of a custom architecture for accelerating the main algorithms of the SKA1-Low central signal processor. After discussing the main characteristics of the algorithms and the challenges offered by the size of the problem, we propose an accelerator architecture that takes advantage of the bandwidth and low-power operations of new 3D-stacked memory devices. Our evaluations show that our architecture, thanks to a data organization that supports the execution of algorithms over wide SIMD cores, is able to process all 512 channels of a sub-band for just 14.4 W, achieving an energy efficiency at application level of up to 208 GFLOPS/W.

4

AN ARCHITECTURE FOR NEAR-DATA PROCESSING

The need for ever increasing computing performance can be satisfied by realizing workload-optimized systems. For the class of data-intensive applications, the 'near-data processing' paradigm seems to be a perfect fit. Many interpretations of this theme exist, ranging from very specific solutions, to very generic solutions.

Since the NDP needs to be optimized for the workload at hand, no assumption should be made on its design. It could be a piece of reconfigurable logic able to implement optimized functions, but also a collection of general-purpose (soft)cores running an instruction set. This also reflects the fact that processing in general is becoming a decreasingly relevant part of the entire system. With performance, as well as energy consumption, being dominated by data transport, we put less emphasis on the actual NDP design and more on the data-related aspects.

In contrast to the exact processing capabilities, the heterogeneous integratability of the NDP in the system is of great importance. With the integration of traditional heterogeneous solutions, e.g., GPUs, still not being mature, adding processing capabilities in another non-trivial place of a computer system raises new questions and challenges. A solid integration, supporting virtual memory, coherence, and the ability to access data stored anywhere in the system, is however of great importance to ease the burden on the programmer, and realize good performance without needing tedious tweaking and optimizations. Another aspect of this is data-locality management. Programmers should be able to optimize the data locality in order to maximize performance, but these mechanisms must be straightforward, and whenever possible use existing operating system support.

To minimize data movement, and avoid any bottlenecks related to copying data around, the NDP should be close to the memory the CPU believes is its 'normal' memory, meaning that the NDP can start working on a data set directly after it is allocated by the operating system.

In this chapter an architecture to enable arbitrary near-data processing close to the main-memory of a CPU, as well as a generic data-locality management mechanism, is presented.

The content of this chapter is based on the following two papers:

An architecture for near-data processing
E. Vermij, C. Hagleitner, L. Fiorin, R. Jongerius, J. van Lunteren and K. Bertels
Computing Frontiers, short paper, 2016

An architecture for integrated near-data processors
E. Vermij, L. Fiorin, R. Jongerius, C. Hagleitner, J. van Lunteren and K. Bertels
Transactions on Architecture and Code Optimization
Under review - Major revision

ABSTRACT

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

4.1. INTRODUCTION

The world's population is producing and storing more digital data than ever before, by means of a wide variety of social media, and modern voice-, text- or image-based communication methods. While Facebook and Google are traditional examples of data-driven companies, big-data has found its way to many different businesses and fields [161]. Creating business value from huge amounts of data is becoming an ever more important task for computer systems. Real-time graph analytics, online fraud-detection, and cognitive computing are examples of data-intensive workloads posing even more challenging requirements on computer systems. Because of this, we have seen a growing interest in the role of data in computer systems. Unfortunately, not only are the tasks becoming harder, but also computer systems are becoming relatively worse at handling them: the increase in memory bandwidth and the reduction in memory latency does not keep up with the increase in processing capabilities, making compute nodes increasingly worse at handling data intensive workloads.

One solution to cope with ever increasing data-rates and data-sizes is to use more, relatively cheap, nodes. This is known as a *scale-out* solution, which has gained much attention in the past few years, and works well for embarrassingly parallel workloads. Many frameworks, such as for instance *Spark* [162], exist, which are either generic or workload-specific and offer massively parallel workload distribution on a standard cluster [163].

Scaling-up the hardware, by making it more powerful or more specific for big-data problems, is another way of increasing performance, and is especially suited for problems that do not scale linearly with the amount of nodes. An increase of performance can be achieved in several ways, ranging from the use of accelerators and co-processors, to the support of large and different type of memories. For example, the IBM POWER8 systems offer a high memory bandwidth and large amounts of DRAM per socket, making it an interesting platform for data-intensive applications. GPUs are well recognized as a platform for boosting the performance of data analytics with respect to CPUs, due

to their high bandwidth, parallelism, and latency hiding mechanism [164], making it the fastest single node solution for the data-intensive Graph500 benchmark [58]. IBM introduced the CAPI interface [51] in its POWER product line, enabling the coherent attachment of workload optimized coprocessors to the CPU's memory space, making handling data in an optimized heterogeneous system much easier. With the rise of various types of non-volatile memory [165] [166], we get high bandwidths to large persistent storage volumes, much higher than what disks can offer. All these, and likewise, products have useful memory-related features, but lack others. For example, the CPU's DRAM offers the storage capacity, but only mediocre bandwidths compared to GPUs, and cacheline-sized accesses. GPUs and the Xeon Phi have a high memory bandwidth, but suffer from a high latency and a relatively small storage volume.

To bridge the gap in memory bandwidth and latency between what today's data-intensive workloads demand and what existing computer architectures can offer, the processing should take place as close to the data as possible, avoiding interconnect bottlenecks. This paradigm is referred to as 'near-data processing', and has recently been re-discovered [73], motivated by the availability of new technologies such as 3D stacking, big-data workloads with high degrees of parallelism, and programming models for distributed big-data applications. By introducing processing capabilities very close to the main memory of a CPU, it is possible to benefit from memory access features characterized by low latency, high bandwidth, small access granularity, and support for large memories. In this work, we propose an architectural extension to support near-data processing in an existing CPU ecosystem.

A high-level view of the proposed architecture is shown in Figure 4.1. The essential aspect is the two levels of memory controllers: at the CPU level, we have memory-technology agnostic memory controllers, while the technology-specific memory controllers are tightly coupled to the main memory. An industry example of such a setup is the memory system of the IBM POWER8 CPU [34], which has eight memory-technology agnostic memory channels each connecting to a 'memory buffer' chip, holding four DDR3/4 memory controllers. Another example is a CPU connected to novel stacked memory like the Hybrid Memory Cube [35], having the technology-specific memory controller in the logic layer of the memory device. We propose the addition of near-data processing capabilities close to the technology-specific memory controllers, as shown in Figure 4.1. For completeness, we show default memory channels as well, to highlight the generality of the approach. It would also be possible to extend it with a second CPU socket or even a modern GPU, without breaking the concept of the proposal. The focus of this paper is to discuss the architectural implications of adding the near-data processing capabilities, and to evaluate how it affects software and workload partitioning.

The main contributions of this paper are as follows:

- We propose a novel memory-centric architecture to enable *arbitrary* near-data processing in a contemporary server environment, by introducing a minimally-invasive component at the CPUs' system bus and a component in the memory system;
- We propose generic methods for data allocation and data locality management based on existing NUMA functionality for the near-data processors (NDPs) and

the CPU;

- We propose generic hardware-managed methods for coherence and for accessing data in the global address space;
- We quantify the coherence and data writeback overheads, and show the performance of the proposed architecture for a variety of small workloads as well as the Graph500 benchmark.

The remainder of this paper is organized as follows: Section 4.2 describes related work and the motivation for the proposed research. In Section 4.3 data placement and memory management is discussed followed by the proposed necessary hardware additions in Section 4.4. Section 4.5 presents a detailed discussion on coherence, while Section 4.6 discusses virtual memory management and access to remote data. A custom system simulator is presented in Section 4.7 followed by synthetic results in Section 4.8 and a case study in Section 4.9. Section 4.10 concludes the paper.

4.2. MOTIVATION AND RELATED WORK

4.2.1. INTEGRATING ARBITRARY NEAR-DATA PROCESSING CAPABILITIES

Several platforms have been proposed to deal with big-data applications. A case study of using existing node and cluster technology to execute big-data workloads is presented in [167]. The work in [76] proposes a system and board design around Flash and DRAM, targeting energy-efficient execution of big-data workloads. Another system-design proposal [98] targets the specific class of graph processing algorithms. Work in [88] shows a generic architecture for big-data analytics, using stacked DRAM and its logic layer. In [168], near-data processing has been investigated for accelerating big-data workloads with poor locality in solid-state disks.

Work in [144] [91] [169] [98] [88] proposes architectures all based on exploiting the logic layer found in 3D-stacked DRAM devices, such as the Micron Hybrid Memory Cube (HMC). In [89] the integration of light-weight cores on top of traditional DRAM is proposed, while [83] adds small compute elements close to the DRAM banks.

We differentiate from all this work by not focusing on a particular type of NDP architecture (general purpose, reconfigurable, etc.), a particular type of memory technology (DRAM, HMC, HBM, etc.), or a particular type of applications, but on adding the hardware components and mechanisms needed for integrating near-data processing. This is done under the assumption of a high-level architectural concept as described in Section 4.1 and shown in Figure 4.1. To make a solid contribution, a clear definition of 'integrated' needs to be specified. The objectives of a generic integration of any type of NDP following the architecture in Figure 4.1 are:

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

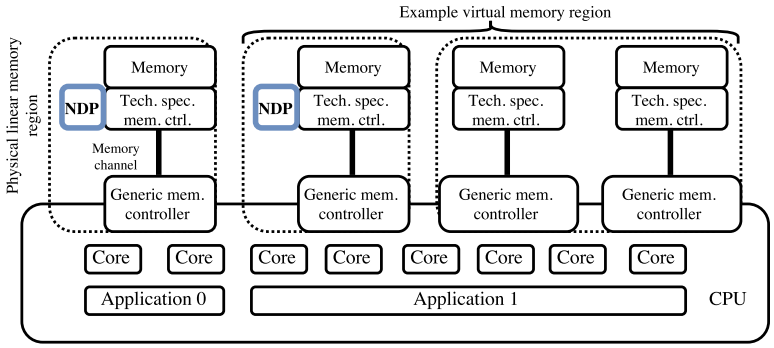


Figure 4.1: The high level view of the proposed architecture, where compute capabilities are added close to the main memory of a CPU.

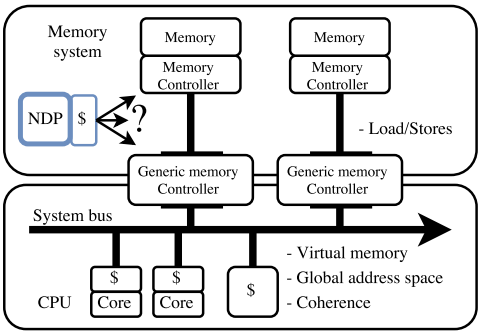


Figure 4.2: The problem of integrating NDPs with a CPU: all relevant integration features stop at the memory controllers.

First, workloads not using the NDPs should not be negatively affected, as that would break the general usability of the system. Second, changes to the CPU must be limited and as non-invasive as possible, to not break the delicate workings, overhaul roadmaps, and incur major costs. Changing non-trivial aspects of the OS kernel in a maintainable way can be considered equally challenging, given the community-like organization, the enormous complexity, and the high costs of quality software engineering. The last two items follow the trend in industry towards coherently attached coprocessors [51], the trend in industry towards a unified and coherent address space between CPU and coprocessor [170], and the literature (many are cited in the remainder of this section) which almost always describes one or more of these aspects to some degree, thus acknowledging the need for them. This is also reflected in a recent expert-opinion overview [171] about near-data processing, which, among others things, stresses the need for a unified shared address space, a unified memory model for both the CPU and the NDPs, communication between the various devices, and a re-visitation of solutions for coherence.

Before proposing a solution, it is necessary to have an understanding of the challenges involved with integrating NDPs to this extent. Figure 4.2 shows both the most

fundamental property as well as the most fundamental problem associated with near-data processing. Being the barrier between the CPU and the memory system, all features relevant for system integration stop at the generic memory controllers. This implies that:

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

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

4.2.2. PROPOSED SOLUTIONS AND THEIR PITFALLS

EXTENDING THE SYSTEM BUS

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

- Several operations on the coherence fabric of the system bus are latency critical, for example a *snoop* or an *invalidate*. Extending these operations to the NDPs would have a dramatic negative impact on system performance [51].
- The hardware TLB *sync* and *invalidate* mechanism to support virtual memory in multi-processor systems [172] are latency critical as well, and extending these operations to the NDPs would also have a dramatic negative impact on system performance.
- For consistency reasons, caches at the CPU have to keep track of an evicted cache line until it is committed to the memory controller. By increasing the latency towards this commitment, caches will require more resources (tracking state-machines etc.) for the same performance.

Therefore, extending the system bus would result in a much poorer CPU performance, which breaks the first goal described in Section 4.2.

VIRTUAL MEMORY SOLUTIONS

Regarding virtual memory support, many NDP solutions propose the usage of a small TLB holding the translations, and describe this TLB to be filled by means of a driver or library [88] [173], or lack the description on how the TLB gets filled [91] [174]. Unfortunately, the effective to real address translation mapping is not static: memory pages allocated by the default OS allocation methods can be subject to physical migration, stealing, and several other operations [175]. For this reason, and many others, the system bus offers the TLB synchronization mechanisms stated before. A software managed solution

would require a deep integration with the OS to monitor and temporally hold these migrations, which is a very difficult task resulting in a non-portable solution. Furthermore, a software managed TLB would be much slower compared to hardware managed ones.

Using pinned or memory-mapped (physically contiguous) memory [90] [83] avoids the need of virtual memory at the NDP but will not work when having data sets with a size close to the main memory capacity due to physical fragmentation, and requiring significant allocation times. Work in [176] proposes to use a novel and completely decoupled page table for the NDPs, allowing for various optimizations and efficient translations, at the expense of integratability. Their proposal requires changes in the operating system and special allocation routines. Several of the already mentioned papers name the exploitation of transparent huge pages [177], or the exploitation of the fact that many page level translations can be grouped into larger chunks due to how the allocators work [178]. But, although important optimizations, they do not solve the underlying synchronization problem. None of the proposed solutions will provide true virtual memory support without substantial OS modifications, which is in contrast to the second goal of limiting OS changes as much as possible.

COHERENCE SOLUTIONS

Regarding coherence, various options have been explored. Putting the NDPs in a separate address space [98], with explicit data management by a driver, avoids the need of coherence, but is also not an integrated solution. Marking pages cache-inhibited [179] [98] [83] [90] makes working with them incredibly slow from the CPUs perspective as every access has to go through the non-cacheable unit [180], which is designed for managing memory-mapped devices, not for handling large amounts of data. Work in [88] proposes the identification of a single cache in the entire system which can hold a certain page, which is stored and managed via the page tables. This is a strong idea as it makes things explicit and thus simple, but also raises many issues. For example, the method is not scalable to larger systems as the page tables have only so much free bits to identify caches, assumes that all memory accesses are done via a TLB which is not the case for hardware prefetches, and it limits the sharing of data between cores. Lazy methods based on a rollback mechanism have been proposed in [181] but the paper lacks details required for a detailed comparison (e.g. description of the rollback mechanism and the proposed 'conflict detection hardware'). Work in [81] proposes a form of integrated near-data processing offloading small instructions to the memory system, but requires significant changes to the delicate inner-core of a CPU, and lacks significant details on the used coherence methods and their performance. It for example assumes that data can only be in the local last-level cache of a core, which will not be true for modern CPUs. Work in [91] proposes a generic hardware-managed method for coherence, but lacks the ability to shared data between CPU and NDP, and furthermore does not support the access of remote data by an NDP, limiting the useability of the proposal. Handling coherence manually (flushing cache lines explicitly) is both hard and unpredictable as the hardware prefetchers can prefetch data the user/programmer believes is still in memory. Besides, software driven cache flushes are very slow, especially on multi-socket systems.

Furthermore, both the cache-inhibited and the manual approach above do not provide us with a consistency model keeping track of when a write from the CPU has actu-

ally appeared in DRAM and can thus be read by the NDP. Without a hardware managed coherence mechanism, a hardware synchronization mechanism is still needed to solve this issue. Other related works do not discuss coherence [173] [174] [99]. None of the proposed solutions will provide a workable coherence and consistency model without breaking the goals stated in Section 4.2. A similar analysis holds for accessing non-local data: none of the above mentioned works describes how an NDP can issue a load at the CPU's fabric to access data stored in a CPU cache, in another memory channel, or on another socket.

IBM's CAPI interface [51] solves a part of the problem: it offers a device a coherent view towards the CPU, but not towards local memory. This is visualized in Figure 4.3b, where CAPI provides the line from the NDP towards the CPU, but not the lines towards the memory.

MEMORY MANAGEMENT SOLUTIONS

The problem regarding memory management is that the CPU wants its data striped over the various memory channels, while the NDP wants entire data sets to reside in the same memory channel. Furthermore the OS does not allow the allocation of data within a specific memory channel. The work presented in [91] solves this by using a custom allocator (or extending the OS allocator) which is able to allocate memory in both a striped as well as a contiguous fashion. Research in [176] describes the use of a custom allocator as well, again requiring OS modifications. Both these works break our goal of limiting the OS changes as much as possible. Work in [90] and [101] describes the concept of page level distribution of data across memory channels, but lacks a description of how that is implemented/realized.

4.2.3. CONCEPT OF THE PROPOSED SOLUTION

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

4.3. DATA PLACEMENT AND MEMORY MANAGEMENT

By default, the POWER8 CPU stripes accesses to subsequent cache lines across different memory channels, to optimize bandwidth for common access patterns [34]. In this case, all memory channels represent a single big contiguous memory space called a *group*, with a certain modulo or hashing scheme to map accesses to the right channel, illustrated by the two default memory channels in Figure 4.1 contained in a single contiguous memory region. It is, however, also possible to make every memory channel its own *group*, implying that every memory channel holds a contiguous part of the physical address space, shown by the NDP memory channels in Figure 4.1. The actual configuration is set in firmware and typically multiple groups are only used when memory channels

have different types or sizes of memory. For the architecture presented in this work, however, every memory channel constitute its own group, and by doing this, data sets can be mapped into a single memory channel. The methods to achieve this and performance implications of this are discussed below.

4.3.1. NDP ALLOCATIONS

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

The OS builds its NUMA organization tree [182] based on information supplied by the firmware. This tree is a hierarchical representation of nodes and possible subnodes in the system, including both the CPUs and the memory. We can adjust the firmware in such a way that it reports the various memory channels and their physical address regions as NUMA memory nodes, as is shown in Figure 4.3a. In this way we can use industry standard functionality to bind a process to a specific memory channel using *numactl*, or bind a memory allocation to a specific memory channel using the *numa_alloc_onnode()* functionality supplied in *numa.h*, all possibly hidden in an NDP runtime library. By doing so, we introduce, from an NDP point of view, the concept of *local* and *remote* data. *Local* data should, whenever possible, be stored in the memory of the specific NDP, while *remote* data can be stored anywhere, even on a different socket or in GPU memory. It should be clear that managing data locality is the key for obtaining reasonable NDP performance on the proposed architecture, this is however exactly the same as the optimizations usually performed on existing NUMA systems.

In case a data set is allocated without a specific NUMA binding, but is accessed for the first time by an NDP, the data locality solves itself. In this case, the NDP access will raise a page-fault (discussed in Section 4.6), and the OS is asked for physical backing of the accessed virtual memory. Since the request for physical memory comes from a specific NUMA domain (the NDP), the physical pages will be taken from the NDPs' local memory. This is a very valuable effect foremost of interest when the NDP initializes data sets.

4.3.2. DEFAULT ALLOCATIONS

To make sure default allocations (i.e. NDP unrelated) do not end up in a single memory channel, dramatically reducing the achievable bandwidth to that data set and thereby breaking with the integration goals, extra care is needed. This can be solved by configuring the CPU node as having no memory of its own (*MEMORYLESS_NODE*), and change the default OS allocation policy to *INTERLEAVE*. That way a page request will be forwarded, in a round-robin fashion, to the other nodes: the memory channels, either with or without NDP. By doing so, we realize a best-effort page level distribution of data across

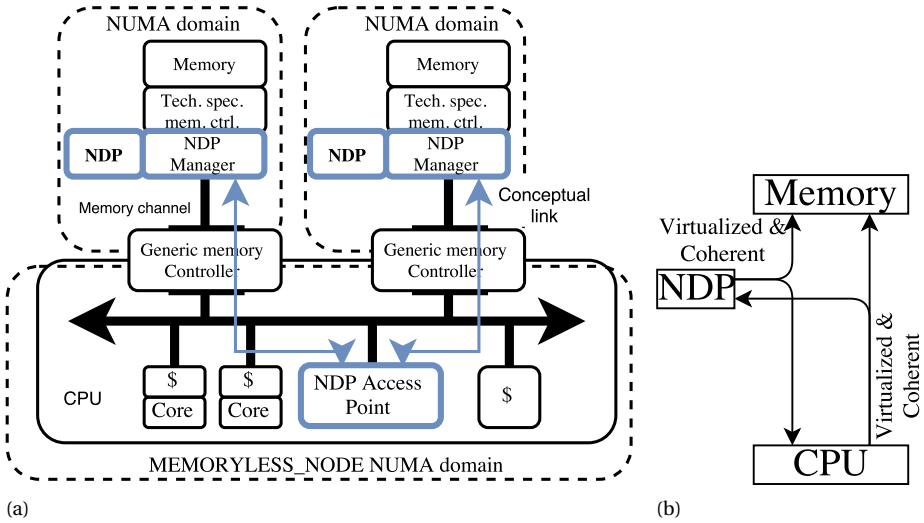


Figure 4.3: (a) A CPU environment enhanced with near-data processing capabilities. By introducing a component in the memory system and one at the system bus, we can extend the necessary integration features towards the NDPs, supporting arbitrary functionality. This connection is illustrated with the two arrows. (b) Both components have a virtualized and coherent view towards each other, and towards the main memory.

the available memory channels. Accesses to a single page still have to go through the same memory channel, but if we have many threads running at the CPU, as we have in modern big-data applications, the bandwidth achievable on a system-level will be like the striped cache line approach, as also indicated in [88]. This mechanism does not imply a large penalty at allocation time, as physical page allocation is only done when the page is touched, and forwarding the allocation to another NUMA node is nothing more than traversing a small data-structure in the OS routines.

4.4. HARDWARE EXTENSIONS TO ENABLE NEAR-DATA PROCESSING

In Figure 4.3a we show the proposed architectural extension. At the memory side we introduce the NDP Manager (NDP-M), and at the CPU side we introduce the NDP Access Point (NDP-AP). Between the two components, a communication channel exists. In the remainder of this section these components will be discussed.

4.4.1. COMMUNICATING WITH THE MEMORY SYSTEM

As discussed, we include processing capabilities at the memory-side of the generic MC. To not interfere with the existing operations of the CPU, the communication with an NDP must be done *on top* of existing functionality. We added a lightweight NDP messaging extension to the protocol of the high-speed serial link of the memory channel. Instead of being load/store-only, it is extended to be able to carry both the original traffic, as well as communication between the NDP-M and NDP-AP introduced below. This is

shown in Figure 4.3a with the two arrows. Communication between the NDP-AP and the generic memory controllers is done by using direct-addressing on the system bus. To ensure the correct functioning of the coherence mechanisms described in this work, messages can by default not overtake 'normal' cache lines traveling up or down the memory channel.

4.4.2. NEAR-DATA PROCESSOR MANAGER

To support virtual memory, coherence, remote accesses, and communication capabilities for NDPs in the memory system, we introduce the NDP-M, as shown in Figure 4.3a. In Figure 4.4, we show the architecture of the NDP-M, with its main functional blocks, described below. On the bottom the interface towards the high-speed link is shown, and on the top the interfaces towards the memory controllers. On the side there are two types of interfaces for the NDP: one for handling messages between the NDP and applications, and one or several interfaces for the memory accesses. The NDP-M is responsible for:

- Offering an interface for any type of NDP.
- Managing coherence and consistency between the CPU and the NDP; the former by means of a distributed directory stored in DRAM, discussed in Section 4.5, and represented by the *Coherence Manager* in Figure 4.4.
- Translating the virtual addresses received form the NDP, discussed in Section 4.6, and represented by the *ERAT* (effective-to-real translation) in Figure 4.4.
- Providing a gateway to access non-local data, discussed in Section 4.6.3, illustrated by the *NDP Remote Rd/Wr Q.* in Figure 4.4.
- Providing information towards the NDP-AP (described below) about address ranges involved with near-data processing.

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

4.4.3. NEAR-DATA PROCESSOR ACCESS POINT

The CPU needs to support the NDP-M in all its needs regarding address translations, remote accesses, etc. To avoid a significant software overhead and unnecessary CPU load, we introduce a single hardware component attached to the system bus, called *near-data processor access point* (NDP-AP). From the NDP-Ms perspective, the NDP-AP is the access point into the global coherent memory space. From the software perspective (either OS or user), it is the component handling communication with the NDP-Ms. In Figure 4.5, we show the architecture of the NDP-AP, with its main functional blocks. The NDP-AP is responsible for:

- Issuing load and stores on remote data on behalf of the NDP-M(s).

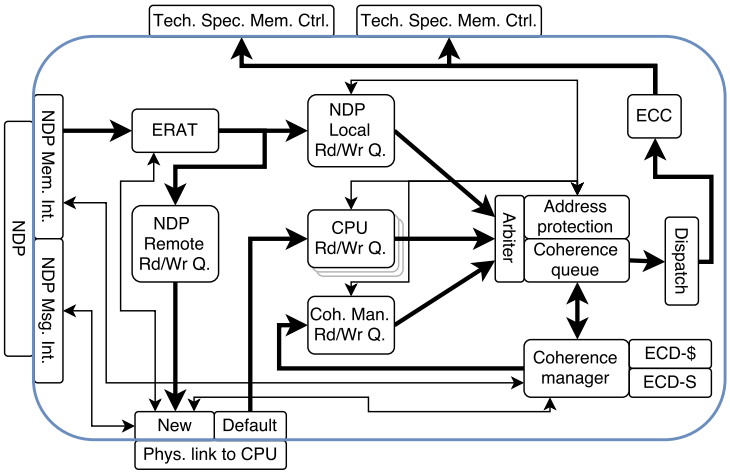


Figure 4.4: Main functional blocks within the near-data processor manager (NDP-M). In reality, the CPU Rd/Wr queue can consist of several queues, for example a separate one for prefetch loads. For simplicity, only the forward path towards the DRAM is shown.

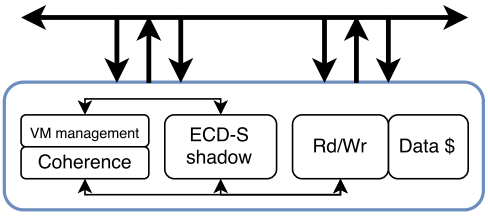


Figure 4.5: The main components in the near-data processor access point (NDP-AP). The fat arrows indicate data ramps.

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

Applications, or the OS, can send messages to the NDP-AP by means of a special instruction (like *icswx* for the POWER ISA [183]), which forces a cache line on the system bus directly addressed towards the NDP-AP. This gives us the possibility to dispatch messages with very little latency and overhead. The NDP-AP can send messages to the NDP-M by generating a special transaction on the system bus, directly addressed towards the correct generic MC, with the reverse path working likewise.

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

4.4.4. THE SYSTEM BUS

Access to the system bus is necessary to communicate between the NDP-AP and the NDP-Ms. We base our work on a system bus as found in modern CPUs such as the POWER8. Such a bus is a capable interconnect fabric supporting the integration features described in Section 4.2. POWER8 features several wide bidirectional buses, with up to 3.7 TB/s of bandwidth between all components. This is much more than the accumulated memory channel bandwidth of 230 GB/s [184] [185] [34]. Note that the system bus, for any system design point, by definition can support the accumulated memory channel bandwidth. The system bus has two snoop buses and thus supports two coherence requests per clock tick from all attached components combined. Given the bus frequency of 2 GHz and the 128 B cache lines, this results in a snoop and invalidate throughput of 512 GB/s [185] [34].

4.5. MEMORY CONSISTENCY AND COHERENCE

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

4.5.1. NDP-M MEMORY INTERFACES AND CONSISTENCY

The NDP-M offers memory interfaces to let an NDP access the memory. The number of available interfaces and their width depends on the memory technology used. The NDP-M does not apply any bandwidth optimization techniques (grouping, etc.) to the

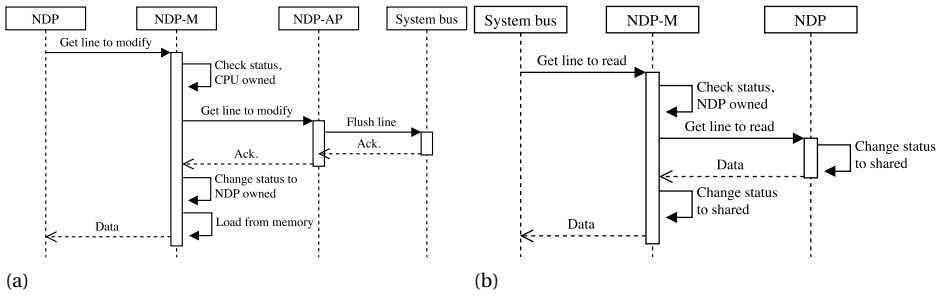


Figure 4.6: Two activity diagrams showing possible coherence interactions between the NDP and the CPU. In (a) the NDP wants to write to a line currently owned by the CPU, and in (b), the CPU wants to read a line currently owned by the NDP.

received memory request, as it assumes the NDP already makes use of the memory interfaces in an optimal way. Nonetheless, the NDP-M implements an unordered memory model, to be able to hide latency for address-translation misses or remote memory accesses. The *Rd/Wr queues* shown in Figure 4.4 allow for out-or-order issuing of memory operations, where the *Address Protection* unit also shown in Figure 4.4 ensures ordering rules.

The NDP can be considered as a thread in the same shared memory as the CPU, and therefore a memory-consistency model is necessary. The recent POWER multicore CPUs implement an unordered memory model [186], relying on (*lightweight*) *sync* instructions to enforce consistency between concurrent threads. After a *sync*, all memory accesses are committed to the global coherent space, and it is ensured that every thread will observe the right value. A *sync*, or likewise *barrier* instruction, executed by the NDP will have to ensure the same. In case the NDP has a hardware managed cache, this means all loads and stores have to be committed to the cache, and the NDP-M will ensure the CPU observes this value as described in the following sections. In case the NDP does not have hardware managed caches, a *barrier* has to travel to the *NDP Rd/Wr queue* shown in Figure 4.4 and ensure that this queue is empty before returning. Only after the queue is empty, all accesses have passed the *Coherence Manager* and have entered the global coherent space, again meaning that the CPU is able to see the latest value.

4.5.2. *Extended Coherence* BETWEEN CPU AND NDP

We introduce a hardware managed *Extended Coherence* solution to enforce coherence between the CPU, the NDP, and the main memory in a generic way. This is illustrated on a functional level in Figure 4.3b, showing the coherent views of both devices towards each other and the main memory. From the devices' (CPU and NDP) perspective towards the NDP-M, we implement a basic MSI (modified-shared-invalidate) protocol [187] . The devices can send messages like 'Get for read' (*GetS*), 'Get for write' (*GetM*), 'Upgrade for write' (*Invalidate*) etc. to the NDP-M. As MSI is a subset of the more complex polling-based coherence protocol found in contemporary CPUs, this *does not* require changes in the CPUs protocol. The *GetS* or *GetM* messages from the CPU travel down the 'normal'

path to the NDP-M, as these are just default loads from the CPU. We discuss *Invalidate* messages separately in Section 4.5.4.

At the NDP-M, the state of the memory is managed by the *Coherence Manager*, and every line has either the *CPU owned* state, the *NDP owned* state, or the *shared* state. The *shared* state is introduced to allow effective processing with shared data sets, such as for example reading data of one NDP by another NDP, without having to invalidate the data on the originating NDP. How the NDP-M manages these states is discussed in Section 4.5.3. We do not make a distinction between cached and non-cached (i.e. in main memory) lines: lines in an *owned* state will not leave that state until the other device takes action. This enables us to minimize coherence state changes, making it as transparent as possible. If we had introduced a *home* state (the line is only valid in the main memory) both the CPU as well as the NDP would be claiming and returning ownership all the time.

In Figure 4.6, we show two possible coherence interactions between the CPU and the NDP. Although not exhaustive, this figure shows the interaction between the various components in the architecture, and illustrates that the proposed mechanism can seamlessly integrate coherent NDP usage with existing CPU coherence. Figure 4.6a shows how the NDP can claim a cache line to modify it. The requests from the NDP enters the NDP-M, which checks the state of the memory and finds it as *CPU owned*, leading to a *GetM* message towards the NDP-AP. In Figure 4.6b we show how the CPU can get shared access to a cache line currently owned by the NDP. In both figures it is clearly visible how the NDP-M and NDP-AP together orchestrate coherence interactions between the system bus (the CPU) and the NDP. In these figures, the NDP-AP communicates with the system bus. This is the bus to which every component and device in the system is connected, e.g., all the CPU cores, the (semi-)shared last-level caches, the memory controllers, other CPUs, or even FPGAs using a coherent external link such as CAPI [51]. Therefore, the NDP-AP is a completely generic interface into the entire system, and the system bus *guarantees* that an, e.g., *Invalidate* message will invalidate the requested line in the entire system, and will be acknowledged back to the NDP-AP eventually.

4.5.3. IMPLEMENTATION OF THE NDP-M COHERENCE MANAGER

At the NDP-M, a combined fine- and coarse-grained distributed directory method is used to keep track of the coherence state of the memory. The *Coherence Manager* stores the state of the complete memory attached to the respective memory channel in an *Extended Coherence Directory* (ECD). When a coherence request is received, the directory is accessed to check and/or change the state of a line. To avoid having to access the memory for every state lookup, which would basically double the amount of memory accesses, an *Extended Coherence Directory Cache* (ECD-$ in Figure 4.4) is introduced, holding the most recently accessed directory items. Directory items are loaded by means of the dedicated Rd/Wr queue shown in Figure 4.4, which has priority over the other queues.

Given that our architecture targets large memories, the size of the directory, even if the three discussed states are stored in two bits, becomes quite large, while relatively occupying only 0.2% of the main memory. If workloads access memory with very little locality, the introduced cache will suffer from misses, which is especially not accept-

able for CPU traffic. Therefore, we introduce a technique similar to the one used to minimize snooping traffic between different processors in a SMP environment [34]. A tiny *Extended Coherence Directory Summary* (ECD-S in Figure 4.4) is introduced to store the coherence state for every (e.g.) 16 megabytes of memory, given that the entire 16 megabyte has the same coherence state. This requires a storage capacity in the kilobyte range. The NDPs are 'transparent' to the CPU and there is no penalty in memory bandwidth or latency. Only in cases where the state of a single cache-lines in the memory block varies and the ECD-S item is *Undefined*, the full directory is required through the ECD-$. The management of the ECD-S is based on existing technology as found in the POWER8 CPU for managing coherence snoop ranges between CPUs [34]. Summary items can either be changed in a transparent, hardware managed way, or by claims issued by an NDP-runtime or the code generated by a compiler. The software methods enable optimized coherence interactions in the system for well understood workloads, without losing correctness.

4.5.4. EXTENDED *Invalidate* AND *Exclusiveness*

As discussed in Section 4.2, we can not let every *Invalidate* message from the CPU travel to the NDP-M *Coherence Manager*, as it would hurt CPU performance. To solve this, the *ECD-S shadow* is introduced at the NDP-AP, as shown in Figure 4.5. When an ECD-S item is either *Shared* or *Undefined*, a CPU *Invalidate* will be forwarded by the NDP-AP to the NDP-M. In case an ECD-S item is *CPU owned*, no extra action is required, and a CPU *Invalidate* can be mirrored directly by the NDP-AP.

Care is necessary to handle the *Exclusive* coherence state at the CPU [187]. When a cache requests a cache line to read, and the memory controller is the only component acknowledging the snoop, the cache gets the line in *Exclusive* state, meaning it can upgrade the line for write access without having to issue an *Invalidate*. To avoid CPU caches upgrading cache lines without the NDP-M knowing, the NDP-AP, by means of the ECD-S shadow, has to acknowledge a snoop for a line currently in the *Shared* or *Undefined* state, letting the CPU know the NDP also has access to the line, and thus avoiding *Exclusive* access.

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

4.5.5. COHERENCE SCALABILITY

For the CPU it has been established that only the coherence actions strictly necessary are forwarded to the NDP-M. From the NDPs' perspective we consider two scalability concerns. First consideration is the amount of actions that need to travel from the NDP-M towards the NDP-AP. When the coherence state returned by the ECD lookup at the NDP-M corresponds with the state required by the NDP, the scope of the coherence request can stay local at the NDP-M, and no interaction between the NDP-M and the NDP-AP is required. This holds, when the application shows basic compute-data affinity, for the far majority of memory accesses, just as would be the case in a multi-core or multi-CPU environment. This is reflected in Section 4.9.2, where it is shown that coherence messages only take up 2% of the NDP-M - NDP-AP traffic, when considering a graph traversal workload. When the NDP wants to write to a cache line not already *NDP owned*, or

read a line currently *CPU owned*, an *Invalidate* message or *GetS* message has to be sent to the NDP-AP, respectively. These scenario's will occur when the NDP starts working, and when there is read-write interaction with the CPU for a data set. In Section 4.8 we will show that this start-up performance penalty is small, or can be completely hidden if there is enough memory access parallelism. Second consideration is whether the NDP-AP is a bottleneck in handling all NDP coherence messages. As is shown above, this question is only relevant when looking at the extreme cases. As discussed in Section 4.4.4, the system bus we consider in this work supports two coherence requests per clock cycle, and implementing a single NDP-AP component able to fully utilize this is trivial. These bus specifications means that, when the NDPs need to invalidate data at a rate higher than 512 GB/s, the system bus is a bottleneck. Note that this holds for *any* method of invalidating the data and is unrelated to the NDP-AP based mechanism: also letting the maximum amount of software threads invalidate data at their full capacity results in a 512 GB/s peak invalidation rate. This is confirmed by experiments on a 10-core POWER8 CPU, showing a peak software-driven invalidate throughput of around 420 GB/s. The invalidate throughput is therefore a bottleneck unrelated to our architectural proposal, as it would occur in any NDP or coprocessor proposal that takes coherence in to account. In case the number of snoop buses at the CPU increases, multiple NDP-APs could be used to make full use of them.The scalability of the coherence mechanisms to multiple CPU sockets is not a concern.

The scalability of the coherence mechanisms to multiple CPU sockets is not a concern. None of the described mechanisms are affected by, or require a notion of, being deployed in a multi-CPU environment.

4.5.6. NDPS WITHOUT HARDWARE MANAGED CACHES

In this work we focus on supporting arbitrary near-data processing, and not all NDPs need to have hardware managed caches. In fact, when considering for example workload-optimized NDPs implemented on reconfigurable fabric, it is far more likely the NDP uses an application specific local memory, and handles coherence manually. In this case, the NDP-M receives loads and stores from the NDP, instead of *GetS* etc. When receiving a load from the NDP, the NDP-M *coherence manager* still has to check whether the corresponding cache line has to come from the CPU or the main memory. When receiving a store from the NDP, the NDP-M *coherence manager* still has to invalidate the line at the CPU. As explained in Section 4.5.1, the NDP accesses now enter the global coherent space at the point of the NDP-M *coherence manager*.

4.6. VIRTUAL MEMORY MANAGEMENT AND ACCESSING REMOTE DATA

4.6.1. ADDRESS TRANSLATION IMPLEMENTATION AT THE NDP-M

In the proposed architecture, the NDPs work with virtual addresses, and all address translation mechanisms are implemented at the NDP-M. The NDP-M holds an ERAT content addressable memory (CAM), and a translation lookaside buffer (TLB) (see Figure 4.4). TLB hit rates have been identified as a problem for big-data workloads [188]. However, when considering a 2013 big-data optimized CPU core [180], using huge pages

in a transparent way [177], TLB reaches of 32 GB directly translatable memory are possible. We consider 32 GB or 64 GB a realistic size for NDP local memory. The NDPs are able to access remote memories as well, but this will be limited to, when the system is used correctly, few data sets, such as for instance *boundary values* when running a distributed grid-based solver, thus not increasing the physical address range significantly. Therefore, realizing a modern, big-data optimized, NDP-M based address translation mechanism is not a particular challenge, and is not further discussed in this work.

4.6.2. EXTENDED VIRTUAL MEMORY MANAGEMENT AND TLB SYNCHRONIZATION

In order to populate the TLB, and to keep the TLB synchronized with the rest of the system (supporting TLB shootdowns etc.), the NDP-M must be connected to the relevant fabric at the system bus. As described in Section 4.2, this cannot be done in a naive way, as it would impact the CPU performance in a very negative way. Therefore, as with coherence, the NDP-M works in close collaboration with the NDP-AP to enable TLB synchronization without overhead. In case of a (demand) miss in the NDP-M TLB, the miss is forwarded to the NDP-AP, which has its own page-walker (see Figure 4.5), to be able to serve the request without having to ask the OS for help. As described in Section 4.3, a page being accessed for the first time, will automatically be allocated in the local memory of the requesting NDP(-M). To limit the amount of TLB synchronization traffic that needs to be forwarded to the NDP-M, we use the *ECD-S Shadow* at the NDP-AP to filter the TLB synchronization commands. Only commands relevant for data ranges marked either *NDP owned*, *Shared*, or *Undefined*, are forwarded from the NDP-AP towards the NDP-M. These are the data ranges for which the CPU knows that, or cannot be sure whether, the NDP is working with them. TLB commands related to data ranges marked *CPU owned*, thus all non-NDP related memory (or NDP-related data owned by the CPU at that moment), are reflected directly by the NDP-AP, incurring no penalty.

4.6.3. ACCESSING REMOTE DATA

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

The NDP-AP can only access data with the full 128-byte cache line size granularity of the system, meaning that every access from the NDP will result in a 128-byte line stored in the NDP-AP's cache. Therefore, in case the NDPs have a smaller access granularity than the CPU (e.g. 32 B), consecutive accesses to the same line can directly be served by the NDP-AP, and do not have to go to a distant memory. The cacheability of remote data at the CPU also implies that NDPs can exploit mutual locality for various access patterns, like one-to-all. This is shown in Figure 4.7a, where the remote read from the

second NDP has a hit in the NDP-AP cache. This property will be a key feature when exploring the Graph500 benchmark in Section 4.9.

CACHEABILITY OF REMOTE DATA

As highlighted in Section 4.2.2, the snooping range of the CPU cannot be extended to the NDPs. Therefore, NDPs with hardware-managed caches holding lines of remote data are a problem, as the lines would be invisible for the system. The snooping traffic could be extended in an optimized way as done with the *Invalidation* messages described in Section 4.5.4, but that would still mean that a snoop has to travel to *every* NDP, potentially wasting bandwidth and energy. Another option can be to allow a little bit of cached remote data by introducing a shadow data cache directory at the CPU side [51]. This has as downsides that 1) it does not resolve the issue when considering large data sets, and 2) it implies that the NDP has to access remote data at the granularity of a CPU cache line, which is typically larger than the NDP access granularity, wasting valuable interconnect bandwidth.

Therefore, the proposed architecture does not allow NDPs to cache remote data, which is enforced by the NDP-M, marking a line *inhibited* when provided to the NDP. This is not as strict as the inhibited flag found in the POWER ISA [189], where it implies, for instance when writing to memory mapped device registers, that an access has to be performed by a special cache inhibited data path. In the proposed architecture, marking a line inhibited means that the line is evicted when the access is completed, or that a write through and eviction is performed. Note that, in case a remote access hits in the NDP-AP data cache, the latency is not much longer than accessing local memory. Thus, as long as high hit rates in the NDP-AP data cache are realized, remote accesses are not heavily penalized. To further alleviate the inconvenience of cache inhibited remote data, we introduce the concept of user/compiler enhanced coherence. By manually setting a data set as 'remote cache uninhibited', the NDP-M will allow the caching of this data in a remote cache. This comes at the expense of having to keep the system coherent manually, typically by flushing the NDP caches at certain points in the application.

In Figure 4.7 we shown the interactions involved with doing a coherent global load and a coherent global store. In case of a remote *GetS*, the NDP-M forwards it to the NDP-AP, which asks for the line on the system bus, puts it in its cache, and sends the correct sub-cacheline to the appropriate NDP-M. In case of a remote *GetM*, extra care is necessary. Because of the cache inhibited nature of remote data, an NDP cache cannot hold ownership of a remote cache line. Therefore, the NDP-AP locks the line on behalf of the NDP, and unlocks it as soon as the NDP returns it, also shown in Figure 4.7b.

As discussed in Section 4.5.6, NDPs do not have to have hardware managed caches, and for many designs this will likely not be the case. When an NDP does not have hardware managed caches, remote data accesses are treated exactly the same as local accesses, and the discussion above does not apply.

SCALING REMOTE ACCESSES AND INTER-NDP TRAFFIC

All inter-NDP traffic is sent over the CPU's memory channels, the system bus, and through the NDP-AP. In Section 4.4.4 it is established that the system bus is not a bottleneck. As discussed in Section 4.4.3, the number of system bus connections of the NDP-AP

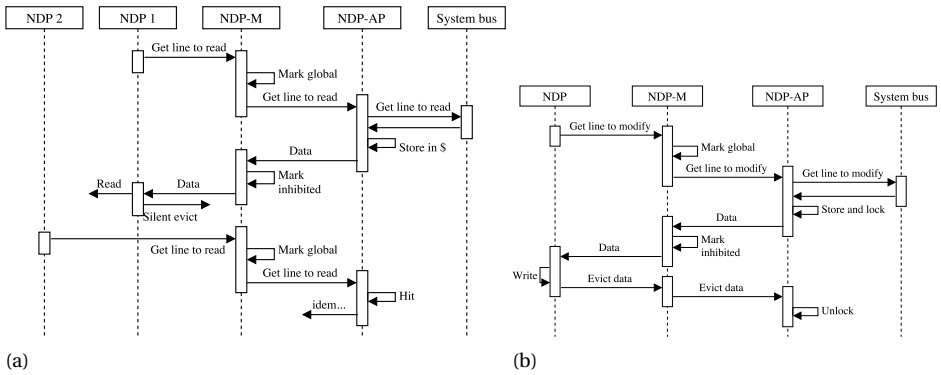


Figure 4.7: Two activity diagrams showing the process of a remote read (a) and a remote write (b).

can scale, as well as the number of NDP-APs. However, when considering the next-generation POWER CPU, a single system bus connection offers already more bandwidth than all memory channels combined [40]. This shows that a single NDP-AP implementation can already serve all inter-NDP traffic.

4.7. SYSTEM SIMULATOR AND IMPLEMENTATION

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

- The NDP-M and NDP-AP and their subsystems are implemented in software as shown in Figure 4.4 and Figure 4.5 and as described in this work.
- A variety of other components (memories, memory channels, system bus, caches, etc.) implemented in software.
- The system simulator tracks individual loads, stores, atomics, barriers, etc. through the entire system following the coherence and remote data access characteristics described in this work.
- The system simulator can track the data associated with every load and store, thereby 'mirroring' the memory of the host system and being able to check the correctness of the simulator and thus actually 'executing' the workload in the simulator.
- The system simulator supports general-purpose CPU and NDP cores running arbitrary applications, which get their loads, stores and operations from a frontend based on Intel PIN [190].

- The system simulator supports workload-optimized NDPs developed in a hardware description language (VHDL/Verilog), based on the Verilog Procedural Interface (VPI). This functionality is not used in this work.

The developed system simulator focuses on the discussed aspects of the proposed architecture, and is able to provide detailed performance information. The tracking of coherence states and even the data field for every load and store is different from existing performance estimators [191], which typically work at a coarser level.

Table 4.1 summarizes some of the most important system parameters. The characteristics are based the POWER8 CPU [185] [34]. We consider four DDR4-2600 memory controllers per memory channel, running at a realistic 60% utilization. Memory (controller) latency is also based on POWER8 characteristics. The best case round trip latency for an NDP memory access, based on the numbers provided in Table 4.1 and others like core-to-NDP-M interconnect, is around 60 ns, a 25% improvement over the CPUs access latency. In case of a coherence-directory cache miss or a remote access, this latency is obviously much larger, about double for the former, and about three to four times as high for the latter. The sizes of the coherency-directory cache at the NDP-M and size of the data-cache at the NDP-AP are set to an appropriate size for the two types of experiments performed. The NDP-AP bandwidth from and to the system-bus is almost high enough to saturate all the memory channels.

As NDP design point we use 64 slow, 1 GHz, in-order cores, with instruction latencies similar as those found in the processor described in [191]. The cache size of the general-purpose NDPs is only 256 bytes. This allows the exploitation of some data locality but makes sure that the observed performance characteristics come from the DRAM and inter-NDP bandwidths, and not from the cacheability of the data sets.

4.8. SYNTHETIC BENCHMARKS

In this section we discuss several synthetic benchmarks related to the bandwidths achievable in the proposed architecture and the parameters from Table 4.1. We explore both streaming and random access patterns for accessing local memory, as well as various communication patterns using multiple NDPs. When using random access patterns, 16 GB data sets per memory channel / NDP are used (128 GB total). Since these data set sizes are in the order of what one can expect in a real system, the ECD-$ and NDP-AP data cache sizes are set to values one would find in a modern CPU [34]: 64 KB and 512 KB, respectively.

4.8.1. LOCAL ACCESSES

First we explore the achievable bandwidths on a single NDP, while varying the coherence properties. Figure 4.8a shows the achieved bandwidth when doing a streaming read, while varying the initial coherence state of the data. The ECD-S is not used in this experiment. When the data is already in the *Shared* state, an 29 GB/s DRAM bandwidth is achieved. When the data is in the *CPU* state, coherence rights have to claimed from the NDP-AP, resulting in a slightly lower DRAM bandwidth of 24 GB/s, and a 2.25 GB/s up- and downstream bandwidth. For both cases, the bandwidth required to load *ECD* items from DRAM is negligible, as the accesses have a high degree of spatial locality.

Table 4.1: System specifications.

Simulated CPU	
Main-memory round-trip latency	80 ns
Access granularity	128 byte
Memory channels	8x, 20:10 GB/s Upstream:Downstream
Main memory (per channel)	
Technology and bandwidth	4 x DDR4-2400 @ 60% : 48 GB/s
Memory controller + DRAM latency	40 ns
Access granularity	32 byte
Target size	16+ GB
General-purpose NDP	
Cores	64 cores, 1 GHz, in-order
Data caches	256 byte private per core
Main-memory round-trip latency	60 ns (best case)
NDP-M	
ECD-$ size	variable, 32 B lines
ECD-$ / ECD-S / ERAT latency	4 / 2 / 2 ns
NDP-M - NDP-AP latency (single trip)	24 ns
Coherence message size	12 B
NDP-AP	
Data cache size	variable
System-bus data interface	128:64 GB/s In:Out
System bus	
Frequency	2 GHz
Snoop buses	2

Table 4.2: Common communication patterns and how they are implemented on the proposed architecture.

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

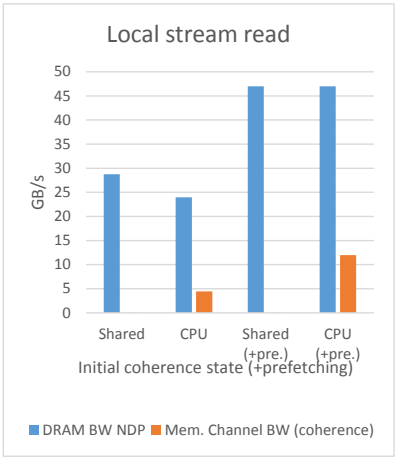
Also included in Figure 4.8a are the bandwidths when including software prefetching. This shows that the penalty of a coherence round trip to the NDP-AP is moderate, or can even be completely hidden.

Figure 4.8b shows the achieved bandwidth when doing random reads, while varying the portion of the data set with a valid ECD-S item, and the coherence state all data is *Shared*. When the portion with a valid summary item shrinks, more directory items have to be fetched from DRAM, and given the completely random nature of the accesses, this doubles the amount of memory accesses in the worst case, and halves performance.

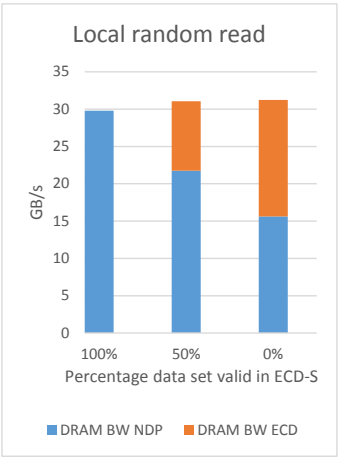
4.8.2. REMOTE ACCESSES USING VARIOUS COMMUNICATION PATTERNS

Table 4.2 shows the most common MPI communication patterns and how they are implemented in software on the proposed architecture. First we explore one to one communication, followed by many-NDP communication. In Figure 4.9a the performance for various one to one communication patterns is shown, using again streaming and random access patterns. A streaming read is bounded by the downstream bandwidth of the receiving NDP at 10 GB/s, with an NDP-AP data cache hit rate of 75%. Similarly, streaming writes are bounded by the downstream of the receiving NDP at 10 GB/s, with an NDP-AP data cache hit rate of 87%. This is higher than with streaming reads, because both the read retrieving the data from the NDP-AP as well as the write delivering again it are counted as hits. Random reads are bounded by the upstream bandwidth of the providing NDP at 20 GB/s, thus realizing a 5 GB/s bandwidth towards the receiving NDP (32 B out of 128 B). The NDP-AP data cache hit rate is almost zero as expected for such a large data set. Random writes are bounded by the downstream bandwidth of the receiving NDP at 10 GB/s, thus realizing a 2.5 GB/s bandwidth for the issuing NDP. The NDP-AP data cache hit rate is 50%, as every read misses but every write hits, as discussed in Section 4.6.3.

In Figure 4.9b the performance for various many-node communication patterns is shown, for which we use eight NDPs (root + seven). Broadcast results in an aggregate bandwidth of 70 GB/s, bounded by the downstream bandwidth of the receiving nodes. Since every node needs the same data, the NDP-AP data cache hit rate is very high,

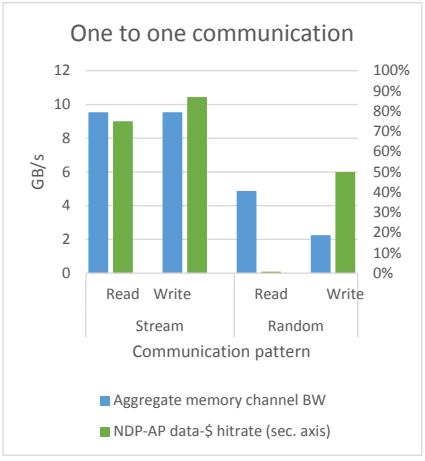


(a)

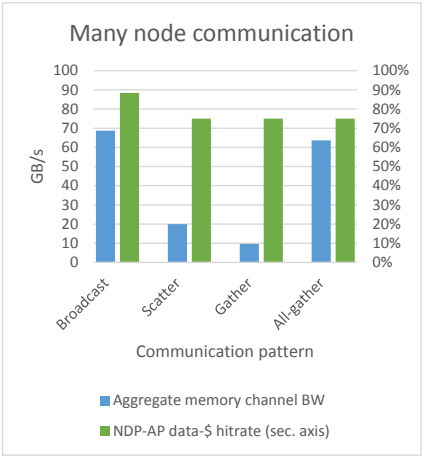


(b)

Figure 4.8: (a) Shows achieved DRAM bandwidth and memory channel bandwidth (coherence messages) for local streaming reads, while varying the initial coherence state, with and without software prefetching. (b) Shows achieved DRAM bandwidth for local random reads, while varying the percentage of data valid in the ECD-S.



(a)



(b)

Figure 4.9: Synthetic benchmark results, showing bandwidths and NDP-AP data cache hit rates for various inter-NDP communication patterns.

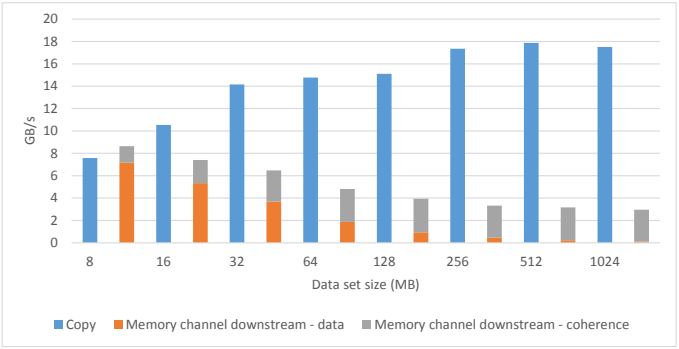


Figure 4.10: Using an NDP to copy a data set, of which 8 MB is stored in the caches of the CPU. The initial coherence state is *CPU owned*.

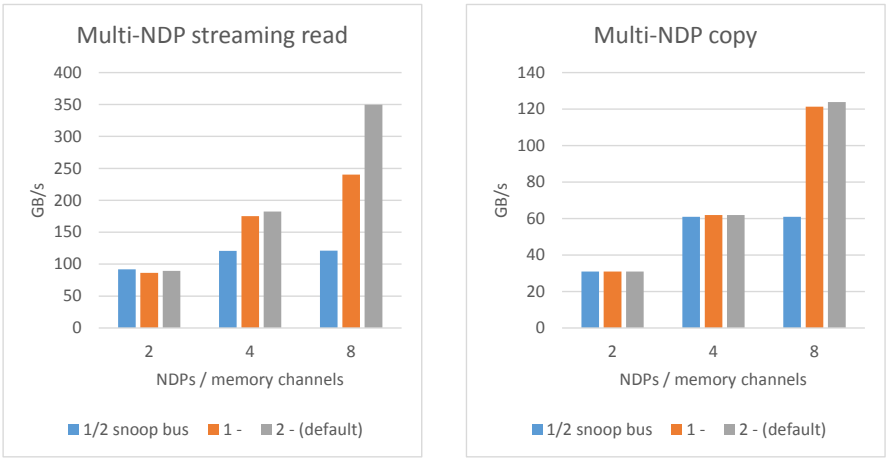
and the upstream bandwidth of the root is not a bottleneck. Scatter shows an aggregate bandwidth of 20 GB/s, bounded by the upstream bandwidth of the root, since now every receiving node needs different data. Gather is bounded by the downstream bandwidth of the root, and shows an aggregate bandwidth of 10 GB/s. All-gather, where every NDP reads from another NDP, shows a 64 GB/s aggregate bandwidth, bounded by the upstream bandwidth of the NDP-AP.

4.8.3. COPYING A DATA SET

In Figure 4.10 we show the performance for the essential operation of copying data, using a single NDP and a single memory channel. The source and destination addresses are both in the local memory of the NDP. When starting all data has the coherence state *CPU owned*, and 8 MB of the data set is stored in a (last level) cache of the CPU. In can be observed that for small data sets, the realized copy performance is very low. This can be explained by the fact that the 8 MB stored at the CPU needs to be written back to the NDP-M over the 'slow' 10 GB/s downstream link. For larger data sets this becomes a negligible portion, and the memory channel is foremost used for coherence interactions between the NDP-M and the NDP-AP. In the case of using a single NDP and the largest data set, the NDP-AP has to process one coherence messages roughly every four clock ticks. The copy performance of the CPU would be 10 GB/s, bounded by its downstream bandwidth. This shows that, although copying data seems a natural fit for NDPs, it only pays of for large data sets when one includes the, for natural applications inevitable, writeback and coherence overheads.

4.8.4. COHERENCE BOTTLENECKS

In Figure 4.11a we show the streaming read performance, while varying the number of snoop buses available at the CPU. The initial coherence state of the data is *CPU owned*. It can be observed that, when considering a limited number of snoop buses and many NDPs, the read performance of the NDPs is bounded by the *invalidate* capacity of the system bus. A single snoop bus can invalidate 256 GB/s, and thus the NDPs can not read data faster, irregardless of their local memory bandwidth. Figure 4.11b shows the same



(a) (b)

Figure 4.11: (a) Accumulated streaming read performance using multiple NDPs, while varying the amount of snoop buses at the CPUs' system bus. (b) Accumulated copy performance using multiple NDPs, while varying the amount of snoop buses at the CPUs' system bus.

experiment when considering the copying of a data set. Since copying data requires two reads and one write to local memory, but only two coherence claims, the system bus limitations are less profound. Both experiments show that for the default design point of having two snoop buses in the system bus [185] [34], no scalability problems become apparent, since the system bus can invalidate data at 512 GB/s while the NDPs have a peak accumulated local memory bandwidth of 384 GB/s. Note that, as mentioned in Section 4.5.5, this potential bottleneck is relevant for any NDP proposal, not just the one presented in this work.

4.8.5. SYNTHETIC BENCHMARKS INSIGHTS

The synthetic benchmarks offer two insights. First, the architectural proposal, implemented with the design point of Table 7.1, shows excellent performance for local memory accesses and various communication patterns. Second, the simple experiments represent an important range of operations discussed to be suitable for near-data processing [80]: copy, search, indexed lookups, very wide vector operations, etc. Our results show that a single of such an operation incurs a significant penalty in terms of coherence traffic and data writeback, and possibly performance. Meaning that, e.g. searching for a key in a data set, should be followed by many more key searches to offset the initial cost. For other workloads, like copying, this is much harder, as the data is likely made dirty by the CPU and possibly still (partly) in its caches.

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

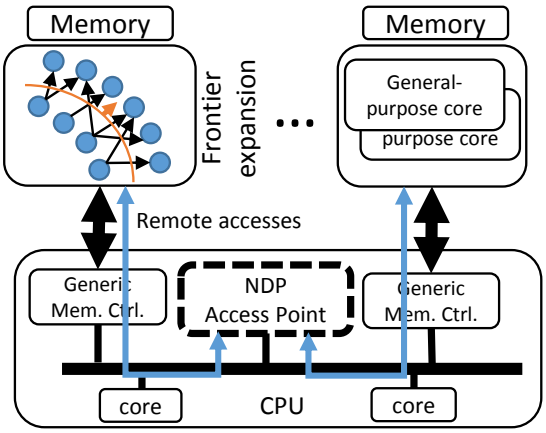


Figure 4.12: Graph500 mapped on our architecture. The frontier expansion is done at the NDPs, with non-local accesses being forwarded to the NDP-AP.

4.9. CASE STUDY: GRAPH500

Graph500 [58] is a well-known data-intensive benchmark and a typical near-data processing problem [192], doing several breadth-first searches (BFS) in a graph, representing workloads like big-data analytics. In several steps, called *levels*, it explores the entire graph, from a given starting node to all its neighbors, and in the next level from all its neighbors to their neighbors and so on. The set of nodes visited in the previous level, is called the *frontier* of the current level. The size of the problem is denoted as scale n, meaning the graph contains 2^n nodes. Key characteristics are unbalanced parallelism, short prefetch depths and random memory access patterns. It has been shown that running Graph500 on a cluster does not scale perfectly, and thus, to further increase performance for analyzing large graphs, increasing the node performance is important [193] [194]. An explanation of distributed Graph500 can be found in [193].

As we are, in contrast to the synthetic benchmarks, unable to run this complex workload with data sets in the 100+ GB range due to unfeasibly long runtimes, the ECD-$ and NDP-AP data cache sizes are set to smaller values: 8 KB and 64 KB, respectively. This way, we assume that our results for giga-scale simulations are relevant and can be scaled to tera- and exa-scale problems. A sensitivity study regarding the NDP-AP data case size is presented in Section 4.9.2. The used graphs are up to scale 26, or 64 million nodes (one billion edges), which consume 22 GB of memory when using the maximum of eight distributed processes, and are about an order of magnitude larger than the graphs used in [98] and [88].

4.9.1. IMPLEMENTATION

To evaluate Graph500, we use the system organization shown in Figure 4.12: every memory channel has an NDP each consisting of 64 general-purpose NDP-cores (see Table 4.1). We use the MPI-parallel *bfs_replicated* Graph500 reference code [58] to generate the distributed graph data sets. The MPI + OpenMP parallelization is replaced by a

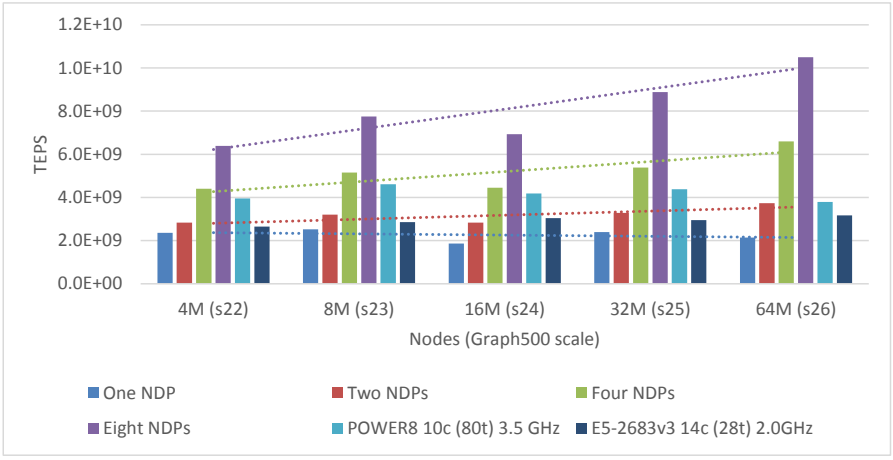
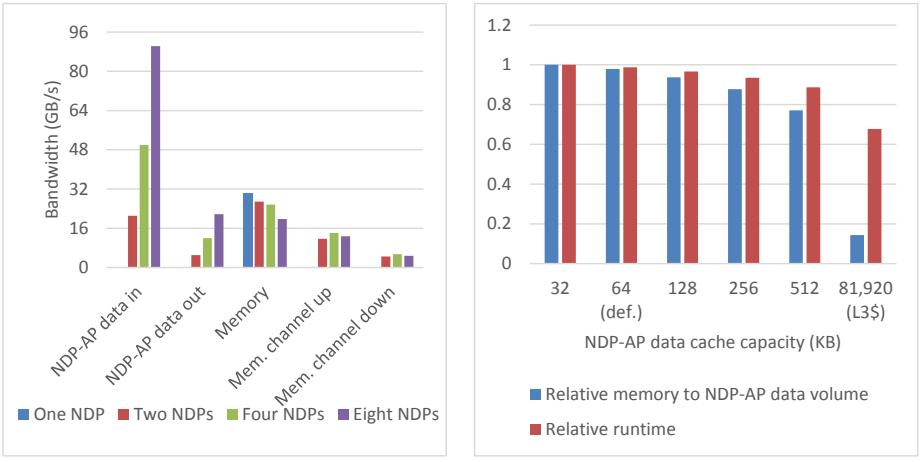


Figure 4.13: Performance for Graph500 running at our NDP architecture, with an IBM POWER8 and Intel Xeon CPU serving as reference.

nested OpenMP parallelization, where we first spawn a thread per NDP, followed by many threads for every NDP-core. All BFS related code is executed in the simulation environment. The BFS code is extended to use the 'direction optimization' to switch between bottom-up and top-down frontier expansions [195]. In the levels explored top-down (the first levels and the last levels), every node in the frontier visits all its neighbors, and sets their parent. The levels explored bottom-up (the middle levels), loop over all remaining nodes and try to find a valid parent in the current frontier. Various data accesses, for example a frontier check in the bottom-up phase, can be remote, meaning the access has to be forwarded to the NDP-AP to be issued there as described in Section 4.6.3. The specifications of the NDP-cores are shown in Table 4.1 and discussed in Section 4.7.

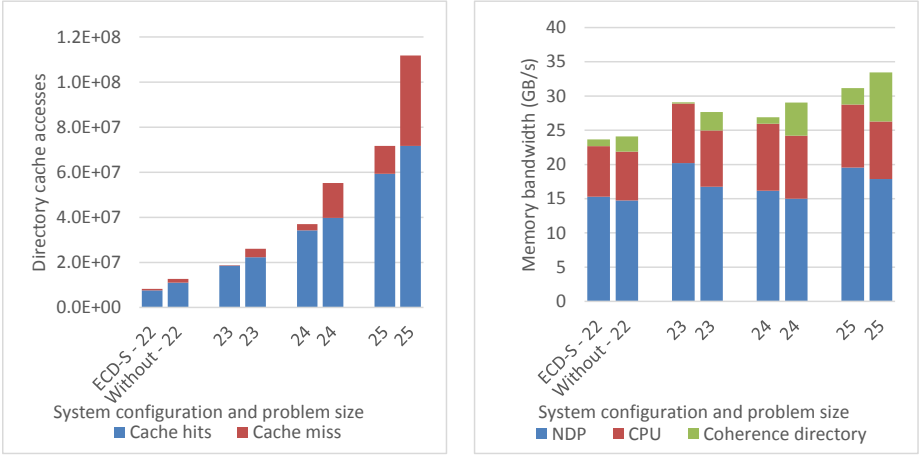
4.9.2. RESULTS AND DISCUSSION

The performance results of our Graph500 implementation are shown in Figure 4.13. As a reference we use the performance of the Graph500 reference code running on a real POWER8 CPU with 10 cores at 3.5 GHz, 230 GB/s memory bandwidth and 85 MB cache. For completeness, also results obtained on an Intel E5 2683v3 (14 cores, 28 threads, 2 GHz) server CPU are shown. When looking at a single NDP, we see around two tera TEPS (traversed edges per second) of realized performance. Running eight problems in parallel using eight NDPs would result in a very good node performance, but we are foremost interested in running large problems on multiple NDPs. The performance of multiple-NDP configurations steadily increases towards larger problems, and shows no sign of dropping for the sizes of the problems we explored. Clearly, a lot of work is needed to utilize all the available parallelism (512 threads in total), and to offset the low-thread-count regions at the start and the end of the execution, inherent to the application. Scaling to multiple NDPs is not perfect, since we now have to perform remote accesses which have a (in case of a NDP-AP cache miss) much higher latency than a local access. When us-



(a) (b)

Figure 4.14: (a) Various system bandwidths when running a scale 24 problem. (b) The impact of the NDP-AP data cache size on hitrate and performance running a scale 24 problem, using four NDPs.



(a) (b)

Figure 4.15: Performance of the coherence architecture as described in Section 4.5.3, with and without the use of the proposed *ECD-S* described in Section 4.5.3.

ing two NDPs, the percentage of remote accesses is 10%, growing to 20% for eight NDPs. Due to this penalty, we see a strong performance scaling of a factor 1.5x and 1.7x when doubling the NDPs. Weak performance scaling is better, showing factors between 1.6x and 1.9x.

We expect the performance to increase further for larger problems, and not to drop due to caching effects, based on the trend lines shown in Figure 4.13 and the following analysis. For a scale 26 problem, the *visited* and *out_queue* bitmaps, essential data sets keeping track of the state of every vertex with a single bit and both accessed in a 'random' way, are eight MB in size, or 512 times larger than the combined cache size of 64 NDP-cores (64 times larger than all caches in eight NDPs). The *predecessor* list, also a critical data structure keeping track of every vertex' predecessor and accessed in a 'random' way, is a factor 64 larger than the bitmaps. This implies that our results do not depend on the cacheability of essential data sets but on the access characteristics of our architecture, foremost the low latency and 32 B access granularity.

In Figure 4.14a we show the measured bandwidth for various aspects of the architecture. The bandwidth into the NDP-AP (the responses to reads issued by the NDP-AP) grows with the amount of NDPs, but does not reach its limit yet. The bandwidth going out of the NDP-AP (the responses towards the various NDP-M's) is a factor four lower, because the reads done by the NDP-AP are at the CPU's 128 B granularity, while the responses towards the NDP-M are at a 32 B granularity. The main memory bandwidth slowly decreases when using more NDPs, driven by the fact that the performance achieved per NDP is lower due to a smaller problem size per NDP and a larger portion of remote accesses. The memory channel bandwidths are stable between system configurations, balancing out the relative longer runtime and the larger amount of remote accesses. The memory channels show the same 4:1 bandwidth ratio for upstream and downstream as the NDP-AP.

NDP-AP DATA CACHE PERFORMANCE

Accesses to remote data are resolved by means of the NDP-AP, as described in Section 4.6.3. The NDP-AP holds a small cache to exploit some locality. When a remote access hits in this cache, the latency is dramatically reduced. Therefore, a large cache is likely to improve performance. Furthermore, a cache hit eliminates the need for data to be accessed in a DRAM and transported to the NDP-AP. In Figure 4.14b we show the reduction in runtime, as well as the reduction in data transports achieved when we increase the NDP-AP data cache size, when running a scale 24 problem with four NDPs. For the default cache size of 32 KB, we have a hit percentage of less than 2%, due to the limited capacity as well as the fact that for Graph500 the remote accesses have little spatial locality. When increasing the cache capacity to 512 KB, the hit rate is 26%, increasing the application performance by 12% and reducing the amount of data traveling from the various DRAMs to the NDP-AP by 23%. As example, in case the NDP-AP data cache would be the size of the entire L3 cache of the CPU (80 MB), the hit rate increases to 97%, and the application performance by 33%, clearly indicating that the latency for remote accesses is key for the performance in the architecture.

IMPACT OF THE COHERENCE METHODS

An important aspect of the proposed architecture is the handling of coherence between the NDPs and the CPU, as discussed in Section 4.5. The NDP-M checks the coherence state of every access from the NDP or the CPU in the ECD via the ECD-$. To limit the ECD-$ accesses, the ECD-S is used as discussed in Section 4.5.3. When starting the benchmark, we set the entire graph structure, which is read-only after initialization, to *Shared* in the ECD-S, meaning accesses to this data set do not have to access the ECD-$.

In Figure 4.15a we show the performance of the ECD-$ for growing problem sizes. For larger problems, the amount of ECD-$ accesses obviously increases. When using the ECD-S as described above, the amount of ECD-$ misses is very limited, with a maximum of 17% for the largest problem. This means almost all accesses are made coherent in either the ECD-S latency, or the ECD-$ latency as described in Table 4.1. When not using the ECD-S, also shown in Figure 4.15a, the amount of ECD-$ accesses increases by 35%, and the ECD-$ misses accordingly.

In case of an ECD-$ miss, the item has to be loaded from DRAM. In Figure 4.15b we show a breakdown of the DRAM bandwidth with respect to the component doing the access. When using the ECD-S, a very small portion of the bandwidth is used for loading and storing directory items. Without the ECD-S, this percentage increases to over 20% of the bandwidth, introducing a significant overhead in time and energy.

In Figure 4.14a we show the bandwidths for the memory channels. Roughly 2% of this bandwidth is spent on coherence traffic between the NDP-M and the NDP-AP. When using only one NDP, we observe around 0.01 GB/s of coherence traffic between the NDP-M and NDP-AP in order to claim all data previously owned by the CPU.

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

4.9.3. POWER ANALYSIS

We used McPat 1.0 [139] to evaluate the power consumption of the general-purpose NDP. When using low standby power (LSTP) 22-nm technology, the power of the entire 64 core NDP is 5.9 W. The power usage for the NDP-AP is estimated at 4 W, based on comparable bus-attached components discussed in [196]. The NDP-M works at a much lower frequency and we estimate a power of 2 W, giving the entire memory side (memory-side chip + DRAMs) a 20% increase in power [197]. This results in a power budget of the entire proposal of 67 W. Given that the CPU is practically idle when running workloads on the NDPs, the CPU cores go to sleep mode, for a power saving of 100 W [197] [198]. This delivers a net power saving of 33 W for our proposal. Given the much shorter runtime, a significant energy-to-solution saving will be realized, using about half of the energy compared to the reference, depending on the exact system configuration and problem.

4.10. CONCLUSION

In this work, we introduced an architecture for a seamless integration of arbitrary near-data processing capabilities in an existing server environment. By introducing a com-

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

5

SIMULATION ENVIRONMENT

*Focusing on workload-optimized systems, implies optimizing **workloads**, running on **systems**. Benefits, for any metric of choice, should be searched for and cared for on a workload level, and not just for specific kernels or operations. Furthermore, these benefits should become apparent when looking at entire systems, and not only at the level of individual devices or components.*

The proposed near-data processing architecture does not focus on a particular type of near-data processor, but assumes the deployment of arbitrary near-data processing capabilities. This includes the use of workload-optimized NDPs, targeting a reconfigurable fabric. To evaluate workloads making use of such NDPs, a system-level simulation environment, able to include NDPs developed in a hardware-description language, is necessary. No such a simulation environment is (publicly) available, hence, effort was needed to develop one. Since developing the entire system in hardware is infeasible, unnecessary, and would result in unrealistically long simulation times, only a strict subset of the system must be simulated in hardware, connected to the rest of the system simulated in software. Besides workload-optimized NDPs, general-purpose NDPs, running software threads, are of key importance as well. Many applications are very complex or have code constantly under development, making the use of software based solutions the only realistic option.

The simulator, including both types of NDPs, simulates all aspects of the proposed NDP architecture, including the CPU. This way we can run and evaluate complete applications on the entire system. In this chapter we discuss the development efforts for the realization of the simulator, and discuss the lessons learned.

5.1. INTRODUCTION

In this chapter we discuss the development of a system simulator. This effort started with the development of several hardware and software components needed for a future near-data processing prototype system. To aid development and to complement the research opportunities viable on the prototype platform, a system simulator was deemed necessary. Other novel hardware platforms, such as the IBM CAPI interface [51] and the systems from Convey Computer [119], also offer a system simulator to make testing and debugging much easier. As the realization of the prototype platform was delayed over and over, focus shifted towards the creation of a capable simulator environment, to be able to create valuable results without having the special hardware in place.

Two branches of the simulator were develop. The first one uses the hardware and software developed for the prototype platform, and is capable of simulating NDPs developed in a hardware description language such as VHDL or Verilog. It simulates the CPU and memory system around these components. With this, we are able to simulate workload-optimized NDPs, and use them as they would be used on a real system. Several applications and NDPs have been tested with the simulator, and valuable lessons are learned. As the hardware core of this simulator limits both the simulation speed as well as the complexity of the features we can implement in reasonable time, another branch of the simulator was developed. This branch supports only general-purpose NDPs running software, but implements the complex architectural features described in Chapter 4. To make an explicit distinction between the two different simulation environments in this chapter, the simulator able to run hardware NDPs is called the *system-simulator*, since it truly mimics the memory system of a CPU. The software-only simulator focusing on evaluating the architectural features at high speed, is called the *performance-simulator*. Both simulators share a large code base, and are implemented in the same library.

The development of the simulation environment has been key to most of the results described in this work. Although it started as an efforts to support the prototype platform, it grew into a very complex and interesting environment able to test a wide variety of aspects. It is a completely homegrown project, as the goals and features needing support differed too much from existing simulators. This has taken a significant amount of time, but many of the obtained insights would not have been possible otherwise.

5.2. COMPONENT DEVELOPMENT

The development of the system-simulator environment started as an effort to support the development of a prototype platform, and many of the features in the system-simulator find their origin there. Therefore, we start with a description of the prototype platform and the hardware and software developed for it.

5.2.1. PROTOTYPE DESCRIPTION

To research the use of novel memory technologies, IBM developed the ConTutto card [199]. ConTutto makes use of the flexibility offered by the POWER8 memory system [34], by replacing one of the Centaur ASICs with an FPGA. This is illustrated in Figure 5.1, replacing the top-right Centaur by ConTutto. By having the Centaur logic implemented

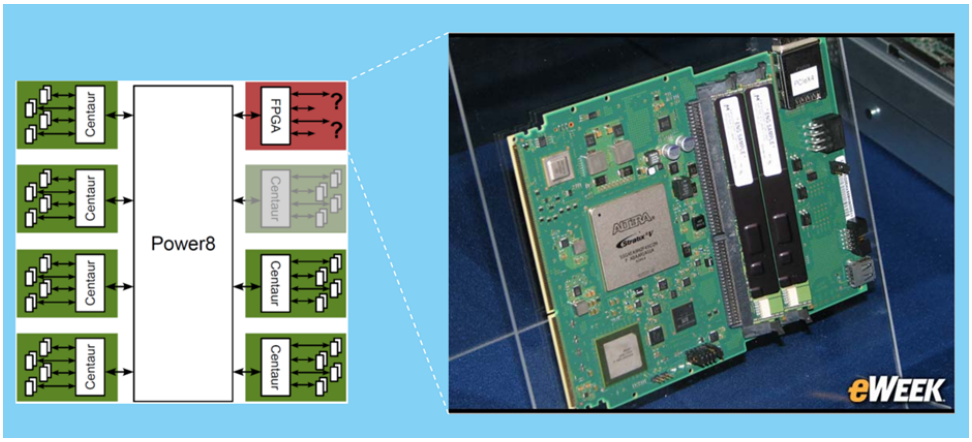


Figure 5.1: An illustration of the 'ConTutto' prototype platform. One of the *Centaur* ASICs in a POWER8 is replaced by an FPGA.

on an FPGA, the use of different types of technology specific memory controllers became possible. A working system using non-volatile MRAM memory based on the ConTutto platform was recently presented [200].

By having an FPGA in the memory system of a CPU, ConTutto also allows for the deployment of near-data processing capabilities. Conceptually, this follows the architecture presented in Chapter 4. There are however important differences as well. As we are not able to change the CPU, the NDP-AP (Access Point) cannot be implemented. Furthermore, the protocol over the memory channel cannot be generalized to also carry messages next to the standard loads and stores. This implies that there is no 'clean' way of communicating with ConTutto, that we cannot have hardware managed coherence and virtual memory support, and that we cannot have inter-NDP communication.

We did not get our hands on a ConTutto card within reasonable time, and thus are not able to show results based on it. ConTutto, with its features as well as shortcomings, is however still the line through Section 5.2 and Section 5.3.

5.2.2. HARDWARE DEVELOPMENT EFFORTS

Given the reconfigurable nature of ConTutto, it is possible to implement the NDP-M (Manager) in hardware. Due to the limitations of the prototype platform and the limitations in time, only a subset of the architectural features described in Chapter 4 are implemented. In Figure 5.2 the basic blocks are shown, and how they connect to the provided ConTutto hardware design. The interfaces towards the provided ConTutto design are based upon the Altera Avalon interface [201], offering an easy to use and flexible data transportation mechanism. Implemented in hardware are:

- NDP-M command snooper. This component scans all downstream traffic for commands targeting the NDP-M.
- Control unit. The control unit interprets the messages it receives from the CPU,

manages the ERAT, the state (idle, running, etc.) of the NDP-M, and the communication with the NDP.

- Effective to Real Address Translation (ERAT). This enables the use of virtual pointers by the NDP.
- NDP-interfaces. These interfaces allow the attachment of an NDP to the NDP-M.
- Arbiter. The arbiter inserts the NDP memory traffic into the traffic stream of the CPU. The CPU always has priority.

The command-snooper implements a simple method of communicating with the NDP from software. The 128 B write requests from the CPU are scanned for starting with a special 32 B 'magic' key, indicating the cache line is intended for the NDP-M. This mechanism allows us to send messages to the NDP-M from user space. A memory-mapped method, although arguably cleaner by design, would require a device driver to accomplish the same. By using this 'key method', we mimic the proposed method from Chapter 4, where we describe the use of a special (e.g. *icswx* [189]) instruction to communicate with the NDP-AP.

The control component foremost takes care of populating the ERAT data structures and the communication between software (NDP-library and applications), and the NDP.

The ERAT is implemented using a parameterizable Content Addressable Memory (CAM) data structure. It translates 48 bit virtual addresses to 40 bit physical addresses, using a 20 bit (1 MB) offset. The creation of these 1 MB large translation blocks will be explained in Section 5.2.3. To populate the CAM, the NDP-library (discussed later) sends a command message containing the physical address of the translation tables, and their size. The control unit starts loading the tables from memory. By doing this, it bypasses the ERAT, as the addresses are already physical. The received data is used to populate the CAM.

The NDP-interface provides the following:

- The communication interface realizes communication between applications and the NDP.
- The memory interface offers two 32 B wide channels, both able to do a read or a write every clock cycle.

Two methods for communicating are implemented. First, applications can use special purpose registers to share values (pointers, constants, etc.) with the NDP, inspired on the Delft Workbench [202]. The second method of communicating is message based, where both the application and the NDP can push 64 B messages to each other. With the design of the NDP-M and its interfaces, we support arbitrary NDP designs to be deployed in the memory system. A clear distinction is made between the part intended as framework provided to the user (NDP-M), and which part should be added by the user (NDP).

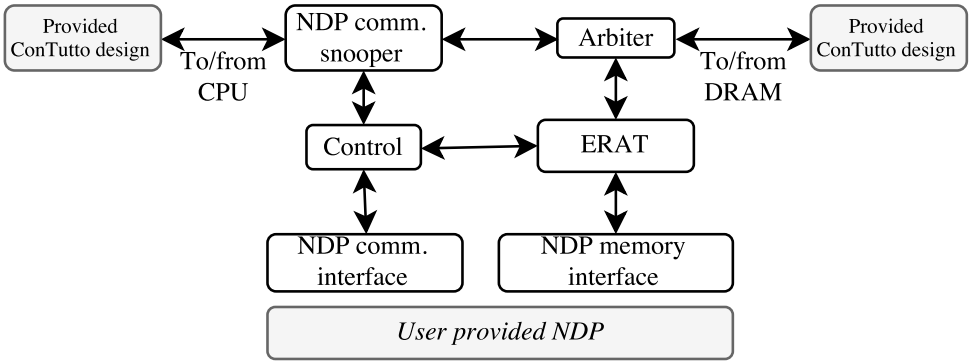


Figure 5.2: The main components of the NDP-M, implemented in VHDL. IO is defined by the provided ConTutto interfaces based on Altera Avalon.

5.2.3. NDP-LIBRARY SOFTWARE DEVELOPMENT

To be able to manage the NDP-M and the NDPs, an NDP software-library was developed. Parts of the features provided are generic, and would also be needed on a future 'real' NDP platform, while others are specific for the prototype platform. The NDP-library contains the following three main classes:

- The *ConTuttoManager* class represents a single ConTutto in a single memory channel. It offers functionality to communicate with the NDP(-M), allocate data structures, setup the address translation tables, and synchronize the CPU and NDP.
- The *ConTuttoAllocation* class represents a data set allocated to be used by an NDP. It can only be created by means of a call to the *ConTuttoManager*. It offers functionality to manually manage coherence, and keeps track of its virtual and physical address ranges. All basic array operators like *[ ]* are overloaded such that a ConTuttoAllocation can be used like any other data set.
- The *POWERConTuttoInterface* class implements the architecture specific layer for the coherence (data-cache block flush) and synchronization (data-cache synchronize) operations. Furthermore it allows to connect to the system-simulator, described later.

ALLOCATING DATA SETS AND VIRTUAL MEMORY MANAGEMENT

Allocation a ConTutto data set can be done by means of a single call to the ConTuttoManager:

```
template <class T>
ConTuttoManager::ConTuttoAllocation<T> allocateData(size_t items)
```

This templated method returns a *ConTuttoAllocation* of type *T*. To aid the virtual memory support of the NDP-M, special care is taken when allocating. To limit the size of the address translation tables, we make use of how the Linux buddy allocator allocates memory [178]. When allocating a large data set, the number of returned physical

contiguous pages typically looks something like this: 1-1-2-2-1-1024-1024-1024-2-1-1. The majority of the data is placed in large physically contiguous blocks of up to 1024 pages (the *MAX_ORDER* compile parameter for the Linux kernel). These large contiguous blocks of pages can be translated with a single virtual-to-real pair. Therefore we want our ConTutto data sets to only consist of these blocks, limiting the storage requirements of the ERAT CAM in the NDP-M. When calling *allocateData*, we recursively over-allocate, until the requested amount of storage can be served by only the returned *MAX_ORDER* sized blocks. An important aspect in this workflow is that every allocated page must be touched, since only then is the virtual space getting backed by physical space, and the physical pages really allocated. These large contiguous blocks are added to a *ConTuttoAllocation* object, to be returned to the application. The *ConTuttoAllocation* also keeps track of the original allocation pointer, in case we want to *release* the object and its storage.

Obtaining the real addresses of a virtual page is not trivial. To realize this, we scan the information available in */proc/self/pagemap/* to get the physical *page frame number* (PFN) of a virtual address [203]. The *pagemap* is a user space accessible (although depending on the kernel version) map of the page tables, storing 64 bits of information per page, and can be indexed by the virtual page number. The bits 0 until 54 represent the PFN. The complete physical address is then constructed as:

$$Real = (GetPFN(virtual) * page\_size) \,|\, (virtual \,\&\, (page\_size - 1))$$

With the ability of retrieving the physical addresses of a data set, we can construct the address translation tables. Adding a data set to the address translation tables can be done by calling

```
template <class T>
void ConTuttoManager::addToAddrTranslation(ConTuttoAllocation<T>& dataset)
```

This method stores all the virtual-real pairs contained in the *ConTuttoAllocation* in a special, physically contiguous, memory region representing the translation tables. A *ConTuttoAllocation* can be removed from the translation tables in a similar fashion. A call to the method

```
void ConTuttoManager::sendAddrTranslationTables()
```

flushes the address translation tables from the caches, and sends a command to the NDP-M. The NDP-M in turn starts reading the tables from memory to populate the ERAT CAM.

MANUAL COHERENCE AND SYNCHRONIZATION

To enforce coherence between CPU and NDP, a manual form of coherence is implemented. First, *ConTuttoAllocation* objects offer functionality to flush certain items, or entire data ranges, from the caches. The most generic method is:

```
void ConTuttoAllocation::flushRangeToRAM(size_t start, size_t items)
```

This method breaks up the intended data range in cache line sized blocks, and calls a *POWERConTuttoInterface* object to writeback and invalidate the cache line with an *dcbf* instruction. This instruction takes a virtual address as a parameter, and can thus be used in an easy and straightforward way. Second, a *ConTuttoAllocation* offers the method

```
void ConTuttoAllocation::readFromRAM(size_t index)
```

which forces a following read to the indicated item to come from main memory. This is implemented exactly the same as the *flushRangeToRAM* method (calling *dcbf*), but for this method the assumption is that the data is *clean* (or not existing) in the cache. In case the line is available but *clean*, the instruction only invalidates the line, and does not write it back to memory. A consecutive read to the address, will result in a read from memory. The two methods to manage coherence manually thus only differ in syntax, easing the programming effort. The first one would typically be used to flush a data set to the memory, while the second would be used to read results of poll for a status.

The *ConTuttoManager* object offers functionality to synchronize the memory system by calling

```
void ConTuttoManager:syncMemSystem()
```


This method calls an *POWERConTuttoInterface* object which implements this functionality with a *dcs* instruction. This instruction flushes all load/store queues and ensures all pending operations are committed. The method however solves only a very small part of the problems involved with synchronization between CPU and NDP. The described *dcs* instruction makes sure all writes are committed to the caches, and all cache flushes are committed to the memory controllers on the CPU. However, once cache lines are committed to the memory controllers, they are beyond our control. This means that cache lines can possibly overtake each other. Cache lines containing an NDP-message to start working on a piece of data, can overtake the data the work should be applied on. In general, there is no way in telling when a cache line is actually committed to main memory, and the NDP can thus start working. A hardware-managed coherence mechanism between CPU and NDP, as discussed for the proposed architecture in Chapter 4, will solve this issue, since the coherence mechanisms will make sure an NDP load to a cache line still 'traveling' to main memory, will be halted, as the NDP does not have the coherence rights to do the load.

COMMUNICATING WITH THE NDP

As indicated in Section 5.2.2, the NDP-M contains a unit which snoops all passing store commands and scans them for a specific key in the first 32 B of the 128 B cache line. When a message is detected, it is forwarded to the control unit, where it is interpreted. In Figure 5.3 we show the layout of a received message. The *Command* field indicates the type, and possibly up to three parameters are provided. An example command is 'load address translation tables from address X'. Other typical commands are starting, stopping, and halting the NDP by the NDP-M. A special type of message is an NDP message: a message from the user-level part of the application targeting the NDP. In this case the control unit forwards the 64 B payload to the NDP.

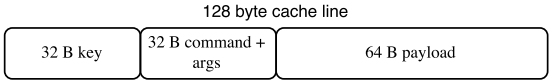


Figure 5.3: Layout of a message intended for the NDP-M, carried on a 128 B cache line.

5.3. SYSTEM SIMULATOR

The system-simulator builds a CPU and memory system around the proposed hardware and software components. This allows us to test the proposed components, but foremost to test complete applications running on the simulated CPU, making use of the simulated NDPs, to be able to test their performance and behavior. Furthermore, it allows for a flexible evaluation of various system parameters, like bandwidths and frequencies.

5.3.1. FOCUS AND CONCEPT

Since near-data processing is all about data, we decided to focus the system-simulator on the role of data in a future computer system. Key elements are, e.g., the flushing of caches, the synchronization, and of course the performance attributes like bandwidths and latencies. The system-simulator can therefore best be described as 'memory-system simulator with NDP capabilities', as visualized in Figure 5.4. The caches of the CPU are therefore included in the system-simulator, but the CPU cores are not, since we are not particularly interested in their exact behavior. As the system-simulator targets the prototype platform, we stay true to the features offered by this platform, and do not add 'tricks' to the system-simulator to ease development. For example, communication between software running on the CPU cores and the NDPs goes through the simulated memory system, and not through a 'special wire' only existing in the simulation environment but not in real life.

Most interesting, and most differentiating feature compared to other work, is the fact that the system-simulator actually executes the workload. This means that the caches in the simulated CPU, the simulated main memory, and all the buffers etc., hold the actual values the host application is using, and is a 'mirror' of the memory of the host system. This implies that a correct implementation is essential for obtaining the correct application results as well as the right performance measurements. In fact, the host application could crash if the system-simulator would return invalid data when considering for example array indices or pointers. With this we ensure that the system-simulator can be used to test various components targeting the prototype platform.

In Figure 5.5 we show the high-level picture from an application binary perspective. On the right we see the application under test, compiled as a stand-alone binary, linking with the discussed NDP library. On the left we see the system-simulator binary, which is compiled with a certain NDP hardware design included. The two applications interact with each other by means of a communication channel.

5.3.2. FEEDING THE SIMULATED MEMORY-SYSTEM

The CPU cores themselves do not exist in the system-simulator. The system-simulator can be considered to start at the load/store units in a CPU core, feeding the simulated

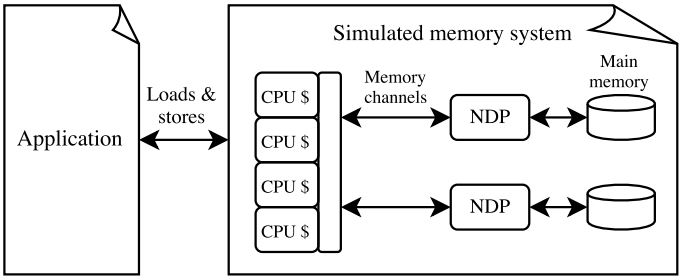


Figure 5.4: The concept of the system-simulator is to simulate an NDP enabled memory system.

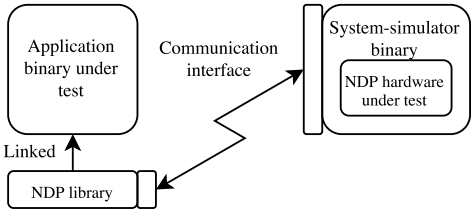


Figure 5.5: High level view of the system-simulator, showing both the application under test, the simulator, and the communication channel.

memory system with data and retrieving data from it. In Figure 5.6 we see how feeding the caches on a real platform relates to the simulation environment.

On the left side we show the prototype system, where multi-threaded CPU code issues load and stores to the memory system of the real CPU. NDP commands are issued to the NDP library, which in turn translates this into a series of loads and stores, also entering the memory system. Platform specific operations targeting the memory system are issued to the *POWERConTuttoInterface* which translates them into the correct *flush* and *synchronization* commands. On the right side of Figure 5.6, we show the version of this process when using the system-simulator. NDP commands behave in the same way, and for these actions applications are not aware of whether they are using the system-simulator or not. Accessing data sets can however not be done in a completely transparent way. An effort was undertaken to realize this with the overloaded *[ ]* operator of a *ConTuttoAllocation* object, and make the distinction between using the system-simulator or not within the *ConTuttoAllocation* object. This is unfortunately not possible, as the overloaded *[ ]* operator does not allow us to make a distinction between a load or a store. Therefore, accesses to a data set targeting the system-simulator are done by explicit calls to an *ConTuttoAllocation* object:

```
T ConTuttoAllocation::readSim(size_t index, size_t threadId = 0)
void ConTuttoAllocation::writeSim(size_t index, T value, size_t threadId = 0)
```

An example of using these calls in a typical OpenMP parallelized loop looks like this:

```
#pragma omp parallel for
```

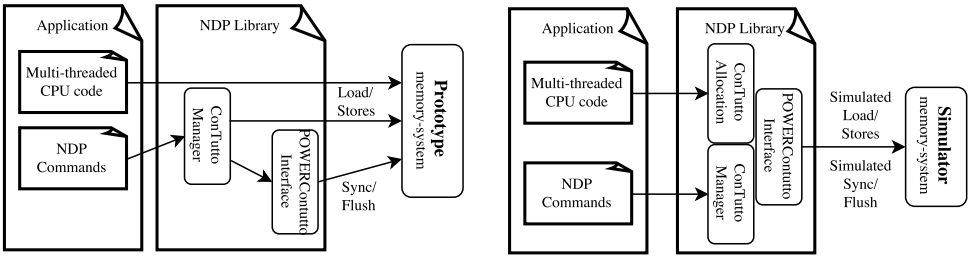


Figure 5.6: Two figures showing how loads, stores, and special commands enter either a real memory system (on the left), or a simulated memory system (on the right).

```
for (size_t i=0; i<maxIter; i++)
{
  data.writeSim(i, 0, omp_get_thread_num()); \\ set data set to zero
  assert(data.readSim(i, omp_get_thread_num()) == 0);
}
```

5

These methods generate a load or store request for the system-simulator, containing all important information like address, size, originating thread etc. The addresses are translated to real addresses based on the information stored in the associated *ConTuttoAllocation* object. This is necessary as the memory system of a CPU works with real addresses. When ready, the load or store requests are, by means of the *POWERConTuttoInterface*, issued to the system-simulator.

The loads and stores issued to the CPU caches are processed at a specified speed, e.g. one operation per clock tick. To be able to mimic the performance of real applications running in the system-simulator, without simulating the actual CPU cores, extra actions are required. To stall the simulated CPU caches for a period of time, thereby simulating computation time, the concept of issuing a 'bubble' to the system-simulator is introduced. A simulated CPU cache receiving a *bubble* simply waits for a cycle before continuing with the next operation. With this mechanism we can easily tweak the performance of the simulated CPU to the performance we measure on a real CPU, as is done for e.g. the sorting work described in Chapter 7.

COHERENCE, CACHE MISSES AND EVICTS

The simulated CPU caches implement a basic MSI protocol to be able to coherently share data between each other. An access hitting in a remote cache is solved by means of this protocol, and incurs a fixed time-penalty.

Cache misses and evicts are issued to the correct memory channel, based on the *ConTuttoId* of the underlying *ConTuttoAllocation* associated with the memory access. This *ConTuttoId* is defined by the *ConTuttoManager* object that did the allocation. Since a *ConTuttoManager* represent an ConTutto in a specific memory channel, this is an easy way to distribute data sets in the system. This is the system-simulator version of the NUMA based mechanisms for the proposed architecture in Chapter 4. Data sets can also be scattered over the various memory channel on a cache line granularity, as is the case in a real CPU. This is done when we do not use the NDPs, or have datasets that are

not NDP related. Data returning from the memory channels is injected into the right cache(s) which did the request. Capacity based cache evicts are done with a true least-recently-used policy.

PREFETCHING, LOCKS, BARRIERS AND SYNCHRONIZATION

To be able to get representative performance and correct results, several complex features are implemented as well. Prefetch loads are supported by the CPU to boost its performance and make the performance more realistic. These prefetch loads differ from default loads only in that they are non-blocking from the application point of view, and thus, from the system-simulator point of view, do not return a response to the application. In case of a cache miss, the data is loaded from memory and stored in the cache, and nothing more happens. Depending on whether the simulated CPU is intended to have hardware prefetchers or not, prefetch loads can be 'free', or take one or more cycles to complete.

To support the correct execution of multi-threaded workloads, locks and barriers need to be handled correctly. Code can lock and unlock data with the following calls

```
void ConTuttoAllocation::lock(size_t index, size_t threadId = 0)
void ConTuttoAllocation::unlock(size_t index, size_t threadId = 0)
```


Within the simulated CPU caches the requested address, once available in the local cache, gets locked, and other threads are not allowed to touch it. After the necessary operations are performed, the address is unlocked again.

Barriers are issued to the system-simulator by the main thread on the host system, to be distributed to all the simulated cores within the system-simulator. When a simulated core encounters a barrier, is has to wait until all other cores also reached the barrier. The synchronization of the memory system, which is implemented with a *dcs* instruction on the prototype platform, is implemented in the system-simulator by 'simply' waiting until all work queues for the simulated CPU cores are empty. Below we show an example workload including barriers and synchronization.

```
// some parallel code

// make sure all host threads are here
#pragma omp barrier
#pragma omp master
{
  // make sure simulated cores wait for each other
  void ConTuttoManager::SimBarrier() // non-blocking call
}
#pragma omp barrier

// other parallel code, input depends on results from previous code,
//  thus requiring the preceding barrier

#pragma omp barrier
```

```
#pragma omp master
{
  // make sure all queues in the system-simulator are empty
  void ConTuttoManager::SyncMemSystem() // blocking call
  // now we can get timing information
  T ConTuttoManager::GetSimTimingInfo() // blocking call
}
#pragma omp barrier
```

5.3.3. INTERACTING WITH THE SYSTEM-SIMULATOR AND SYSTEM-SIMULATOR SETUP

Figure 5.7 shows a detailed picture of the application and the system-simulator. The communication between the two is done by means of Unix named FIFOs, which has the advantage of realizing a very high throughput at a low latency of messages, but implies that the application and the system-simulator run at the same machine. A version communicating over TCP-IP was also implemented, thereby being able to run both applications on different machines, but this turned out to be too slow.

A large majority of the messages sent over this link will be loads, stores, and bubbles, as described earlier. There are however also system-simulator specific messages. At startup time, the system-simulator is initialized with parameters like the number of CPU cores, the cache sizes, number of memory channels, memory bandwidth, frequencies, etc. During runtime, the system-simulator can be asked for timing information and other statistics like bandwidth utilization. The *synchronization* and *flush* commands have their own messages as well.

The system-simulator itself consist out of two main components: 1) the hardware components, and 2) a software library. The hardware components are the NDP-M and NDP, both developed in a hardware description language. These components are compiled by means of Cadence NCSim. The software library contains the rest of the simulated memory system and all communications mechanisms etc., and is discussed in the next section. During compilation, we link our system-simulator software library, developed in C++, as *Verilog Procedural Interface* (VPI) library with the hardware design. The VPI allows us to observe values of registers and wires in the hardware design from software, force values from software to hardware. Furthermore it allows us to call software methods from hardware. At compile time, we indicate a method in the software library ('-loadvpi library.so:register_VPIMethods') which registers all methods and their signature which should be callable from hardware. Similarly, we include an *accesslist* file during compilation containing signals in the hardware design which should be observable from software. At startup time, we start NCSim loading the correct workspace, which initializes and starts both the hardware components as well as the software library.

5.3.4. SYSTEM-SIMULATOR SOFTWARE LIBRARY

The system-simulator software library is, together with the hardware modules, one of the two big components in the simulation environment. The library is linked with the NCSim hardware-simulator as a VPI library, but in fact behaves as the leading component in the entire system-simulator. In Figure 5.8 we show the main components and

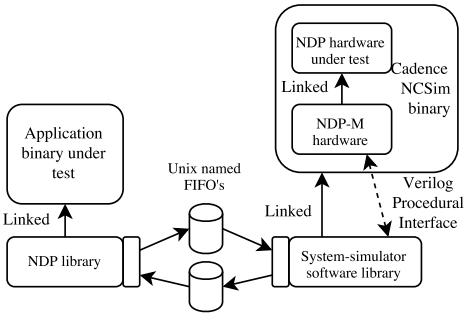


Figure 5.7: Overview figure of how the various applications, libraries and hardware components interact with each other.

how they are connected to the hardware modules. The library offers a multi-threaded communication interface which receives the loads and stores from the application, and forwards those to the simulated CPU caches, based on the *threadId* provided, and discussed in Section 5.3.2. In case the cache had to service a load from the application, the response, when available, is returned to the communication interface. The memory channels connect, by means of the VPI interface, to the hardware components described earlier. We take a closer look at these VPI mechanisms when considering the (software) memory channel <-> (hardware) NDP-M interface. The hardware checks every clock cycle whether the software has set a *load* or *store* register to '1' , and processes the access if that is the case. Responses from the NDP-M towards the CPU work the other way around: in this case the hardware calls a method notifying the software that certain values are ready to be read from hardware to software. By notifying the software that certain values can be read, instead of continuous polling, we limit the number of VPI actions, as those are expensive and limit the system-simulator performance. To be able to store large amounts of data and retrieve it efficiently, the main memory is again implemented in the software library by means of a standard hash map. The hardware - software connection between NDP-M and the main memory is implemented in a similar way as described for the memory channels.

Shown in Figure 5.8 is a hardware clock feeding into the software library. Based on the used ratio between CPU and memory-system frequency, the caches and memory system can do a certain amount of operations per received hardware tick. The memory channels are furthermore limited by their bandwidth and latency parameters. The communication methods are implemented in a multi-threaded and asynchronous way, and share data with the synchronous part by means of double-ended queue objects.

5.3.5. SIMULATING GENERAL-PURPOSE NDPS WITH A HARDWARE NDP-M

Until this point, we discussed the ability to test NDP designs developed in a hardware description language. The sorting work presented in Chapter 7 is a good example of this. Running more complicated algorithms on one or more general-purpose NDPs is however also a valid proposal. This could be either softcores on an FPGA as well as cores on a future ASIC design. To evaluate the challenges involved here, as well to evaluate the

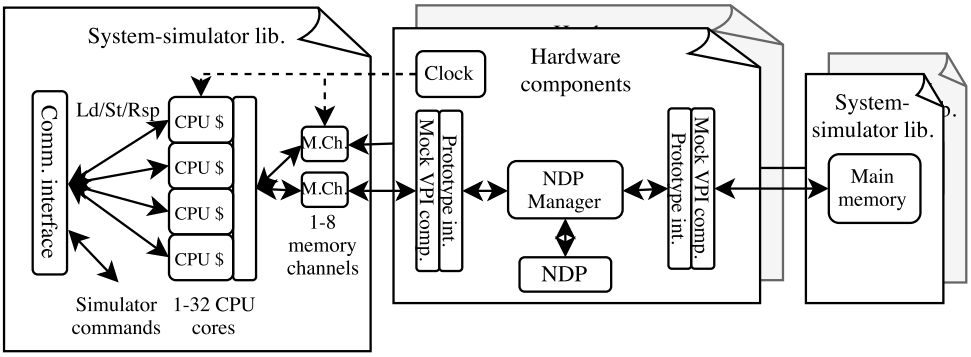


Figure 5.8: Overview of the system-simulator. Clearly identifiable are the parts implemented in hardware, and how they are connected to the parts implemented in software.

kind of optimizations that can / must go into the NDP-M, we developed a method to 'run' arbitrary code on general-purpose NDPs, without having to implement actual soft-cores, which would be an unrealistic amount of work and furthermore result in very slow simulation times. We reused the software used to simulated the CPU caches and memory channels as NDP cores, and inserted the resulting loads and stores into a hardware crossbar, functioning as interconnect. This is illustrated in Figure 5.9, complementing Figure 5.8.

Software kernels of arbitrary algorithms intended to be executed on the general-purpose NDPs can now make use of similar mechanisms as kernels intended to be executed on the simulated CPU. The methods shown in Section 5.3.2 are extended like this:

```
T ConTuttoAllocation::readSimAtNDP(i, omp_get_thread_num()) == 0)
void ConTuttoAllocation::writeSimAtNDP(i, omp_get_thread_num()) == 0)
```

which now forward the load and store request to the intended NDP core. In this case, the *ConTuttoId* maintained in the *ConTuttoAllocation* dictates to which NDP a memory access is send, while the *threadId* supplied as argument dictates the NDP-core. In contrast to the loads and stores issued to the simulated CPU caches, the address is now not translated in the NDP library, but is kept virtual. The translation happens at the NDP-M, similar to when using hardwired NDPs.

Similarly, mechanisms for flushing and synchronization are extended to the general-purpose NDP cores. Extra care is necessary for barriers however. To make sure correct ordering is ensured towards commitment in memory, the barrier must travel through the in hardware implemented crossbar as well, complicating its design.

5.3.6. LESSONS LEARNED

We describe some of the key lessons we learned by developing and using the system-simulator that drove the architecture described in Chapter 4.

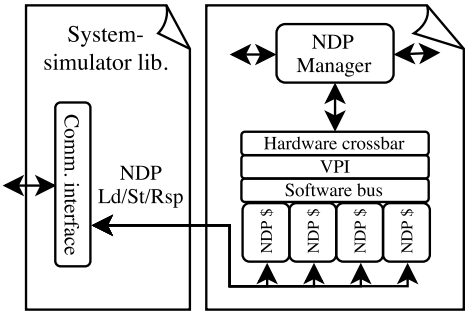


Figure 5.9: Complementary to Figure 5.8, showing the ability to run arbitrary software on general-purpose NDP cores, while still using the hardware NDP-M.

VIRTUAL MEMORY MANAGEMENT

With all the discussed virtual memory management features implemented as discussed, we ran into issues every now and then. For no apparent reason, in the middle of a run, the software mechanism for translating a virtual address to a real address (to be supplied to the system-simulator), would assert. In this mechanism, we used the stored lookup tables in the *ConTuttoAllocation* object, but, as a check, also did a clean translation of the address based on the *pagemap* mechanism as described in Section 5.2.3. Every now and then the two resulting real addresses would not be the same. This let into the realization of the complexities involved with virtual memory support, of which some are highlighted in Section 4.3, and hardware support for this in the proposed architecture in Chapter 4.

PREFETCHING AND MANUAL CPU-NDP COHERENCE

The simulated CPU cores support prefetching of data. In our case, this is done in a manual way, but the principle is the same when having a hardware module doing it transparently. Also described is the need for manual coherence between the CPU and the NDP, as the prototype platform would not allow us to do this in a hardware managed way. This implies that the programmer has to keep track of which data sets he or she believes are in the caches, and which are in the main memory. Prefetching data based on a guess (i.e. stride detection and extrapolation), breaks this principle, which was discovered while researching sorting data sets (Chapter 7). A kernel execution working on only a sub set of the data set, would also prefetch data not used in that specific kernel execution, thereby frustrating the understanding of which data is in the caches and which data is not. This gets even more complicated as hardware prefetchers do not even know about the start and end of a data set, and would even start prefetching data of another data set if it is physically adjacent to the one working with. These problems, and many other factors as described in Section 4.5, let to the understanding that a hardware managed coherence protocol between CPU and NDP is essential for successful deployment.

5.4. PERFORMANCE SIMULATOR FOR COMPLEX SYSTEM-LEVEL INTERACTIONS

The system-simulator presented so far is a powerful tool to test and evaluate NDP hardware designs and their system integration, as well as aid the development of a prototype system. By means of the general-purpose NDPs, we can even run arbitrary software kernels connected to the hardware NDP-M. However, the implementation of complex features, such as coherence and remote data accesses, in hardware was infeasible within the scope of this work. Besides, given the hardware core of the system-simulator, simulation times are slow. This led to the development of the performance-simulator, in which every component is modeled in software, allowing the testing of more complex features and higher simulation speeds. In Figure 5.10 we shows the performance-simulator, with clear similarities as well as differences with Figure 5.8. Foremost difference is the addition of the NDP-AP component, and the software-only NDPs and NDP-M. The NDP-M, the NDP-AP, and the rest of the system, now capture all complex features described in Chapter 4, and drive the performance results in Chapter 4 and Chapter 6. To further increase simulation speed, reduce implementation complexity, and increase the ease of use, we also removed the feature of actually running the application in the simulator. Instead, the performance-simulator traces the loads, stores, and other instructions to the best extent, but does not keep track of the associated data, thereby 'only' producing performance information.

The simulated NDP-M implements, as described in Section 4.4 and shown in Figure 4.4:

- Load / store queues.
- Coherence manager maintaining NDP-M <-> CPU coherence.
- Extended coherence directory cache.
- Extended coherence directory cache summary.
- Issuing / managing of remote data accesses.
- Issuing / managing of coherence messages from / towards CPU.
- Interfacing with the NDP.
- ERAT (only as latency penalty).

The simulated NDP-AP implements, as described in Section 4.4 and shown in Figure 4.5:

- Data cache
- Accessing remote data on request of the NDP-M(s).
- Handling coherence messages to maintain CPU <-> NDP-M coherence.
- Issue atomics, remote data locks, etc.

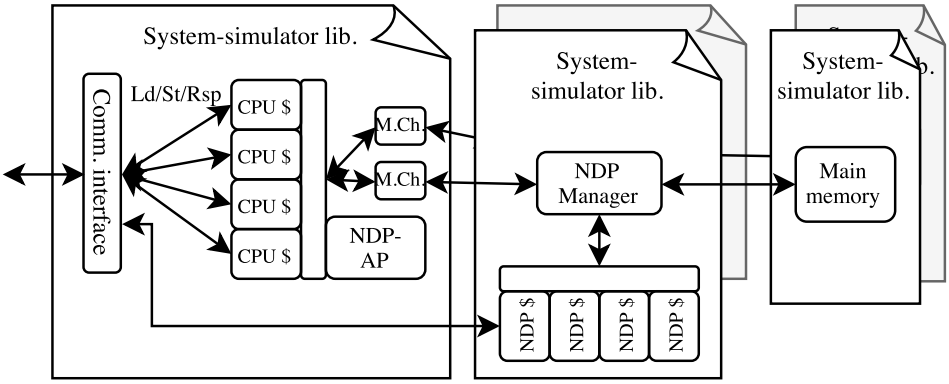


Figure 5.10: Overview of the performance-simulator. The NDP-M and NDP-AP are implemented in software supporting complex system-level interactions. The NDPs are general-purpose cores.

The other components, e.g. the memory channels, are implemented in a relatively simple way, able to capture their bandwidth and latency characteristics.

The physical distribution of data among the various memory channels is managed by a central *AddressMapManager* component, which keeps track of the virtual address ranges of relevant data sets and their physical *home* location. This avoids the need of explicit data set allocation management as is done with the *ConTuttoAllocation* class. The mapping maintained by the *AddressMapManager* is sent to the performance-simulator by means of a new set of messages. The coherence state of all data is kept in a central *CoherenceStore* component, and is not maintained in separate directories. Meaning that, the ECD-$, and other data structures like the NDP-AP data cache, do keep track of which data they hold on to, but the state of that data is kept in the *CoherenceStore*. Coherence messages still have to be send, and rights have still to be claimed and exchanged between components, but deadlocks and the need of various transient-states are avoided. This way, all interesting behavior of the proposed architecture can be captured and studied, without having to implement all the tedious details associated with an actual coherence implementation in such a complex system. In some boundary cases this could lead to different behavior than experienced on an actual implementation, but it does not lead to different application level results.

5.4.1. INTEL PIN FRONT-END

Given we do not need the data associated with every load or store anymore, we can use a more generic front-end to feed the performance-simulator, in contrast to the usage of the *ConTuttoAllocation* class mechanism described before. As illustrated in Figure 5.11, a PIN tool [190] is used to instrument the application under test and capture all the load and stores. These are issued to the performance-simulator using the same interface as the NDP library uses. Initializing the performance-simulator, and, e.g., retrieving timing information, still uses the NDP library communication path. Special calls in the application can start the instrumentation process, map OS thread-ids to performance-simulator thread-ids, and send barriers. This thread-mapping mechanism takes two lev-

els of parallelism into account: first level is the NDP, followed by the thread on that NDP. The call to map an OS thread to an NDP thread looks like and is executed by every thread:

```
setNDPThreadMap(NDPIndex, ThreadIndex);
```

Which is captured in the PIN tool where the mapping is stored and used to issue the loads and stores from the real thread into the performance-simulator using the correct simulated NDP and NDP-core. A typical use case of this call is shown below:

```
setNDPThreadMap(0, 0);
#pragma omp parallel num_threads(NDPs)
{
  size_t NDPIndex = omp_get_thread_num();
setNDPThreadMap(NDPIndex, 0);

  #pragma omp parallel num_threads(NDPcores)
  {
    setNDPThreadMap(NDPIndex, omp_get_thread_num());

    #pragma omp for
    for (size_t i=0; i<count; i++) {}

    unsetNDPThreadMap(NDPIndex, omp_get_thread_num());
  }
}
```

The initial thread-mapping at the very first line is to also include the code executed to spawn all the threads in the benchmark, as this turned out to significantly affect performance as we are typically simulating many, very slow, NDP cores.

As the application under test does not wait anymore for the response of loads issued to the performance-simulator, the application and the performance-simulator will run asynchronously. If the host hardware platform supports enough threads, all the NDP threads will quickly overflow the performance-simulator with memory accesses, and therefore back-pressure mechanisms are implemented. When the application does a blocking *ConTuttoManager::SyncMemSystem()* call, the queues in the performance-simulator are emptied before continuing.

5.5. RESULTS

With the developed system-simulator, we tested several applications and coprocessors, which are not all reflected in the other chapters of this work, and are therefore described below.

5.5.1. CLEAN - SKA IMAGING

The imaging pipeline is one of the most compute intensive aspects of the SKA radio-telescope. CLEAN is an essential step in the imaging process, and due to its limited complexity, made for an interesting test- and use-case. CLEAN takes as an input a so called

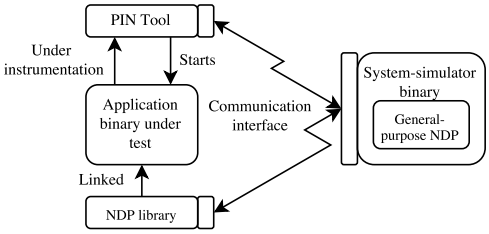


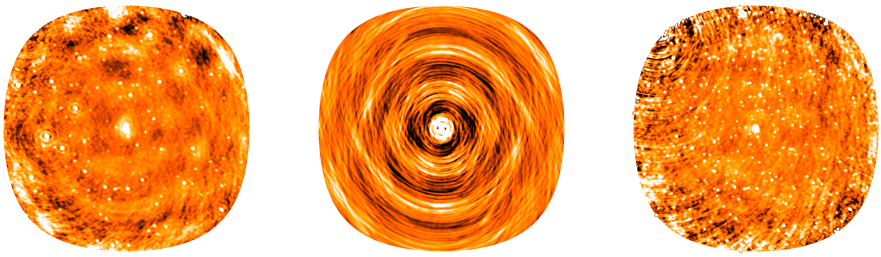
Figure 5.11: High level overview of the performance-simulator, with an Intel PIN front-end.

dirty sky image, and iteratively deconvolves a point spread function (PSF), representing the telescope response-function, from it, until it arrives at a *clean* image. The three images involved here are shown in Figure 5.12, showing on the left a dirty image, in the center the PSF, and on the right the clean image. The clean image shows many more sky sources (dots), representing stars or other objects, and less 'garbage' around the sources.

Two methods are of foremost interest for CLEAN, as they dominate the 'inner loop' of the application. The first method searches for the brightest pixel in the image. The second method centers the PSF around that pixel, and subtracts it from the image. As the image changes with every iteration, the next brightest pixel can be anywhere, resulting in little data locality. For very large images, the caches of a CPU become useless, and for every iteration we have to load the image from main memory to search for the largest pixel, and then load all surrounding pixels to subtract the PSF. We optimized these two methods, focusing on avoiding data transport and bandwidth bottlenecks.

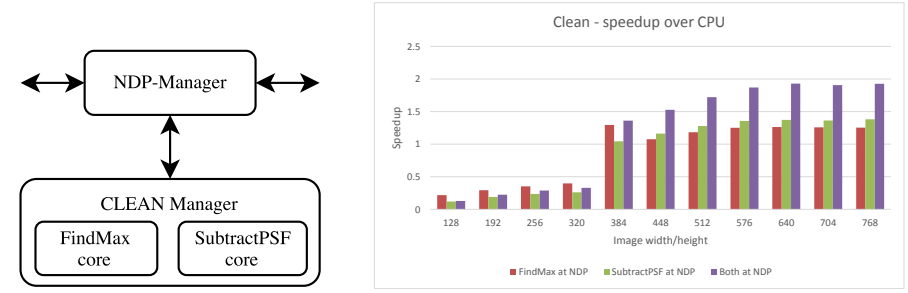
On the software side we used the CLEAN implementation found in the standard radio-astronomy CASA package [204]. To obtain a large amount of test images of various sizes, we developed a Matlab script able to generate dirty images using various parameters. We developed two hardware NDPs in VHDL able to execute the discussed methods, and they are, together with a small manager component, connected to the NDP-M. This is illustrated in Figure 5.13a. The FindMax core is a simple streaming function, while the SubtractPSF core is more complicated, especially due to the format the image is stored in a typical CLEAN implementation, of which the explanation is beyond the scope of this section. We adjusted the software to run in the system-simulator, and extended the discussed methods with the option of using the NDP.

In Figure 5.13b, we show the performance results for running many iterations of CLEAN on a system containing two CPU cores with a 64 KB cache each and a single memory channel. We show the results for executing one of the two, or both, methods on the NDP. When offloading only one of the methods, coherency traffic is introduced, which is a significant penalty when considering small images. For larger images, all types of offloading speed up the processing with respect to the CPU solution. An interesting observation is that the biggest speedup when offloading only *FindMax* is achieved for an image 384x384 pixels in size. For this size, the *FindMax* related data structure just fits in the two caches, and the *SubtractPSF* data structure is foremost accessed by the NDP. In case of executing both methods on the CPU, both datasets would be fighting over the cache every single iteration. By using the NDP for only *FindMax*, we speed up both *FindMax*, as well as *SubtractPSF*, since the latter now has exclusive usage of the cache,



(a) (b) (c)

Figure 5.12: Three CLEAN related images. Left an input dirty image, in the middle a point-spread function representing the telescope response function, and on the right the dirty image deconvolved with the point-spread function.



(a) (b)

Figure 5.13: NDP hardware design for CLEAN implemented in VHDL, showing the two type of coprocessors for the two offloaded methods. (b) Results for clean, showing speedups when running one of the two, or both, methods on the NDP. Horizontal axis shows the image-size along a single side.

resulting in a higher speedup than we see for larger images, where the *SubtractPSF* data set does not fit in the cache anymore. For this application several NDP advantages show off, the higher bandwidth, the higher IOPS, and the possibility of workload-optimized logic for the NDP.

5.5.2. GRAPH500

Graph500 is a complex graph traversal workload already discussed in Chapter 4. As we deemed it too complex to develop a workload-optimized hardware NDP for it, we used the general-purpose NDP cores as discussed in Section 5.3.5, still using the hardware NDP-M. Note that this is different from the Graph500 implementations discussed in Chapter 4 and Chapter 6, which both use the performance-simulator version. We used the MPI parallel *bfs_replicated* version of the Graph500 reference code for our implementation.

ALL-TO-ALL COMMUNICATION

As the *bfs_replicated* name suggests, this implementation replicates the *frontier* bitmap across all nodes, in contrast to implementations where every node has a piece of the bitmap. By doing so, there is no inter-node communication in the kernels. Only after each level is finished is there an all-to-all binary *or* operation to update the bitmaps with the results from the other nodes. This implementation is somewhat simpler than having a distributed bitmap, but does not scale to more nodes and larger problems. As the prototype platform, and thus the system-simulator, does not support direct inter-NDP communication, the all-to-all runs on the CPU cores. This can be implemented rather efficiently and achieves a high bandwidth, and given we are (for now) only interested in up to eight NDPs, the scaling issues are not a problem.

IMPLEMENTATION OF BARRIERS AND SYNCHRONIZATION

The system-simulator actually executes the workload, and holds the values the application is working with. To obtain the correct Graph500 results and the correct performance measurements, all used aspects of the system-simulator must therefore be implemented flawlessly. Given the multi-threaded and multi-node nature of the workload, this was not a trivial task.

At the NDP level, foremost the correct implementations of barriers proved challenging. Coherence between the cores is difficult as well, but was already solved as the general-purpose NDPs make use of the same code as the simulated CPU caches. A barrier implies that every core has committed all its memory accesses to the memory system, and wait for each other, to make sure everyone observes the same and correct values afterwards. The barriers therefore also have to travel through the crossbar shown in Figure 5.9, to make sure a read does not overtake a write of a different thread. An extra difficulty is that a barrier not necessarily involves all the NDP cores and thus not all ports of the crossbar.

Similar problems arise between the CPU and the NDP. Before entering the all-to-all communication phase, all NDP accesses must have been committed to the main memory. Likewise, before starting the NDP kernels again, all CPU accesses must have been committed to the main memory. As described for the system-simulator, these mechanisms cannot be implemented completely correct. In the system-simulator, the *ConTuttoManager:syncMemSystem()* therefore, after waiting for all queues to be empty as defined, waits a couple of dozen cycles, for the NDP accesses to propagate through the crossbar and the NDP-M, and for the CPU accesses to propagate through the memory channels and the NDP-M.

RESULTS

We succeeded in a correct implementation of the system-simulator as well as a correct implementation of Graph500 running on that system-simulator, together able to generate valid breadth-first graphs, using up to eight NDPs and the CPU for the all-to-all communication. The performance results achieved with this implementation are to some extend rendered obsolete by all Graph500 results produced in Chapter 4 and Chapter 6, and we will therefore focus on the lessons learned described in the next section.

5.5.3. LESSONS LEARNED

COMPLEX INTERACTIONS OF DATA SETS

The execution of CLEAN in the simulation environment gave us an interesting insight in how more than one method and their related data sets interact with each other when running at the NDP or not. This is a key insight that could only have been obtained by running the entire application and all its methods in the system-simulator. It is a clear example that it is not always trivial what to run on an NDP and what not, since the current location of the data as well as the location of other data, the current coherence state of that data, and the access profile over time, need to be known to make a complete and informed decision.

CORRECT SYNCHRONIZATION AND CONSISTENCY

With Graph500 we stumbled onto issues regarding consistency, coherence, and synchronization. This led to the extensive study of this subject, resulting in the proposed architecture in Chapter 4. A key aspect of the design is to decide when an access has entered the global coherent space. For an NDP with hardware managed caches this is the case when the access is out of the load/stores queues and committed to the cache. For NDPs without hardware managed caches, like the ones as described above, commitment happens only at the level of the NDP-M, meaning that barrier / synchronization also has to travel to that level before returning. When the CPU and the NDP are both part of the same global coherent space, as in the proposed architecture in Chapter 4, no explicit CPU-NDP synchronization is necessary. If both devices do a synchronize, both devices will observe the correct values.

6

BOOSTING THE EFFICIENCY OF HPCG AND GRAPH500 WITH NEAR-DATA PROCESSING

To evaluate the architectural proposal, as well as the validity of the near-data processing concept, thorough evaluation is needed. We promote the use of industry-standard benchmarks, and the use of existing real-life computers as comparison, to avoid comparing the performance of arbitrary applications against arbitrary simulated system design points. By doing so, the performance of the studied architecture has a meaning in the absolute sense. It makes it possible to put our results next to already published results studying the benchmarks on high performance computer systems.

The content of this chapter is based on the following paper:

Boosting the efficiency of HPCG and Graph500 with near-data processing
E. Vermij, L. Fiorin, C. Hagleitner and K. Bertels
International Conference on Parallel Processing, 2017

ABSTRACT

HPCG and Graph500 can be regarded as the two most relevant benchmarks for high-performance computing systems. Existing supercomputer designs, however, tend to focus on floating-point peak performance, a metric less relevant for these two benchmarks, leaving resources underutilized, and resulting in little performance improvements, for these benchmarks, over time. In this work, we analyze the implementation of both benchmarks on a novel shared-memory near-data processing architecture. We study a number of aspects: 1. a system parameter design exploration, 2. software optimizations, and 3. the exploitation of unique architectural features like user-enhanced coherence as well as the exploitation of data-locality for inter near-data processor traffic.

For the HPCG benchmark, we show a factor 2.5x application level speedup with respect to a CPU, and a factor 2.5x power-efficiency improvement with respect to a GPU. For the Graph500 benchmark, we show up to a factor 3.5x speedup with respect to a CPU. Furthermore, we show that, with many of the existing data-locality optimizations for this specific graph workload applied, local memory bandwidth is not the crucial parameter, and a high-bandwidth as well as low-latency interconnect are arguably more important, shining a new light on the near-data processing characteristics most relevant for this type of heavily optimized graph processing.

6.1. INTRODUCTION

The LINPACK benchmark [55] has been the golden standard for benchmarking supercomputers for many years, and it represents the metric used for drawing up the highly regarded TOP500 list of fastest supercomputers in the world [3]. However, the LINPACK benchmark does not represent anymore modern day's workloads, as it foremost stresses the compute capabilities of a system. In fact, novel workloads in the field of high-performance computing show a low compute-to-bandwidth ratio, as well as irregular memory accesses.

The HPCG [205] and Graph500 [58] benchmarks are gaining momentum as industry standard benchmarks next to LINPACK. HPCG is a preconditioned conjugate gradient algorithm, consisting of a variety of kernels together capturing a significant amount of representative compute and communication patterns. Graph500 is a data-intensive benchmark, doing several breadth-first searches (BFS) in a large graph, stressing features like random memory accesses and global communication in a cluster. Graph500 models big-data analytics problems of companies and governments dealing with massive amounts of unstructured data.

Although HPCG and Graph500 solve very different problems, both prefer a *balanced* system, in which the compute capabilities, the memory bandwidths, and the network specifications match. However, at the node level, the last decade has presented us with systems offering less bandwidth per compute with every product generation. This is true for CPUs, but even more for GPU coprocessors, which typically offer a high bandwidth, but also an even higher floating-point performance. To battle the mismatch between what the benchmarks (and thus many real-world applications) demand and what computer systems can offer, the near-data processing paradigm has recently been rediscovered [73]. This paradigm refocuses system-design at the memory, aiming at high

bandwidths and low latencies, benefiting modern applications.

In this work, we analyze the usage of near-data processors (NDPs) to boost the performance and foremost efficiency of these two key HPC benchmarks. We will study both a variety of parameters for a given hardware architecture, as well as several software optimizations to optimize the applications for the architecture. The main contributions of this paper are as follows:

- We present for the first time a design exploration for a near-data processing architecture, and complimenting software optimizations, targeting two relevant HPC industry benchmarks.
- We show the relevance of the novel architectural features like the exploitation of data locality for inter-NDP traffic and user-enhanced coherence.
- We show, for the HPCG benchmark, a factor 2x to 2.5x performance and power-efficiency improvements with respect to CPU and GPU based solutions, as well as a much higher utilization.
- We show, for the Graph500 benchmark, that extensive software optimizations make inter-NDP bandwidth equally important compared to local memory bandwidth.

The remainder of this paper is organized as follows: In Section 6.2.2 we discuss related work regarding the two benchmarks and near-data processing, followed by the architecture in Section 6.3. In Section 6.4 and Section 6.5 we discuss the optimizations and results of HPCG and Graph500, respectively, followed by conclusions in Section 6.6.

6.2. MOTIVATION AND RELATED WORK

6.2.1. MOTIVATION

Figure 6.1 shows the floating-point utilization (fraction of compute capabilities used) for the top 10 supercomputers running LINPACK as well as HPCG, taken from their respective websites [3] [57]. It can be observed that LINPACK reaches a very high utilization, between 75% and 95%. HPCG on the other hand, shows a very low utilization, with the *K* system scoring the highest value with a 5.3% utilization, and the *Sunway TaihuLight* scoring the lowest value with a 0.3% utilization. Showing compute utilization for the Graph500 benchmark is not possible as it does not use floating point compute instructions.

Figure 6.2 shows the performance of the top 10 supercomputers running LINPACK, HPCG, and Graph500, normalized against the current number one system (*Sunway TaihuLight*). For LINPACK, the results obviously decrease towards the number 10 in the list, as it is the benchmark used to draw up the list. For HPCG, we see very different results. Several systems score better than the number one, and until we reach number eight going down the list, there is no decreasing performance trend at all, while for LINPACK, these systems already run 10-20x slower than the number one. For Graph500, unfortunately not all systems have an entry, but the analysis is very similar to the one obtained with HPCG. Figure 6.2 also shows the introduction date of the system: it is notable that the *K* system and the *Sequoia* system, both performing equal or better than the *Sunway*

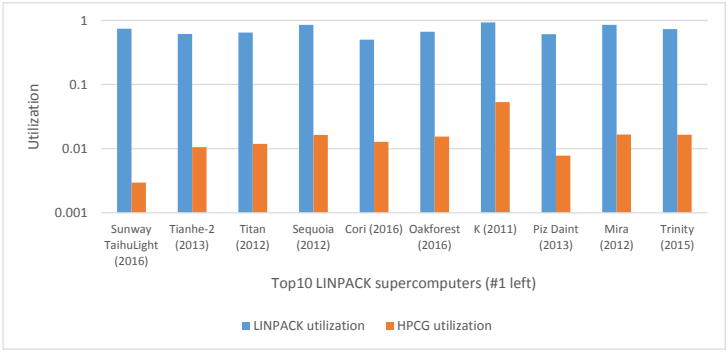


Figure 6.1: The utilization of the top 10 (out of the TOP500) supercomputers, running the LINPACK and the HPCG benchmark. [3] [57]

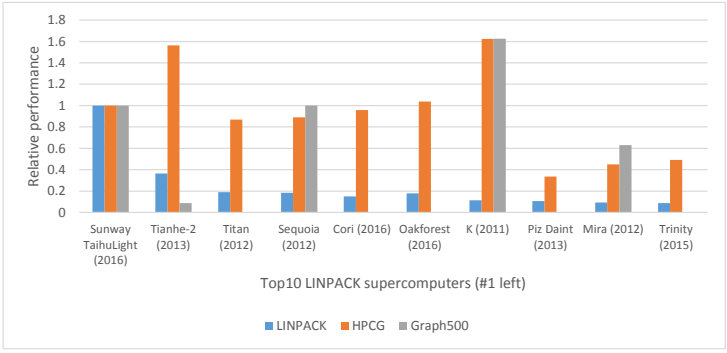


Figure 6.2: The relative performance of the top 10 (out of the TOP500) supercomputers, running the LINPACK, HPCG, and Graph500 benchmark. [3] [57]

TaihuLight system, are five and four years older than the current number one supercomputer, respectively.

Figure 6.1 and Figure 6.2 clearly sketch that the behavior of modern supercomputers running LINPACK is very different compared to the two modern and relevant benchmarks. When considering progress over time, it is clear that little progress has been made in the last years to improve their performance. Apparently, the metric we focus on when developing new supercomputers, is not the most relevant one when considering modern workloads. To improve the performance and the efficiency for HPCG and Graph500, a focus on a novel class of hardware architectures is needed, which is the main topic of interest in this work.

6.2.2. RELATED WORK

The amount of available research done on HPCG is still rather limited. Early work performed in [206] distills a performance model for the benchmark running in a cluster, concluding that the main-memory bandwidth is a key parameter. In [207], an evaluation of parallelization strategies for one of the compute kernels is presented, as well as per-

formance numbers for various Intel platforms. This research is updated in [208], showing extra performance numbers when using optimized sparse matrix methods. Research presented in [44] shows the benchmark running on GPUs using single-source code using OpenMP 4.5 pragma's. Although not about HPCG, interesting work is presented in [209], presenting a deeply optimized sparse matrix-matrix multiplication engine in the logic layer of 3D stacked memory. We complement this work by showing results on a novel bandwidth oriented architecture, for which we study key parameters like the data cache size, the memory access granularity, and bandwidth. Furthermore, the impact of features like software prefetch and user-enhanced coherence is analyzed.

The performance of Graph500 has been actively researched on supercomputers [194], and optimized solutions can traverse trillions of edges per second on big supercomputers. An optimized shared-memory implementation of Graph500 is presented in [210], where the key optimizations are the sorting and rearrangement of the data structures to enhance data locality, letting the performance scale well to tens of sockets in a NUMA system. Another thorough NUMA system analysis is performed in [211], focusing on the tradeoffs in either copying or sharing data sets across sockets, and the optimized use of several critical data structures. We complement this work by optimizing Graph500 for an architecture with very limited local memories, but with the ability to exploit data locality for remote data accesses. Other explored features are user-enhanced coherence, and software prefetching.

Near-data processing architectures have been presented in a variety of ways in the last years. In [98], a special-purpose graph processing architecture is presented, utilizing the logic layer of stacked memories to achieve very high performance. The proposal uses custom hardware prefetchers and messaging techniques to utilize all available bandwidth. A more general approach is presented in [88], proposing general-purpose cores in the logic layer of stacked memories, accelerating highly parallel analytics workloads. Work in [212] and [91] explores the use of wide vector units inside stacked memory targeting HPC workloads. Executing standard benchmarks on near-data processing systems, enabling fair comparisons between approaches, is not explored so far.

6.3. SYSTEM DESCRIPTION

A high-level view of the studied system is shown in Figure 6.3, based on the work presented in [213]. We use as a basis the memory system of the POWER8 CPU [34], which has two levels of memory controllers (MCs): at the CPU level, we have eight memory-technology agnostic MCs, while the technology-specific (DDR3/4) MCs are located on separate 'memory hub chips', tightly coupled to the main memory. These chips are connected to the CPU by means of CPU- and memory-agnostic high-speed serial links. The architecture in Figure 6.3 introduces processing capabilities at the level of this chip. The concept is not tied to the POWER8 and its memory system, but also applies to, e.g., a CPU with the Hybrid Memory Cube [35] attached, where the 'memory hub' is in the logic layer, and the technology specific memory controllers are the so called 'vault controllers' [35].

From Figure 6.3 we can distill three key components of the studied architecture, highlighted as dashed blocks in the figure: 1. NDP cores, 2. NDP Manager, 3. NDP Access Point. The key parameters of the studied architecture are described in Table 6.1. This work uses the architecture proposed and discussed in [213], and does not discuss the

detailed workings of e.g. the coherence interactions, data placement, or virtual memory support beyond what is described below.

1. NDP cores—Every memory channel contains one NDP, consisting of several general-purpose NDP-cores. Every NDP-core can coherently access data in the global virtual address space, spanning multiple memory channels or even sockets. For this work we consider very small and slow cores, with their specification shown in Table 6.1.

2. NDP Manager—The NDP is connected to the NDP-Manager (NDP-M), which offers an interface into the system, including virtual-memory support and coherence. The NDP-M keeps track of the coherence state (*CPU owned*, *NDP owned*, *Shared*) of every cache line sized block of main memory in a directory, which it can access by means of a small cache, as shown in Figure 6.3. In case a line is needed by the NDP but owned by the CPU, the NDP-M sends the appropriate coherence message to the NDP-AP, which issues it to the global coherent space. The other way around, where the CPU wants a line owned by the NDP, works likewise. Data from a remote memory channel can not be cached in the hardware managed caches of an NDP-core, as we would not be able to find it anymore without extending the CPUs coherence snooping towards the NDPs, which would have a dramatic impact on the system performance.

3. NDP Access Point—At the CPU the the NDP Access Point (NDP-AP) is introduced. This is the point where NDP accesses to non-local data enter the global coherent address space. Communication between the NDP-M and the NDP-AP is done with special transactions over the memory channels and the system bus. An NDP access to remote data (not stored in the local memory channel) results in an transaction towards the NDP-AP, which performs a read in the global coherent space, and returns the requested values to the NDP. The NDP-AP has a small data cache (see Figure 6.3) to hold frequently accessed remote data, avoiding the need for data to be fetched from a memory channel in case of a hit.

6.3.1. HIGHLIGHTED FEATURES OF THE ARCHITECTURE

In this section we will discuss three specific features of the studied architecture that can be used to improve performance.

First, as can be observed in Table 6.1, the data cache sizes of the NDP cores explored in this work are very small, limiting the amount of locality that can be exploited for local accesses. The NDP-AP has a much larger data cache, to be able to store remote data accessed by the NDPs. In modern multi-node CPU architectures, the latency towards a remote memory is only about 25% higher than to local memory [214]. On the studied architecture however, this latency is, based on the system specifications provided in Table 6.1, about three times larger than the latency for accessing local memory, thus penalizing remote memory accesses hard. The exploitation of locality for remote accesses by means of the NDP-AP data cache, and thus reducing access latency, is a key feature when the NDPs access the same data set, as will be shown for foremost the Graph500 benchmark.

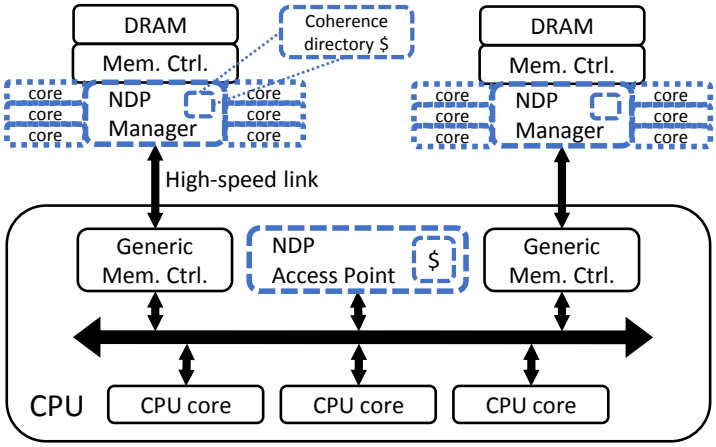


Figure 6.3: The studied system, with processing capabilities included close to the main-memory of a CPU.

Second, the architecture introduces the concept of *user-enhanced coherence*. The implemented method for coherence introduces unavoidable overheads, e.g. the coherence directory lookup at the NDP-M and the cache-inhibited nature of remote data for the NDPs. The programmer can try to avoid these overheads in two ways: first, by instructing the NDP-M to give an entire address range a certain coherence state and keep that in a small separate data structure. This way, e.g., the entire graph data structure used in Graph500 can be set to *Shared*, because after initialization it will be read-only, thereby elimination all coherence directory lookups. The process of monitoring address ranges for coherence optimizations can be realized in a hardware-managed fashion as well [34], but a user-level control lets us optimized this problem efficiently by utilizing knowledge about the application. Furthermore, by marking data sets *remote cache uninhibited* (meaning the data can be cached in non-local NDP caches), the NDP-cores can exploit locality on remote data, at the expense of the extra programming effort needed to flush the NDP caches at the correct moments when coherence is required.

Last, we highlight the execution of atomics by the NDP-AP. Since the NDP-core caches can not hold data belonging to another memory channel in a coherent way, atomic operations on remote data have to be performed in collaboration with the NDP-AP. By default, the cache line lock is done at the NDP-AP, after which the NDP can check/update the data, and return it to the NDP-AP which can then release the lock, resulting in a long lock time. By executing atomics on remote data directly at the NDP-AP, we eliminate this issue.

6.3.2. SIMULATION ENVIRONMENT AND PARAMETERS

To evaluate the architecture, a custom system simulator is used. The simulator can capture instruction traces of arbitrary applications, and insert them into the correct simulated NDP cores, which are in turn connected to various NDP-Ms, the memory channels, the NDP-AP, and the rest of the system. The simulator captures, in a cycle accurate way, all relevant aspects like the coherence interactions between NDPs and the CPU, remote

Table 6.1: Baseline system specifications.

Main memory (x8 channels)	
Technology and bandwidth	4 x DDR4-2400 - DDR4-3200 @ 60% : 46 - 61 GB/s
Mem. Ctrl. + DRAM latency	40 ns
NDP (x8 channels)	
Cores	64 cores, 1 GHz, in-order, scalar, non-pipelined, instr. latencies as in [191]
Data caches	512-8192 B per core
Access granularity	32-128 Byte
Main-memory round-trip latency	60 ns (best case)
NDP-M	
Coherence dir. cache size	16 KB
Coherence dir. cache latency	4 ns
NDP-M - NDP-AP latency	24 ns (single trip)
NDP-AP	
Data cache size	8-128 KB
Access granularity	128 Byte
System-bus data interface	128 GB/s in, 64 GB/s out

data accesses (and their coherence requirements), and atomics. The specifications of the system used throughout this work is shown in Table 6.1, with aspects like latencies and interconnect bandwidths based on the POWER8 memory system [34], and the cache latencies validated by [139].

For the power analysis of the NDPs, we rely on McPat [139], providing as input the switching activity derived from the execution of the two benchmarks under study. A detailed power analysis of the NDP-M and NDP-AP is beyond the scope of this paper, but to be able make first order comparisons, we set the additional power of the NDP-M and the NDP-AP, based on [196] and [197], to 2 W and 4 W, respectively, for a combined power addition of 20 W when considering eight memory channels.

6.4. HPCG BENCHMARK

The HPCG benchmark implements a preconditioned conjugate gradient method with a local symmetric Gauss-Seidel preconditioner and a three-level multigrid. The implementation uses a regular 27-point stencil discretization in three dimensions of an elliptic partial differential equation, as found, for example, in heat diffusion models [57]. The benchmark is intended to run in a multi-node environment. As an input, the *local problem dimension* per node is defined. Depending on the amount of nodes, the code generates either a square or rectangular *global problem* in three dimensions. Every local problem is a cube, and holds the three-level multigrid described above. Throughout this section, problem sizes, both local or global, will be denoted as, e.g., 64^3 or 64x128x128, indicating the dimensions of the cube or rectangle. There are four impor-

tant kernels: sparse matrix-vector multiply (*SpMV*), preconditioner (*SymGS*), global dot-product (*Dot*), and a scaled-vector summation (*WAXPBY*). Before the *SpMV* and *SymGS* kernels, the boundary values of every local problem are communicated with the neighboring local problems. All these kernels combined realize the conjugate gradient solver, and with that the HPCG benchmark. Various properties such as the irregular memory accesses, a low operational intensity, and a neighbor communication pattern make this a relevant benchmark. The performance is reported as GFlops/s.

6.4.1. IMPLEMENTATION AND BASELINE OPTIMIZATIONS

The reference code provides reasonable performance for three of the four mentioned kernels, except for *SymGS*, which can not be parallelized at the thread level as it has loop dependencies. Therefore, a block multi-coloring method was used to restructure the data in order to eliminate the dependencies and allow parallel processing [215]. The other kernels are adjusted as well to use the new data layout. This version of the code is used as a baseline implementation. Instead of the MPI + OpenMP parallelization typically used in a distributed memory environment, we use a nested OpenMP approach for our shared-memory architecture, where we first spawn a thread per NDP, followed by a thread for every NDP-core. In this section we will explore the use of the (highlighted) features of the studied architecture described in Section 6.3.1, as well as software optimizations.

6.4.2. OPTIMIZATIONS

TOP LEVEL PARALLELIZATION

In the baseline code, every loop has its own *omp parallel for* pragma, implying that for every loop there is a single-threaded overhead to create the 64-thread large thread-team. Since our architecture's performance relies on many very slow cores, this turned out to be a huge overhead. Therefore, the code was rewritten to create a parallel region on top of all the kernels, requiring only a *omp for* pragma per loop. By doing so, the performance, when running a 64^3 problem using a single NDP, increases by 42%, stressing the need for highly parallel code on the studied architecture.

USER ENHANCED COHERENCE

As discussed in Section 6.3, the NDPs and CPU are kept coherent by means of a directory structure stored in main memory, accessed through a small cache. To ease the pressure on the coherence mechanisms, address ranges can be explicitly claimed as a certain coherence state, as indicated in Section 6.3.1. The data sets storing the matrix data will, after initialization, only be accessed by the NDPs. Therefore, it is possible to claim the corresponding address ranges to the *NDP owned* state, and thereby eliminating the need for a full coherence directory cache lookup. When considering a 64^3 local problem size, 10.6% of the memory bandwidth is needed to load coherence directory items, increasing up to 14.6% when considering a 104^3 local problem. These percentages are reduced to 0% and 2%, respectively, when the matrix data sets address ranges are claimed as *NDP owned*. The performance impact of this optimization depends on the available bandwidth, which depends on various aspects discussed in the remainder of this section. For

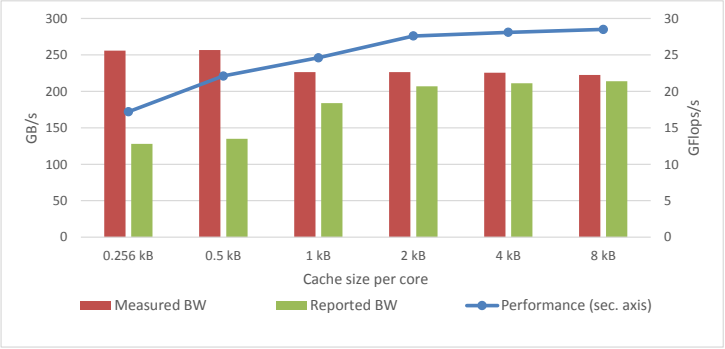


Figure 6.4: Impact of the NDP core cache size. HPCG is able to achieve peak floating-point performance with a 4 KB cache per core.

the design point of 4 KB caches and DDR2400 memory technology, the performance improvement is 8% for the largest problem size.

NDP CACHE SIZE IMPACT

We study the impact of the data cache size on performance and bandwidth usage, by considering eight NDPs and the baseline specifications depicted in Table 6.1, running a 128^3 global problem. In Figure 6.4 we show the performance, the *measured* bandwidth, and the *reported* bandwidth for a variety of data cache sizes. The *reported* bandwidth is the bandwidth listed in the HPCG benchmark result file provided by the benchmark itself, listed next to, e.g., the achieved compute performance, and is based on a fixed calculation within the code using runtime, problem size etc. The *measured* bandwidth, on the other hand, is the bandwidth reported by our simulation environment, which can be considered a 'measurement'. The difference between measured and reported bandwidth when using small caches, is clearly visible. Apparently, the calculation for the reported bandwidth assumes a certain amount of data reuse, which the tiny caches can not realize. When increasing the cache size to (a still very modest) 4 KB per core, the reported bandwidth is now much closer to reality. Performance also increases dramatically towards larger cache sizes, and stabilizes around 28 GFlops/s. From existing research it can be learned that, in a CPU-like environment, HPCG scores in the order of 10 GFlops/s per 100 GB/s of main memory bandwidth, and our results are now roughly in line with this observation. The cache size impact can be explained by the fact that the items of the diagonal matrix fetched in both the *SpMV* and *SymGS* kernels, by definition of the benchmark, are reused around 27 times [57], which can only be accommodated by a cache from 4 KB in size upwards, while also showing that larger local memories are not useful.

MEMORY BANDWIDTH AND ACCESS GRANULARITY IMPACT

Memory bandwidth has shown to be the most relevant metric for the performance of HPCG. We study the performance of the used architecture for four different DDR-SDRAM design points, ranging from the default DDR2400 as shows in Table 6.1 to DDR3200, which is the fastest traditional DRAM standard currently realizable [216], while running

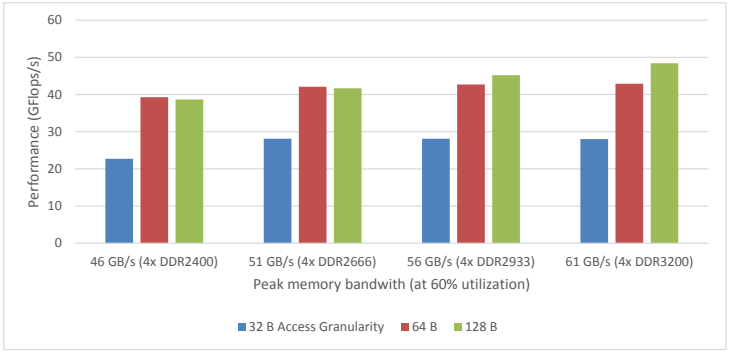


Figure 6.5: Impact of the access granularity and peak bandwidth for the HPCG benchmark.

a 64^3 problem. Utilizing more bandwidth requires more outstanding accesses, and as we do not change the NDP core design point, bandwidth might be left unused. Therefore, we also study the access granularity, ranging from 32 bytes up to 128 bytes. For this experiment we use a 4 KB cache per core, with cache lines as wide as the access granularity. In Figure 6.5, the performance is shown for the various design points. Using 32 B accesses, the available bandwidth can not be utilized and the performance saturates already using DDR2600 memory. The results when using 64 B and 128 B accesses are alike for the slower memories; only for the fastest DDR3200 standard the 128 B accesses show a clear improvement of around 10%.

As we are working with a shared memory architecture, the inter-NDP communication is also affected by the access granularity. The data access pattern for the remote reads shows a high spatial locality, and larger access granularities are therefore beneficial. When increasing the access width from 32 B to 64 B, we see an expected drop of a factor two in the number of remote data accesses done by the NDP. At the NDP-AP, the data cache hit rate changes from 75% to 50%, explained by the fact that the NDP-AP always issues 128 B wide accesses in the global coherent space. The overall communication time drops however only with 10%. This minor improvement can be explained by the fact that the amount of data transmitted is very limited in size: over the entire run of the application, every NDP reads only 2 MB of remote data, which is not enough to fully exploit the interconnect characteristics.

SOFTWARE PREFETCHING

As the NDP cores or the memory system do not employ any form of hardware prefetching or out-of-order processing, software prefetching can improve performance by hiding the memory latency. For the *Dot* and *WAXPBY* kernels software prefetching is a trivial task, since they both loop over data in a linear way. For both the *SpMV* and *SymGS* kernels, prefetching is more complicated since we have to deal with an indirect data lookup, where the needed data directly depends on the previous load. Let us denote this situation as *A[b[i]]*, where the data items needed from matrix *A*, depend on the data items loaded from vector *b*. In this case *b* can be prefetched perfectly and efficiently, but the indices found in *b* follow the diagonal structure of the matrix *A*, which shows spatial locality

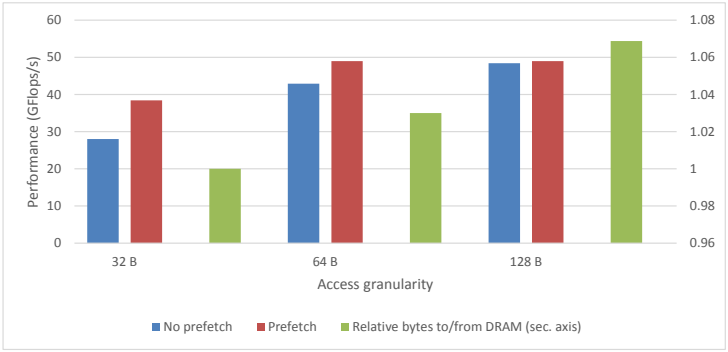


Figure 6.6: Impact of software prefetching for the HPCG benchmark, while also varying the access granularity.

for some accesses, but not for others. As the benchmark description forbids the exploitation of knowledge about the exact data layout, only generic implementations are possible. We investigate the impact of prefetching on the achieved performance for the three access granularities, using DDR3200 as memory standard. In Figure 6.6 the results are shown. When using either a 32 B or 64 B access granularity, the performance improves 35% and 15%, respectively. For the largest access granularity, software prefetching does not improve performance, as the bandwidth is already utilized by the non-speculative traffic. When using a 64 B access granularity with prefetching we can reach the same performance as in the case of the 128 B access granularity with prefetching. Also shown in Figure 6.6 is the relative amount of bytes loaded from, and stored in, memory. As can be observed, using larger access granularities results in more bytes accessed from memory, indicating that not all bytes loaded from memory are actually used. Since accessing memory, and transporting the data accordingly, costs energy, using a 64 B access granularity with software prefetching is preferred over using a 128 B access granularity, as the performance they achieve is the same. This holds when considering 128 B granularities including prefetching, as well as without prefetching.

6.4.3. CONCLUDING RESULTS AND COMPARISON

In Figure 6.7 we show the absolute performance achieved for the HPCG benchmark on the studied architecture, when varying both the problem size as well as the amount of NDPs used. We use all the software optimizations described, and the fastest architectural design point: a 4 KB cache per core, DDR3200 memory technology and a 64 B access granularity including software prefetching. When keeping the amount of NDPs constant, while increasing the problem size, the performance stays constant. The scaling from two, to four, and up to eight NDPs can be considered linear. In Figure 6.8 we show both the absolute and the relative amount of time spend in the inter-NDP communication phase. In can be observed that, when using eight NDPs, the communication time increases from one millisecond to 1.4 millisecond when increasing the global problem space from 128^3 to 208^3. The relative amount of time however drops from 3.3% to 0.9%, making the inter-NDP communication a negligible phase in the entire application. The high-bandwidth and low-latency hardware mechanism enabling communication between the NDPs, or-

chestrated by the NDP-AP, clearly makes the system appear as a single strong node for this application. This scaling behavior suggests that a dual-socket system will double the performance of the studied single-socket solution.

We use McPat to make a first order estimation about the power consumption of the proposed NDP. By filling in the architectural parameters, as well as the switching activity of the cores, the caches, and the memory controllers, when using a 22nm technology node, we obtain 8.6 W. If we exclude from the calculation the power evaluated by McPAT for the memory controllers (as the host CPU would have its own memory controllers), we obtain a power consumption equal to 6.1 W. When considering the upper bound of 8.6 W per NDP, combined with the NDP-M and NDP-AP power numbers discussed in Section 6.3.2, when considering eight memory channels, we obtain a total power budget of 88.8 W.

In Table 6.2, we compare the performance of our architecture to the same code running on commercial quality servers, as well as to results taken from literature. The POWER8 used in [44] uses four memory channels, so a comparison with four NDPs is in place, resulting in a performance improvement of a factor 2.5. Adding four NDPs to a single socket increases the system power with roughly 15% [197], thus realizing an energy-to-solution saving of a factor of two over a CPU-only system. The dual-socket, dual GPU system studied in [44] reaches a performance of 95.8 GFlops/s, or around 48 GFlops/s per socket, similar to the studied architecture when considering eight NDPs. Power numbers are not included in [44], but the TDP of the K80 is 300 W [217]. If we consider that the realized power usage of the K80 is 75% of the TDP, the proposed NDP solution is a factor 2.5 better regarding power efficiency at the device level.

The achieved performance per NDP is 6.25 GFlops/s, thus 100 MFlops/s per core. Within the compute kernels, the multiply-accumulate pattern shows a 1:1 ratio between multiplications and additions, for an average instruction latency of three cycles. Given the non-pipelined nature of the modeled cores, the theoretical peak would be 333 MFlops/s for the compute kernels, assuming that no other instructions are executed. The realized floating-point performance utilization is therefore 33%, which corresponds to the measured core activities. Including all instructions, the cores are waiting for data roughly half of the time. The K80 solution, with a peak double-precision floating point performance of 2.91 GFlops/s (5.82 GFlops/s for the entire system), runs at 6% utilization. The utilization of the K80 system roughly corresponds with the supercomputer utilization mentioned in Section 6.2.2, while the utilization of the studied NDP architecture is significantly higher.

6.5. GRAPH500 BENCHMARK

The Graph500 benchmark does 64 breadth-first searches (BFSs) starting from a random selected root in an undirected Kronecker graph. The output, for every root, is a predecessor list storing the predecessor of every vertex. The size of the problem is denoted as scale n, meaning that the graph contains 2^n vertices and 16 times as many edges. The graph is traversed in several steps, called levels. The first action performed is storing the *root* vertex in a data structure called the *frontier*, representing the edge of the current exploration level. In a traditional top-down exploration, as provided by the reference code, for every level, every vertex in the *frontier* has all its neighbors traversed and when a neighbor is

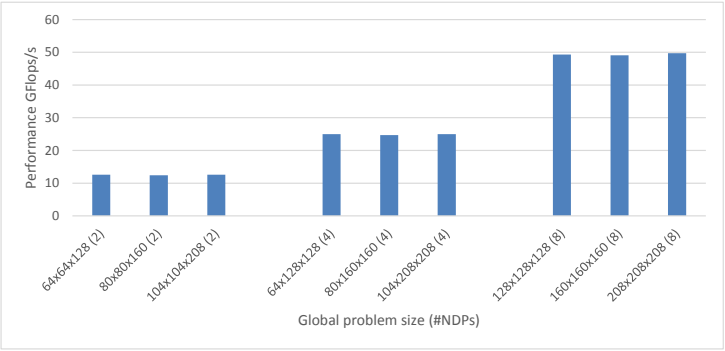


Figure 6.7: Performance scaling for the HPCG benchmark, using various problems sizes and number of NDPs.

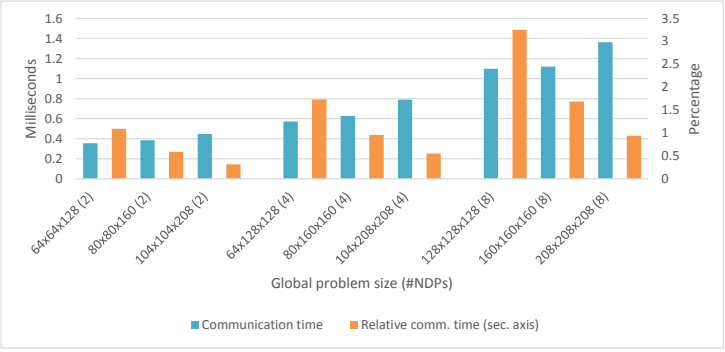


Figure 6.8: Absolute and relative communication time for the HPCG benchmark, using various problems sizes and number of NDPs.

Table 6.2: Comparison of HPCG performance, in GFlops/s

System	Problem size	Perf.
1-Socket 4 NDPs	104x208x208	25
1-Socket 8 NDPs	208x208x208	49
1-Socket POWER8 10c, 4.2 Ghz, 4 memory channels (our code)	208x208x208	9.66
2-Socket Xeon E5-2697v3 [208]	384x192x192	18.4
2-Socket POWER8 [44]	Unknown	21.14
2-Socket POWER8 + 2x K80 [44]	Unknown	95.85

not visited yet, it is claimed as child and put in the *out* queue. At the end of a level the *out* set becomes the next *frontier*, and this is repeated until no new vertices are claimed anymore. Because the benchmark implements a distributed BFS, some neighbor can be stored at other computing nodes, requiring, e.g. messages to be sent, depending on the exact implementation and platform used. Since there are no floating-point operations in this workload, the performance is measured in Giga Traversed Edges Per Second, or GTEPS. A detailed explanation of distributed Graph500 can be found in [193].

6.5.1. IMPLEMENTATION AND BASELINE OPTIMIZATIONS

In our implementation, we use a compressed sparse-row format to distribute the graph data set among the NDPs. As with HPCG, the MPI + OpenMP parallelization is replaced by a nested OpenMP parallelization. All BFS related code is executed in the simulation environment. The benchmark has little strict requirements regarding processing and data storage, which opens up possibilities for significant optimizations in data layout and graph traversal approaches [194] [218] [210] [219]. A key BFS optimization, called 'direction optimization', implements the switch between bottom-up and top-down frontier expansions [195]. Depending on two parameters α and β and on the current state of the exploration, the optimization picks for every level the use of the top-down approach described earlier, or a bottom-up approach, where all not-visited vertices search for a parent in the current *frontier*. Since results without this optimization can be considered irrelevant today, we added this optimization to the baseline implementation of our code.

Another optimization added to our baseline implementation is described in [210], proposing to sort the adjacency list of every vertex by out-degree (number of neighbors). In this way, in the bottom-up expansions, for every unvisited vertex, first the most relevant vertices in the graph are checked, which have a higher probability of being visited before and thus being a valid parent. Discussed in the same work, and also added to our baseline implementation, is a method to sort the vertices by out-degree. Since about half of the vertices have no neighbors at all, this allows us to skip large portions of vertices in the bottom-up explorations. Both optimizations improve the performance significantly, foremost thanks to an improved data locality. A last optimization added to the baseline, which also increases the data locality, stores the index of every vertex' heaviest neighbor in a separate data set, to be accessed without accessing the entire adjacency matrix.

In the remainder of this section we discuss several explorations of the architectural

parameters, as well as software optimizations. As an architectural baseline, unless stated otherwise, we consider 4096 KB NDP caches, a 64 B access granularity, and DDR3200 memory technology. Unless stated otherwise, we use a scale 25 problem and eight NDPs.

6.5.2. OPTIMIZATIONS

TWEAKING THE DIRECTION OPTIMIZATION

As stated, the baseline version implements the 'direction optimization', switching between bottom up and top down frontier expansion [195]. In [195], optimal α and β for a CPU architecture are determined as 14 and 24, but on the studied architecture an α value of 50 was found to perform 6.5% better. In practice, this means that the third level is done almost always with a bottom up exploration, while the original α results in a somewhat even distribution between top down and bottom up explorations.

In a top-down exploration, all neighboring vertices are visited, and many of these vertices will be stored in different memory channels. This is a process hard to optimize and with little data locality, while the bottom-up exploration can be optimized much more easily, as will be shown in the remainder of this section. When using a top-down exploration, 50% of the accesses is remote, with an NDP-AP cache hit rate of 53%. When using the bottom-up approach, these percentages are 74% and 73%, respectively, clearly indicating that this type of exploration runs more efficiently on the architecture.

UTILIZING LOCALITY FOR REMOTE ACCESSES AND SELECTIVE BITMAP-SUMMARY USAGE

As described in Section 6.3.1 accessing remote data is done by means of the NDP-AP. The NDP-AP has a data cache, which allows it to cache data requested by NDPs, and serve them quicker in case of a data cache hit. For HPCG this was relevant for the communication phase, where every one out of two 64 B remote accesses resulted in a hit, since the NDP-AP works with a 128 B access granularity.

For the Graph500 benchmark, the most relevant remote accesses are in the bottom-up explorations, where the vertex under study checks whether a neighbor is in the *frontier*, which is stored distributed over all the nodes. This data set is typically stored as a bitmap, where every bit represents a single vertex, to improve cacheability. To improve cacheability even further, a *summary* of the frontier can be used, where every bit represents the logical *OR* of 64 bits in the frontier. As this summary bitmap has a 64 times smaller memory footprint, it can be cached more easily. However, using it only increases performance if the summary bit is *0*. In case of a *1*, the full bitmap still has to be checked, eliminating the data locality advantage. On a typical CPU architecture with large caches and out of order processing this is an easy optimization, but on the studied architecture the performance only increased significantly by using the summary for a selection of graph explorations, based on the following rule:

$$use\_summary = \frac{n_g}{n_f} > \phi$$

In which n_g is the amount of vertices in the graph, and n_f is the amount of vertices in the frontier. We only want to use the summary bitmap if we have a small amount of vertices in the *frontier*, meaning the summary bitmap is likely to contain a *0*. In case of a scale 25 problem, a ϕ of 650 was found to provide good performance, while for a

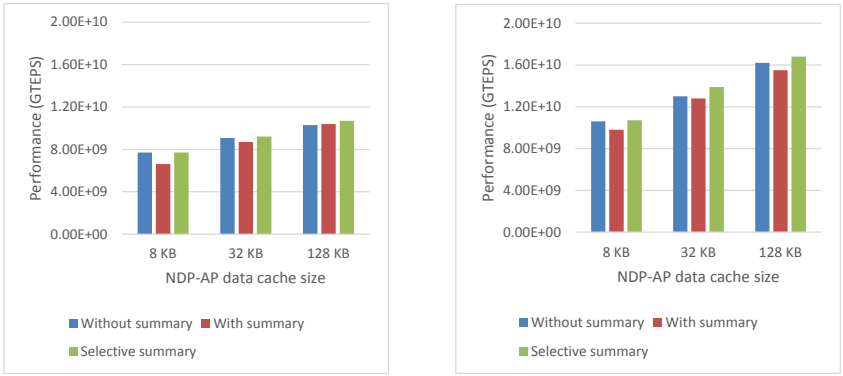


Figure 6.9: Impact of the NDP-AP data cache size on Graph500 performance, including the (selective) use of bitmap summaries.

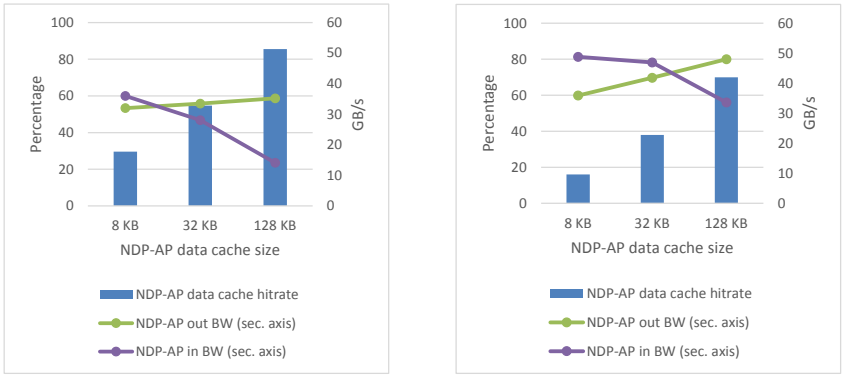
scale 23 problem, 1400 is optimal, using the summary more often. This can be explained by the fact that a larger problem with semi-random accesses will benefit more from an optimization improving locality.

In Figure 6.9a and Figure 6.9b we shown the impact of the NDP-AP data cache size when considering eight NDPs, and a scale 23 and scale 25 problem, respectively. In both figures the performance with, without, and with selective frontier bitmap summary usage, is shown. It can be observed that the size of the data cache at the NDP-AP has a significant impact on performance. Increasing the size from 8 KB to 128 KB improves the performance of 39% and 57%, for the scale 23 and scale 25 problem, respectively. The selective use of the summary improves the performance in all cases, showing a highest speedup of 7% when considering a scale 25 problem and a NDP-AP data cache of 32 KB The average increase in NDP-AP data cache hit rate is in the order of 5%.

In Figure 6.10a and Figure 6.10b we show the NDP-AP data cache hit rate for the same problems as discussed above, with selective summary usage. It can be observed that the hit rate considering a scale 25 problem is much lower compared to a scale 23 problem. Also visible is the significant increase in hit rate of 55% when considering a scale 25 problem, when going from the smallest to the largest data cache size. Figure 6.10a and Figure 6.10b also show the measured input and output bandwidth of the NDP-AP. It can be observed that these measured bandwidths show an opposite trend. With the data cache hit rate increasing, less data needs to be transported from the system into the NDP-AP, thus we see the bandwidth reducing. Due to the higher hit rate, the runtime reduces, but the amount of outgoing data stays the same, resulting in a higher outgoing bandwidth.

NDP DATA CACHE SIZE IMPACT

Similarly to the HPCG benchmark, we evaluate the impact of the NDP data cache size, when considering a scale 25 problem and eight NDPs. When increasing the data cache size from 512 B to 4 KB per NDP core, the performance increases by 25%. In both cases,



(a) (b)

Figure 6.10: Impact of the NDP-AP data cache size on the NDP-AP data cache hit rate and NDP-AP incoming and outgoing bandwidths.

the realized bandwidth to main memory is around 8 GB/s per NDP, where the requirement for more data is apparently offset by the extra runtime.

GLOBAL ATOMICS

As discussed in Section 6.3.1, the NDP-AP supports the execution of atomic operations. This is relevant in the top-down levels because checking whether a node has been visited, and then setting it as visited, needs to happen atomically, and it is often performed on remote data. About every 40 cycles an atomic operation is issued to the NDP-AP. In the baseline, 2.5% of these operations encounter a lock and the average waiting time is over 500 ns, most likely so high because the adjacency lists are sorted. When the NDP-AP supports atomics, this waiting time is removed. The amount of writes issued to the NDP-AP drops by 16%, saving some DMI bandwidth. However, it has no significant impact on performance (< 1%), as the amount of atomic operations hitting a lock is too low, and the exploration switches to the atomic-free bottom-up type relatively fast.

USER ENHANCED COHERENCE

The studied architecture implements the possibility to enable the cacheability of non-local data in the NDP caches, at the expense of manual handling coherence. This optimization eliminates a part of the remote accesses, and can thereby improve performance. When studying a scale 25 problem with eight NDPs, we see a reduction of a factor three in the amount of remote reads processed by the NDP-AP. The remaining remote accesses achieve a 10% lower hitrate at the NDP-AP data cache, since the remote accesses able to exploit locality are already served by the NDP caches. The performance increases by 13%, and the flushing of the NDP caches at the right moments does not influence the performance in any measurable way.

SOFTWARE PREFETCHING

We studied the use of software prefetching to improve performance. As the bottom-up explorations take up most of the time, we focus our effort on that kernel. It turned out

that prefetching the data set items accessed in a semi-random way, offered no performance improvements. The baseline data-locality and graph restructuring optimizations discussed earlier make sure most of these accesses hit in the cache. The best performance was achieved when only prefetching the data set used to determine which vertices are still not visited, which is accessed in a linear fashion in the upper most loop. Measured performance improvements are between 15% and 22%, when evaluating a set of three problem sizes (24, 25, 26) and three system configurations (2, 4, 8 NDPs). It is an insightful result that the performance improves significantly by reducing the pressure of streaming loads, while the semi-random loads are performing well on their own.

6.5.3. CONCLUDING RESULTS AND COMPARISON

In Figure 6.11 the performance of various systems and problem sizes is shown. For this experiment we use all optimizations described above, and a system with 4 KB NDP data caches, a 128 KB NDP-AP data cache, and DDR3200 memory. We observe a declining performance trend going towards larger problems, due to less data locality at both the local and global level. Doubling the amount of NDPs from two to four, while keeping the problem size constant, results in a *strong* scaling factor between 1.69 and 1.71, while doubling from four NDPs to eight NDPs results in a *strong* scaling factor between 1.38 and 1.52. This less-than-perfect scaling is due to the increasing amount of non-local accesses, as well as the decreasing amount of work per NDP. Doubling the amount of NDPs, while keeping the work per NDP constant, results in a *weak* scaling factor between 1.13 and 1.53.

Graph processing has been described as the killer application for near-data processing [192]. With Graph500 being *the* graph processing benchmark, valuable insights should become apparent. As described, many optimizations for Graph500 exist, focusing both on the algorithmic level (direction optimization), as well as the data locality level (sorting the vertices and edges, optimized data structures). As it turns out, with all optimizations added to the baseline of our code, the most typical near-data processing advantage of having a high bandwidth to local memory is not a crucial system parameter anymore. When considering eight NDPs and a scale 25 problem, we reach an accumulated local memory bandwidth of 100 GB/s, with an accumulated inter-NDP bandwidth is 80 GB/s. When considering two NDPs and a scale 25 problem, we reach an accumulated local memory bandwidth of 34 GB/s, with an accumulated inter-NDP bandwidth is 21 GB/s. This highlights that a high-bandwidth interconnect between the NDPs is equally important compared to local memory bandwidth, when considering optimized graph explorations. When exploring a scale 26 problem, the local memory bandwidth increases with 10%. It is stated in [210] that the optimizations presented in that work are foremost effective in the small-world Kronecker graphs as used in Graph500. This highlights that there is an interesting tradeoff between tedious software optimizations regarding data locality for specific graph applications, and running generic code on architectures offering high bandwidths.

Table 6.3 compares the performance of the studied architecture with the performance of high-end servers running our code and several results from related approaches. The provided references to existing research achieve around 10 GTEPS per socket, which is significantly higher than our code running on a POWER8 CPU, most likely due to addi-

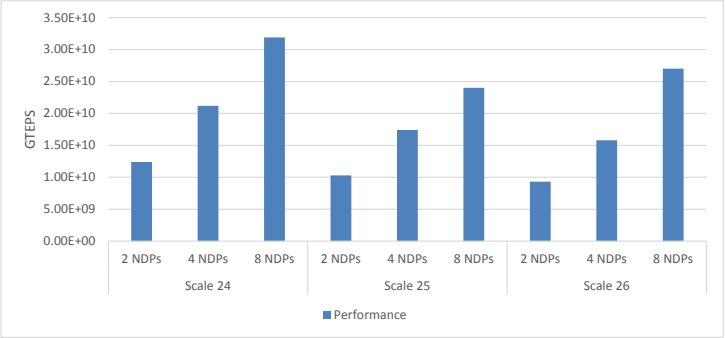


Figure 6.11: Performance for the Graph500 benchmark, varying the problem size as well as the amount of NDPs.

Table 6.3: Comparison of Graph500 performance, in GTEPS

System	Problem size	Perf.
1-Socket 4 NDPs	25	17.4
1-Socket 4 NDPs	26	15.8
1-Socket 8 NDPs	25	24
1-Socket 8 NDPs	26	27
1-Socket POWER8 10c, 4.2 Ghz, 4 memory channels (our code)	25	4.04
4-Socket Xeon E5-4640 [210]	27	42
1-Socket Xeon E5-4640 [219]	25	11.16

tional optimizations in the data layout, as the kernels themselves are too simple to offer optimizations able to achieve such a speedup. It can be observed that, both four and eight NDPs are able to outperform a CPU for equally large and even larger problems.

To evaluate power and energy efficiency, we use the same methodology as with the HPCG benchmark. First we evaluate the NDP solution against our code running on a POWER8 CPU, having four memory channels. For this experiment, we modified the code to better fit the single CPU, by for example removing the notion of having multiple compute nodes. Adding four NDPs to a single socket increases the system power with roughly 15% [197], and increases the performance with a factor 3.5x. This results in a factor 3x energy-to-solution saving. Comparing with the work in [210] and [219] is non-trivial, as their baseline performance is much higher than the performance achievable with our code. When comparing device to device, and setting the power usage of the Xeon E5-4640 at 71 W (75% of its TDP), four NDPs are 85% more power efficient compared to the Xeon and its optimized software implementation. Comparisons with GPUs are unfortunately not possible, since the problem sizes explored in this work are much larger than what would fit in GPU memory.

6.6. CONCLUSIONS

In this work we have presented a near-data processing architecture, were the NDP-cores all reside in a coherent shared-memory space, and showed the performance of two important industry-standard benchmarks running on this platform. For the HPCG benchmark we identified the optimal data caches sizes, access granularity, and software prefetching strategy. By having a bandwidth oriented architecture, with high-bandwidth low-latency inter-NDP communication mechanisms, this benchmarks runs 3.5 times faster compared to a CPU, and with a factor 5x higher utilization compared to a GPU. For the Graph500 benchmark, the exploitation of locality for inter-NDP traffic is a key characteristic for application performance, as well as the cacheability of remote data by means of user-enhanced coherence. After various complex optimizations, the performance of this optimized graph-traversal benchmark depends equally well on a fast interconnect as local memory bandwidth, resulting in a 3.5x speedup with respect to a CPU. While current supercomputers are being optimized for LINPACK, we have shown how novel node designs could optimize them for HPCG and Graph500, resulting in higher a utilization and higher performance for those benchmarks and the applications they represent.

7

SORTING BIG DATA ON HETEROGENEOUS NEAR-DATA PROCESSING SYSTEMS

Heterogeneous systems are about having multiple different types of processing devices within the same computer system. There are various ways of making use of this property. Applications can be executed as a whole on one of the type of devices, picking the type which fits its overall characteristics best. Another options is to have parts of an application on one device, and other parts of the applications on the other device. This is the way the popular GPU coprocessors are used: the CPU runs the complete application, and makes use of the compute power of the GPU for certain kernels. Last, we can make use of heterogeneity within a single kernel, using a fine-grained collaboration between the devices.

The last method is explored in this chapter. With a system based on a CPU and various workload-optimized NDPs, we explore the sorting of large data sets. Sorting has, by nature, phases with much data locality, as well as phases with little data locality. CPUs, with their large caches, excel at the phases with data locality, while the NDPs, with their high-bandwidth access to main memory, excel in the phases with little data locality.

By making use of both the CPU and the NDPs in a heterogeneous fashion, we do not only improve performance for the application under study, but also reduce data movement dramatically. With data movement being one of the big energy drains in a computer system, this is a key benefit. By making use of workload-optimized NDPs, we push the performance and power-efficiency of the NDP itself as well.

The content of this chapter is based on the following paper:

Sorting big data on heterogeneous near-data processing systems
E. Vermij, L. Fiorin, C. Hagleitner, and K. Bertels
BigDAW17, big data analytics workshop, co-located with Computing Frontiers 2017

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.

7.1. INTRODUCTION

The efficient analysis of big data workloads represents a key element in many businesses and scientific and engineering applications [167]. 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 [220].

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 [167].

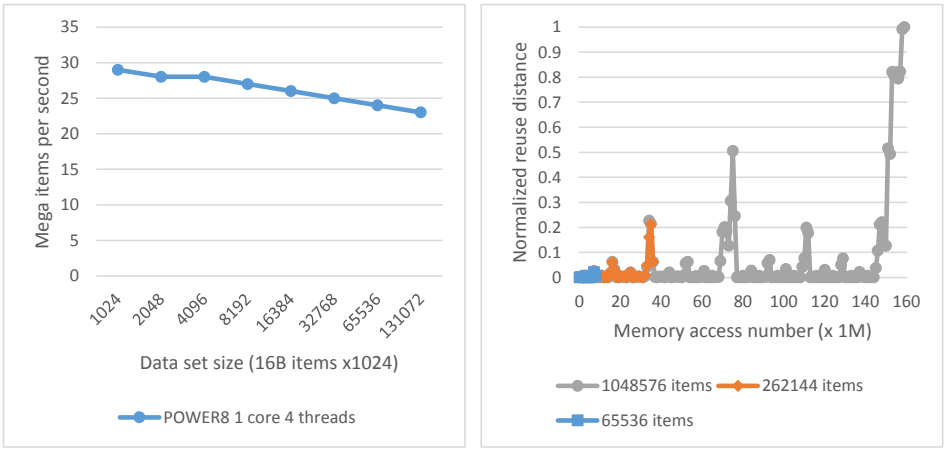
This work analyzes the sorting of big data workloads on near-data processing architectures [171]. This computing paradigm, recently rediscovered, allows to alleviate the classical "memory wall problem" by moving the computation closer to the memory. For certain types of workload which typically do not 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 [221].

In particular, 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 7.2 presents background information and related work. Section 7.3 discusses the reference architecture and its system simulator used in our experiments. Section 7.4 discusses the optimized implementation of the sort algorithm on the heterogeneous architecture. Section 7.5 presents an analysis of the experimental results, while Section 7.6 concludes the paper.

7.2. 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 sockets 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,



(a) (b)

Figure 7.1: (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.

and then repeatedly merge two sublists at the time to produce new sorted sublists, until only one sorted list remains.

Figure 7.1 shows the results of our preliminary evaluation for the case of 4 threads per core, in which we operate on 16 B items, consisting of an 8 B index (to be sorted), and an 8 B pointer. As Figure 7.1a shows, the performance decreases with the size of the data set. This behavior can be explained by analyzing Figure 7.1b, 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 7.1b 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 7.1a.

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.

7.2.1. OVERVIEW OF RELATED WORK

Sorting unstructured items represents an important task for big data applications [167]. 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 [167]. A custom system and board design is proposed in [76]. 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 [220], or a cluster of FPGAs [222]. 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 [223], 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 [73]. 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) [35], by adding small general purpose cores or dedicated accelerators to it [88, 91, 98, 144, 169], 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 [98], while the work in [88] discusses an architecture which targets the execution of big data analytic frameworks near the memory. In [174], a sorting accelerator is implemented as part of a 3D-stacked memory system. 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 [213], 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 [213] 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.

7.3. NEAR-DATA ARCHITECTURE

A high-level view of the reference system architecture is shown in Figure 7.2 [213]. 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 [34], which has eight memory-technology agnostic memory channels each connecting to a 'memory

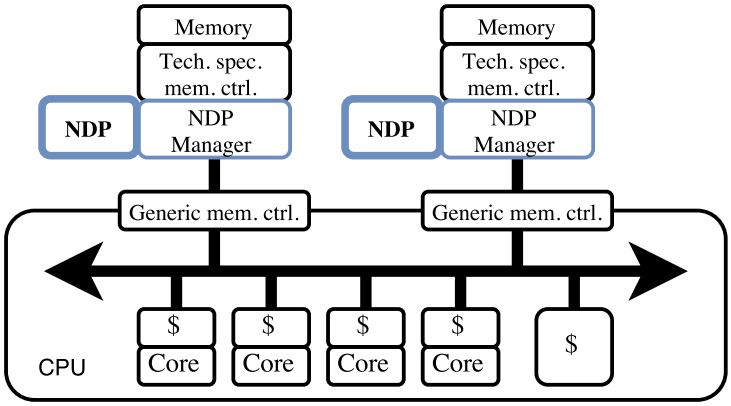


Figure 7.2: High-level view of the system organization. Near-data processors are tightly integrated with the memory controllers.

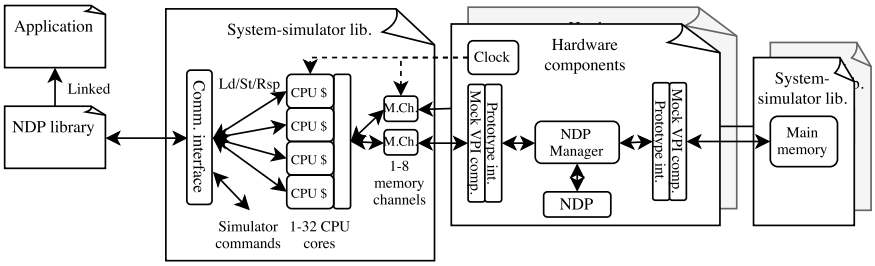


Figure 7.3: The simulator environment used in our experiments.

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 [35], in which technology-specific memory controllers (called *vault controllers*) are implemented in the logic layer of the device.

As shown in Figure 7.2, near-data processing capabilities are added in the technology-specific memory controllers. Each NDP relies on a hardware component, called NDP-Manager (NDP-M) [213], 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 [51]. 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 7.1 shows specification for the reference system considered in this work. The system template we use in the 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 7.1 is smaller than the one of a real CPU. However, we scaled appropriately simulation results to take into account for this limitation, by evalu-

Table 7.1: 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

ating 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 7.3 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

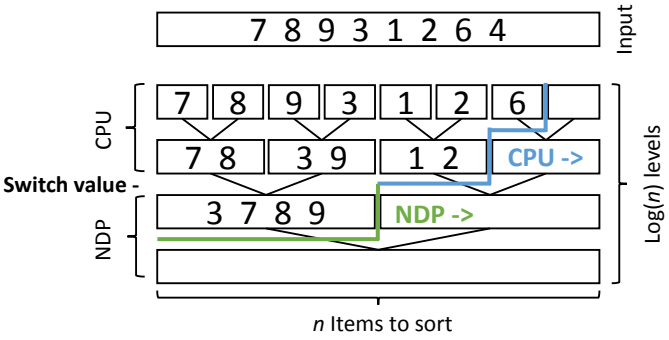


Figure 7.4: Implementation of the heterogeneous (CPU + NDP) mergesort.

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.

7.4. 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 7.4. 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 7.4, the first iterations of the algorithm offer plenty of small merges that can be easily parallelized 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 [224]. 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 7.5 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

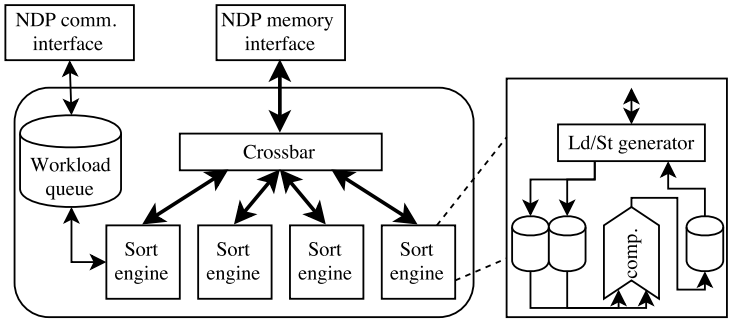


Figure 7.5: Block diagram of the workload-optimized merge core implemented in the NDP.

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 7.3, 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 7.1 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 7.4, 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 7.3, 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.

7.5. ANALYSIS AND RESULTS

7.5.1. 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:

$$\frac{log_2(n)}{log_2(Max\_switch)} \leq Speedup \leq \frac{log_2(n)}{log_2(Min\_switch)} \tag{7.1}$$

in which n is the data set size. For a data set size of one mega items and the parameters in Table 7.1, 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:

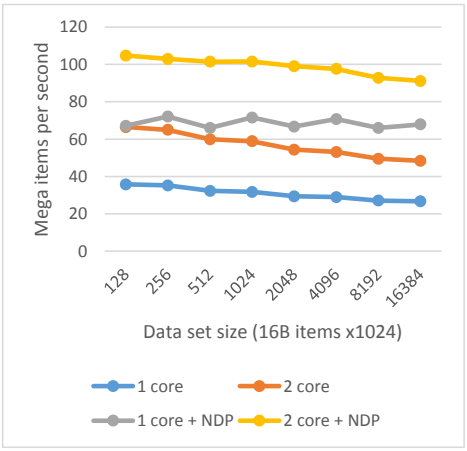
$$\lim_{n\to\infty} Speedup = BW_{NDP}/BW_{Memory\_channel} \tag{7.2}$$

When using the values shown in Table 7.1, the speedup is approximately equal to 1.5.

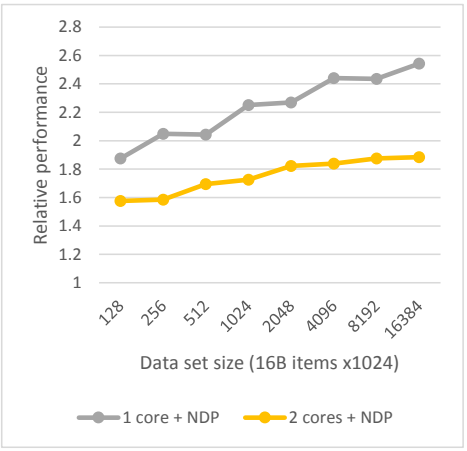
7.5.2. RESULTS

In our experiments, we evaluate the performance of heterogeneous systems with different combinations of CPU cores and NDPs. Figure 7.6a 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 7.6b 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 7.6b, up to a factor 2.5 for the largest data set, as estimated in Section 7.5.1. The measured speedup is, in fact, within 5% difference



(a)



(b)

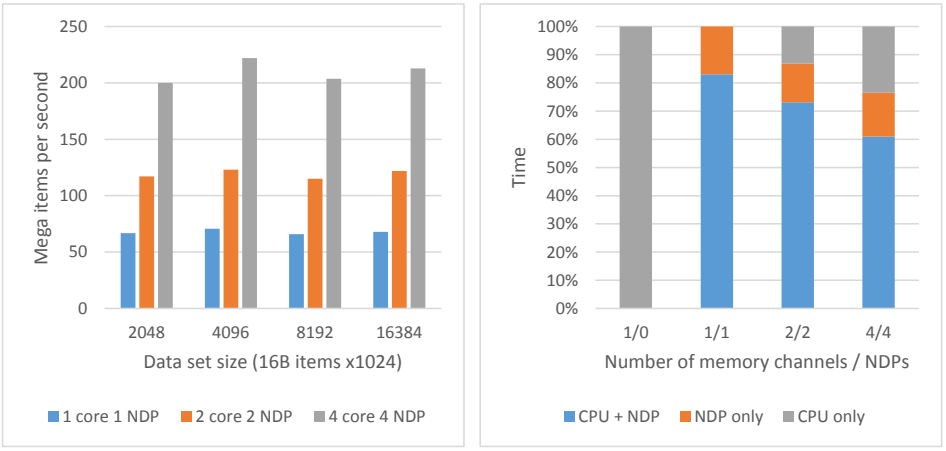
Figure 7.6: (a) Absolute mergesort performance for four different systems configurations. (b) Relative performance with respect to configuration with no NDPs.

from the average speedup calculated considering the smallest and highest speedup values estimated in equation 7.1. 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 7.7a, 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 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.

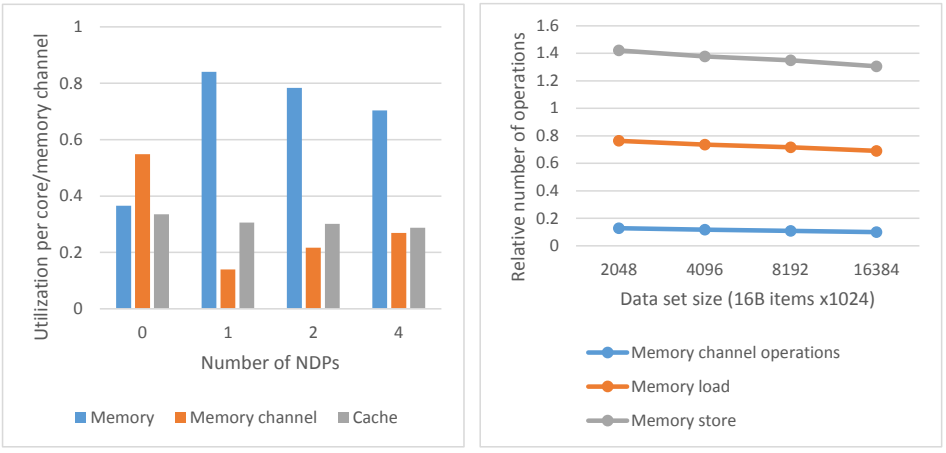
In Figure 7.7b, we show the portion of time in which both components (CPU + NDP) are working, and the time in which only one of them is active. It is clear that, when using more NDPs, a larger portion of time is spend in the final merge steps done by the CPU, which directly results in the above mentioned not perfect scaling.

In Figure 7.8a, 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 7.8b 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



(a) (b)

Figure 7.7: (a) Performance for systems containing more memory channels and NDPs. (b) Time distribution of the heterogeneous usage of both the CPU and the NDP.



(a) (b)

Figure 7.8: (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 configuration.

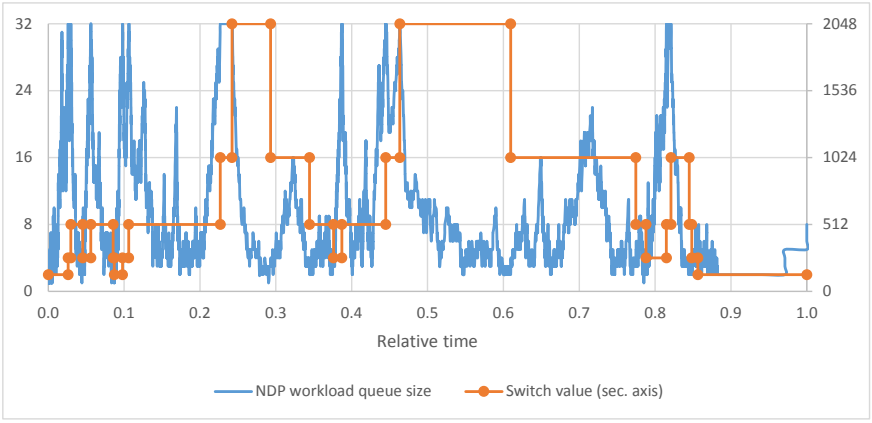


Figure 7.9: NDP work queue size and switch value over time, when sorting four mega items using two CPU cores and a single NDP.

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 7.9 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 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.

7.5.3. 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* [225] 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 [197]. 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 [197] [198]. In such a system, our proposal would add 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.

7.6. 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 a 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.

8

FROM NEAR-DATA PROCESSING TO DATA-CENTRIC SYSTEMS

Near-data processing is a term used to fuel a wide variety of research topics. All these research approaches have a goal in common: they focus architectural (hardware / software) improvements to improve data-related aspects of computer systems. In this chapter we take a look at where the preceding chapters have taken us, and how future steps at the hardware level as well as the software level can look.

8.1. INTRODUCTION

In the preceding chapters we presented an architecture to enable near-data processing in a modern CPU architecture, including methods to manage data locality between memory channels. In this case, and in nearly all related near-data processing research, the NDP is still a slave of the CPU. Looking towards the long-term future, a more dramatic reorganization of computer systems might be necessary. Key questions are the attachment of components to the CPU and/or their placement in a system, and the data-locality management in complex heterogeneous systems composed of many devices. Specific aspects of existing POWER processors and systems, as well as some announced technologies, give us hints as to what is feasible, but this chapter should be primarily read as a discussion of a different and more generic NDP design point, with the objective to motivate further study.

8.2. SERIAL ATTACHED MEMORY AS KEY NDP-ENABLER

In [98] it is stated that the key enabler for near-data processing is 3D stacking technology, and in [103] it is stated to be the reason for the recent 'reinvention' of the topic. Given the large amount of research targeting stacked technologies like the Hybrid Memory Cube (HMC), the same is implied in many other works. In this work we do not make assumptions on the type of memory technology used, or the exact physical realization. What is assumed in this work, is a serial, memory-technology agnostic, connection between the CPU and a 'hub component', containing the NDP-M, the NDP, and the technology-specific memory controllers. As shown in the taxonomy in Figure 1.14, the memory integration can be 3D stacked, 2.5D stacked, or 'traditionally' attached memory. From an architectural perspective, the memory attachment is irrelevant: the key enabler of the near-data processing concept is the memory-technology agnostic serial link towards a hub component. Stacking can improve energy efficiency, but in case of the HMC does not necessarily improve the memory bandwidth [35]. In fact, as discussed in Section 1.6.3, stacking has disadvantages as well, for example the fixed device-storage ratio.

The HMC combines both a novel high-speed serial link with 3D stacking, but the concept of serial attached memory is much older. Both the POWER7 (2010) [185] and POWER8 (2013) [34] CPUs use serial attached memory with a hub component ('memory buffer chip'), and this concept has been used as a baseline throughout this work. Figure 8.1a shows this concept next to direct attached memory. The missing piece to enable near-data processing has been the system-integration, addressed in this work.

8.2.1. UPGRADING THE LINK PROTOCOL

In this work a near-data processing enabling framework *on top* of an existing load/store serial interface is proposed by means of the NDP-M and NDP-AP component, as shown in Figure 8.1b. This allows the integration features necessary, without impacting the original functionality of the link. In the announced POWER9 CPU [40], the pure load/store serial interface is augmented by the more capable OpenCAPI protocol [45]. Planned features, for future versions of OpenCAPI, are the enabling of sharing of near-data processor ('attached device') memory with the CPU, conceptually similar to what is proposed is this work (Figure 4.3b). This is arguably a cleaner solution than the one proposed in this

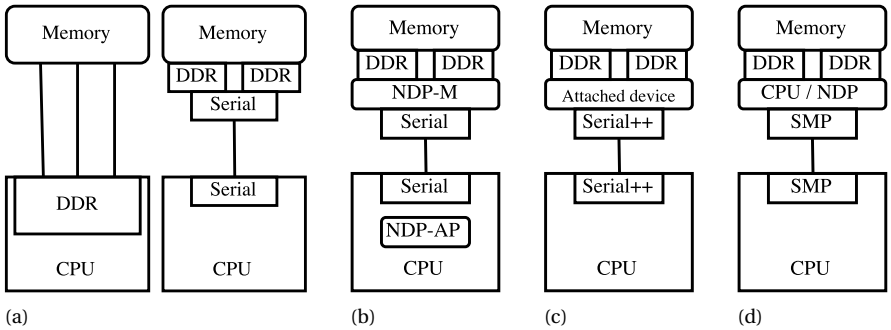


Figure 8.1: Evolution of memory attachment technologies. (a) Shows both traditional direct-attached DDR memory, as well as serial attached DDR as found in the POWER7 and POWER8 CPUs. (b) Shows the NDP-integration strategy proposed in this work, using the existing load/store serial interface. (c) Shows future serial-link protocols enabling the attachment of devices without the need of extra components [45]. (d) Shows the far end of the spectrum, attaching devices / NDPs over an SMP link, and making them equal to the CPU.

work, but will introduce extra latency for normal load/store traffic, impacting the CPU performance. This concept is shown in Figure 8.1c.

A next step in the evolution of the serial link between the CPU and the hub component containing the NDP, is making it a Symmetric Multi-Processor (SMP) link, and thus making the NDP a node in an SMP domain, able to run the operating system. This is shown in Figure 8.1d, and discussed in the next section.

8.3. NDPs AS FULL CPU PEERS

By upgrading the link between the CPU and the NDP to an SMP link, the NDP can become a full CPU peer, optimized for specific data-intensive tasks, as shown in Figure 8.2. Here NDPs occupy a place in the system similar to the CPU, or even replace the CPU, and are considered first-class citizens. The GPU will, at least for this generation of products, still operate as a discrete device, and thus a slave of the CPU. The concept of having NDPs as CPU peer is already introduced in the NDP taxonomy described in Section 1.6.3 and shown in Figure 1.14. It can be understood as the far end of a spectrum, or the 'most generic architecture to achieve the near-data processing goals'. Specialized hardware components such as the NDP-Manager and NDP Access Point, as discussed in Chapter 4, are no longer needed. Neither do we need an NDP software-library for managing the NDPs etc. Also important features such as coherence are resolved in a generic way. When a CPU and an NDP are each others equals, every node manages coherence in its own (optimized) way, and inter-node coherence is solved by means of the SMP fabric. Existing optimizations in the SMP fabric will limit the coherence interactions [34]. Novel rules for coherence, e.g. the user-enhance rules as discussed in Chapter 4 and Chapter 7, still apply as interesting research topics, to optimize both the local as well as the inter-node coherence interactions.

A variety of system designs are possible. The NDP can be used as a stand-alone solution, or can complement a CPU in a multi-node system. For the CPU it is possible to

have its own direct attached memory, but having memory via an NDP, as shown for the CPU in Figure 8.2, is possible as well. This is also the option that served as the baseline throughout this work, with the difference that the architecture in Chapter 4 enables near-data processing as an extension to, while in this chapter, *near-data processing enabled memory is attached to a CPU*. In case the CPU has its memory (or a part of its memory) attached by means of an NDP several performance aspects need discussing. An SMP link can offer very high bandwidth, since the connections can be realized with high-frequency serial links, suffering from less physical complexities compared to e.g. the parallel DRAM interfaces. Furthermore, DRAM technology is in general not optimized for peak performance, but for cost per bit. Specifically, it is more difficult to create a high-bandwidth interface on a (memory process based) memory chip than on a (logic process-based) CPU. The POWER8 CPU offers an aggregate SMP link data bandwidth of 312 GB/s, 50% higher than the peak memory bandwidth [34]. This total bandwidth is divided over six links, three for communication with the CPUs in the same node, and three for communicating with CPUs in different nodes, resulting in a system with a maximum size of 16 CPUs in four nodes. Besides bandwidth the other important performance metric is latency. In [214] the performance characteristics of a POWER8 SMP system is analyzed. It is shown that the latency to local memory reaches almost a 100 ns, and the latency to non-local memory (but within a node), is around 125 ns, both without prefetching. The penalty of crossing an SMP link is therefore in the order of 25 ns. With prefetching enabled, the average latency penalty in both cases drops to 10-20 nanoseconds, depending on the exact prefetching strategy. By studying the Graph500 benchmark on a dual socket POWER8 machine, using a single core in four-thread mode (to focus on latency and not bandwidth), we see a performance degradation of 15-20% when having all the data in the remote memory over having all the data in local memory. This paragraph shows that the attachment of memory to a CPU by means of an SMP link, instead of being directly attached, is certainly possible, especially when taking into account that memory latency sensitive applications should now run on the NDPs, and not on the CPU.

8.4. THREAD AND DATA-LOCALITY MANAGEMENT AS KEY NDP-TECHNOLOGY

In case the NDP behaves as a CPU, processes and threads can be scheduled at either a CPU or an NDP at the OS' liking. In case the system consists of both CPUs and NDPs, possible ISA differences have to be solved by means of e.g. multi-ISA binaries [226]. Data allocations, with or without explicit NUMA affinities, work as on existing multi-CPU systems. In NUMA systems, especially heterogeneous ones, managing data locality is key for good performance. Manual methods, based on *pthread* or *numactl* functionality exist, but also automated data-locality management is an active research topic. Work in [227] presents an overview of several frameworks helping the programmer with data-compute affinity. An example is [228], proposing both compiler and runtime methods to manage data locality, providing up to 4x improvements with respect to static heuristics. A runtime system based on OpenMP managing data locality is OmpSS [229]. Their methods focus on asynchronous task executing based on data-dependency graphs and

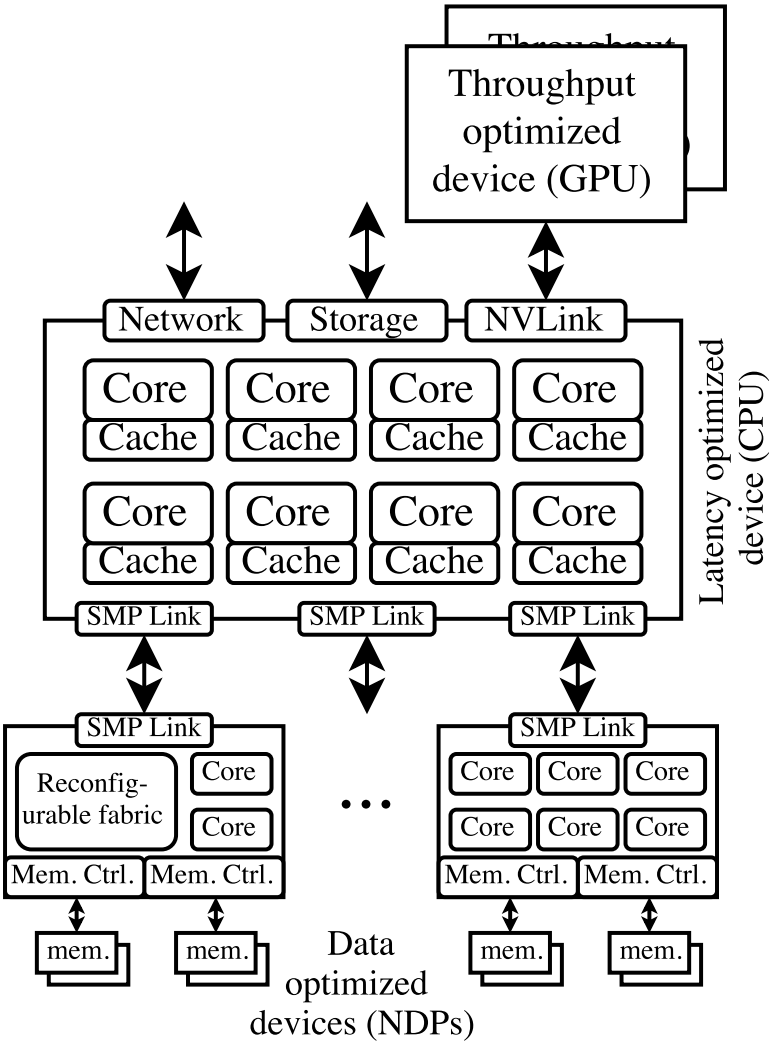


Figure 8.2: Placing NDPs as full CPU-peers in a system.

optimal data-compute affinity in heterogeneous systems.

The IBM AIX operating system [230], targeting large (up to) 16 socket SMP systems, has several interesting features that can be exploited and developed further to aid successful NDP exploitation in this setting. On top of the base kernel, the Dynamic System Optimizer [231] (DSO) offers autonomous performance improvements in complex systems. For example, threads working on the same data are migrated to (virtual) cores close together to optimize for cache affinity. Furthermore, DSO supports page migration between SMP domains. If a private page is accessed from a remote domain for a prolonged period of time, the page is migrated to that domain, to optimize access bandwidth and latency. This allows for data locality optimizations in a completely transparent way, *without needing knowledge of the workload(s) running on the system.*

Technologies like this will be key in the successful exploitation of near-data processing, especially when used in complex heterogeneous systems. It is unrealistic to assume to OS will develop in such a way that it becomes completely aware of all characteristics and features of various memories and devices. As described in Section 4.3, when discussing NDP specific memory allocations, the OS has at this moment not even a basic understanding of memory controllers, thus OS-level optimizations to match a data-intensive application to the right high-bandwidth memory controller will not be deployed any time soon. The use of an intermediate software layer that is aware of all the system details, such as the AIX DSO, can fill the void. Page migration can help moving data to the NDP, aiding the user with the data locality challenge described in 4.3, and provide (semi) optimal solutions in complex cases. The clearest example of this is when data is initialized by the CPU without specific data locality optimization, but is used by the NDPs afterwards. Besides optimizing the location of data, also threads can be migrated around. Threads running on the CPU can be monitored for data stalls, and after evaluation be migrated to an NDP, sitting closer to the data. Even more elaborate optimizations can be envisioned. In case the various memory controllers offer different access characteristics, e.g. a different access granularity, spatial locality characteristics can be collected from the cores, and pages accessed with little locality can be migrated to the most appropriate memory.

8.5. TOWARDS DATA-CENTRIC SYSTEMS

In this chapter an NDP is generalized towards a 'device optimized for data intensive applications', and we can position it relative to a CPU and a GPU. Following this terminology, a GPU would be a 'device optimized for compute throughput', and a CPU would be a 'device optimized for low-latency processing'. In Figure 8.3 a radar chart is shown containing a variety of metrics for two existing devices: an IBM POWER8 CPU and an NVidia Tesla P100, both state-of-the-art products. The CPU and the GPU are clearly worlds apart in specifications, most striking the processing capabilities, the memory capacity, and the control flow handling (instruction dispatch / ALU). This difference is of course the reason they are used together in heterogeneous systems, where both devices take care of the part of the application that best matches their characteristics. Shown as well is a band of interest for the NDP, picking a different design point. In this section we will highlight the differences characteristics of CPUs and NDPs, where they originate, and how they can help us creating data-centric systems.

8.5.1. WEAK PARALLEL PROCESSING

Similar to GPUs, NDPs should offer relative weak parallel processing, but for different reasons. GPUs use weak parallel processing to 1) have an energy-efficiency sweetspot for the floating-point units, and 2) utilize the assumed regularity in the compute-intensive kernels. NDPs, on the other hand, use weak parallel processing because 1) typical NDP workloads do not require a lot of processing, and 2) to hide the memory latency for irregular memory accesses. This follows the rationale that, given the situation that threads have to wait 60-80ns for most of the memory accesses, optimizing the processing related to that access from, e.g. 5 cycles to 3 cycles, is not worthwhile. Although this concept is shown in several related works, it is taken to the extreme in Chapter 6, using 64 very weak in-order cores. This allows for, when assuming one level of software prefetching, 128 outstanding memory accesses (two per core). When considering eight NDPs, this results in 1024 outstanding memory accesses, while a 10-core POWER8 CPU can support 'only' 400 [180]. In Chapter 4 and Chapter 6 cache sizes in the 0.5-2 KB range are used, realizing a storage capacity similar to the accumulated register file size in a modern CPU core or coprocessors core [180] [232], again illustrating the 'weakness' of the used solution, while realizing high performance for both HPCG and Graph500. In Section 1.6.3 already some techniques are discussed to make weak cores appear stronger for typical NDP workloads, foremost focusing on memory accesses. Using reconfigurable logic to create optimized processing capabilities for the workload at hand can be considered the ultimate case, allowing for a perfect balance between processing and memory access parallelism, as explored in Chapter 7. The weak parallel processing aspect is reflected in Figure 8.3 by 'processing / thread', 'instruction dispatch / ALU', and to some extent by 'bandwidth / processing', giving NDPs more weak parallel processing compared to CPUs.

8.5.2. LARGE MEMORIES, HIGH BANDWIDTH, AND LOW LATENCY

Focusing on large memories is instrumental for near-data processing. This directly aligns with the 'big-data' trend, as well as the usage of increasingly large data sets for HPC and analytics workloads. It is also a key differentiator with GPUs: e.g. the graph problems explored in Chapter 6 are much bigger than a typical memory directly attached to a GPU. By performing near-data processing directly on the main-memory of a CPU, the capacity is already available, and the memory capacity shown in Figure 8.3 is therefore similar to a CPU. As discussed throughput this work, the fact that the memory is distributed over several memory channels is a challenge, but not fundamentally different from working with multi-socket CPU systems.

High memory bandwidths can be realized by using novel memory technologies, like the Hybrid Memory Cube. However, in more general terms, a possible higher bandwidth follows from the fact of having several smaller devices instead of one large device. As discussed in Section 1.3.1, bandwidth is an IO problem, and several small devices have a larger accumulated perimeter, allowing more wires to escape the chip. Throughout this work we assumed having several NDPs, each having four DDR DRAM memory controllers at an aggressive design point, realizing a higher bandwidth than a CPU. Realizing a much higher bandwidth however, comparable to what GPUs offer, for the targeted storage capacity, is however not realistic or not likely to be cost-effective.

By sitting physically closer to the memory, with less of a memory hierarchy compared to a CPU, some of the memory access latency can be cut. However, there are several factors limiting the lower bound of the latency. First, the memory controller latency will have to be paid in any case, also when considering a near-data processor stacked on top of DRAM. Second, when considering a generic processing architecture consisting of many processing elements, and a non-perfect (i.e. realistic) data-compute affinity, the practical memory access latency can still reach hundreds of cycles, even when considering an optimized 3D stacked near-data processor [102].

8.5.3. HIGH IOPS

The most differentiating aspect, following Figure 8.3, is the number of memory accesses that can be performed per unit of time (IOPS). This directly relates to the already discussed focus points for a near-data processor. We have more memory access parallelism and a lower latency, together allowing a smaller access granularity, resulting in, especially with a higher bandwidth, more realizable IOPS. Throughout this work we assumed an access granularity of 32 B or 64 B, resulting in over 2x or 4x more IOPS compared to a CPU. This is especially valuable for graph workloads performing many random memory accesses, as is shown in [213], but has also shown to be valuable for sparse matrix methods in Chapter 6. Using such small (or even smaller) access granularities is however not cheap, and requires significant parallelism from both the application as well as the hardware. Furthermore, a hardware mechanism like coherence will become more costly, as it has to keep track of the state of the data at a smaller granularity. Tradeoffs will have to be made between the load / store efficiency of the architecture (small access granularity), and its simplicity (large access granularity). The integration of NDPs by means of SMP fabric does however not pose a problem. As stated in Section 8.3, and proposed in Chapter 4, every device can manage coherence in its own optimized way, as long as it complies with the interconnect fabric, making it a challenge as well as an opportunity.

8.6. CONCLUSIONS

In this chapter we took the step from near-data processing towards data-centric systems, where IOPS-focused devices, optimized for data intensive applications, co-exists next to CPUs and GPUs. By making NDPs full-class citizens in a computer system, various NDP related problems are solved automatically, while also opening up interesting opportunities for hardware and software managed system-level data-centric optimizations.

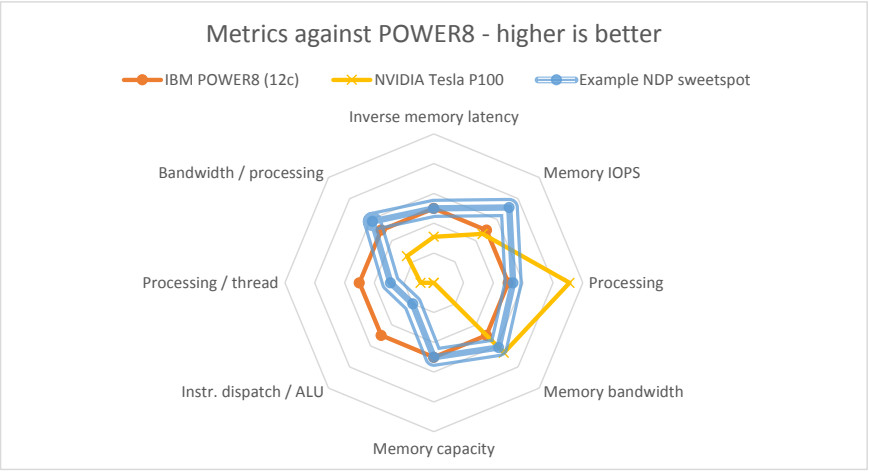


Figure 8.3: Positioning NDPs between a CPU and a GPU, creating a device optimized for data intensive applications. The chart uses a logarithmic scale.

9

CONCLUSIONS AND FUTURE WORK

In Chapter 1, an extensive overview of the challenges and fruitful research directions in the field of computer architecture and high-performance computing is given. There are no free rides anymore in the realization of ever faster computer systems. Technology scaling is coming to an end, and maintaining historical performance growth rates is turning out to be harder and harder. Data movement is a major limitation in both performance and power-efficiency, and with applications becoming more data-intensive, the problem is becoming more apparent. The use of workload-optimized systems, optimized for data-intensive applications, can increase the performance of relevant applications beyond what is possible with more traditional systems. The near-data processing paradigm fits this goal very well, by bringing processing capabilities closer to the memory.

9.1. CONCLUSIONS

We have shown in Chapter 1 that, in general, the computer system industry has to resort to workload-optimized systems to further boost performance beyond traditional scaling laws. It has furthermore been shown that data-intensive workloads are becoming more important. From this we conclude that a new focus on workload-optimized systems for data-intensive applications is necessary.

We described in Chapter 2 the extraordinary requirements in terms of compute capabilities, bandwidths, and storage for the Square Kilometre Array (SKA) radio-telescope. The characteristics of typical SKA workloads are analyzed, and the SKA design points are compared to historical trends, showing the mismatch between what industry will most likely be able to deliver, and what the SKA requires. From this we conclude that realizing the SKA at its full potential will not be possible without significant efforts in the development of optimized computer systems.

A custom ASIC, able to run all algorithms necessary for the Central Signal Processor (CSP) subsystem of the SKA, is presented in Chapter 3. The ASIC is connected to novel memory technologies and can handle the bandwidth, local storage, and compute

requirements of the CSP, at an impressive 208 GFLOPS/W power-efficiency. From this we conclude that novel workload-optimized design can truly aid in the realization of the SKA.

To realize an easy to use computer system following the near-data processing paradigm, optimized for data-intensive applications, we performed an extensive study on the subject matter in Chapter 4. To integrate near-data processing capabilities in a computer system, various hurdles have to be taken. We conclude that a selective extension of the system bus is essential to enable a deep integration, required for ease of use and efficiency. By keeping track of the coherence state of the memory in fine- and coarse-grained way, we are able to realize an effective coherence mechanism between CPU and NDP. The coarse-grained coherence state tracking is mirrored at the CPU side, and used to limit the amount of coherence and virtual memory management traffic from the CPU towards the NDPs, thereby limiting the negative effect the NDPs have on the CPU performance. Various synthetic benchmarks show the working as well as the good performance of the envisioned coherence and inter-NDP communication mechanisms. This is again shown for a complex multi-NDP graph-traversal benchmark, easily outperforming a CPU.

HPCG and Graph500 are recognized as key data-intensive benchmarks, and studied while running on the proposed architecture in Chapter 6. We deem the use of standard benchmarks essential, as it is the only way to make absolute performance comparisons between the proposed architecture and existing work. For the benchmarks, an architectural sweet-spot is established, and a variety of software optimizations are presented. When studying the HPCG benchmark, we conclude that a local memory size of 2 KB per core is optimal, and that a 64 B access granularity with software prefetching is more efficient than a 128 B access granularity. For the Graph500 benchmark, we conclude that the high-bandwidth inter-NDP communication fabric and the exploitation of locality of remote data accesses is essential. The user-defined ability to cache data belonging to another memory channel in the hardware managed caches of an NDP improves performance as well.

We evaluated the sorting of large data sets when making use of both the CPU and the NDP in a heterogeneous configuration in Chapter 7. By distributing the workload between the two devices based on the data access pattern, we make use of the NDP for its most fundamental property. From the analysis we conclude that, when the data for a kernel execution has to come from main-memory anyway (and the kernel is sufficiently large etc.), it is better to do the operations at the NDP. Another conclusion is that the heterogeneous usage of both the CPU and the NDP enables the utilization of all available bandwidth in the memory hierarchy.

9.2. FUTURE WORK

- Throughout this work we used traditional DRAM as memory technology, with the intent of keeping the proposal as realistic and cost-effective as possible. Faster memory technologies are already available, and are of clear interest for near-data processing. Interesting future research is to see how the proposed integration mechanisms scale to much higher local memory bandwidths, or whether fundamental different approaches are needed or become the better choice.

- The coherence between CPU and NDPs has been a key point in this work. A particular solution has been presented, but other options are named as well. The ability to cache remote data raises many questions and concerns, and no clear 'best' solution is available. This fuels the need for further study, describing the various options in detail and studying their performance for a variety of workloads.

- This work focuses foremost on system-level integration aspects, and application-level effects. Less attention has gone in to specific NDP related optimizations. An example optimization is the use of various access granularities. The access granularity has shown to be a topic of interest for HPCG, and is accepted as important parameter for graph workloads. The transparent use of several access granularities in a coherent CPU + NDP system is however full of challenges, e.g., how to decide which granularity to use for which access. Shown for the HPCG benchmark is that a large access granularity realizes high performance, but also results in unnecessary data transport. User-steered access granularities, similar to the user-enhanced coherence proposed in this work, is a future research topic.

- The use of workload-optimized coprocessors, implemented on a reconfigurable fabric, is a fruitful direction to boost power-efficiency and performance. This is also true for NDP coprocessors, as is for example shown for sorting large data sets. Workload-optimized NDPs have the advantage of being able to optimize their memory accesses to a great extent, by for example having very deep queues. The manual development of such NDPs is however a time consuming process, and automated methods should become available. Although steps are made in the field of automated hardware generation, pragma-based generation of optimized NDPs which can automatically connect to system interfaces as offered by the NDP-M, is still open research.

- The discussed simulation environment is developed completely from scratch. This was necessary as existing simulators were not able to capture the initial specifications of being able to simulate NDPs developed in a hardware description language. The software-only branch of the simulator however, looks much more like existing simulators. Isolating the architecture specific parts of our simulator and integrating those as an extension to existing simulators, could enable more types of experiments and easier usage, and should be considered for future work.

- We briefly touched upon the subject of making NDPs full CPUs, foremost intended to fuel future research. Many challenges are identified, foremost in the field of automated data-locality management, but also regarding the integration of various types of coherence management in a single system.

LIST OF PUBLICATIONS

JOURNAL ARTICLES

- **E. Vermij**, L. Fiorin, R. Jongerius, C. Hagleitner, J. van Lunteren and K. Bertels, *"An architecture for integrated near-data processors"*, Transactions on Architecture and Code Optimization, under review - Major revision.
- L. Fiorin, R. Jongerius, **E. Vermij**, C. Hagleitner, *"Near-memory acceleration for radio astronomy"*, Transactions on Parallel and Distributed Systems, under review - Major revision
- R. Jongerius, A. Anghel, G. Dittmann, G. Mariani, **E. Vermij**, H. Corporaal, *"Analytic multi-core processor model for fast design-space exploration"*, Transactions on Computers, under review.
- L. Fiorin, **E. Vermij**, J. Lunteren, R. Jongerius, and C. Hagleitner, *"Exploring the design space of an energy-efficient accelerator for the SKA-low central signal processor"*, International Journal of Parallel Programming, April 2016.
- **E. Vermij**, L. Fiorin, R. Jongerius, C. Hagleitner and K. Bertels, *"Challenges in exascale radio-astronomy: can we ride the technology wave?"*, The International Journal of High Performance Computing Applications, September 2014.

CONFERENCE PROCEEDINGS

- **E. Vermij**, L. Fiorin, C. Hagleitner and K. Bertels, *"Boosting the efficiency of HPCG and Graph500 with near-data processing"*, International Conference on Parallel Processing 2017.
- **E. Vermij**, L. Fiorin, C. Hagleitner, and K. Bertels, *"Sorting big data on heterogeneous near-data processing systems"*, BigDAW17, big data analytics workshop, co-located with Computing Frontiers 2017.
- **E. Vermij**, C. Hagleitner, L. Fiorin, R. Jongerius, J. van Lunteren and K. Bertels, *"An architecture for near-data processing"*, International Conference on Computing Frontiers 2016, short paper.
- V.V. Kritchallo, B. Braithwaite, **E. Vermij**, K.L.M. Bertels, Z. Al-Ars, *"Balancing High-Performance Parallelization and Accuracy in Canny Edge Detector"*, International Conference on Architecture of Computing Systems 2016.
- V.V. Kritchallo, **E. Vermij**, K.L.M. Bertels, Z. Al-Ars, *"Fidelity Slider: a User-Defined Method to Trade off Accuracy for Performance in Canny Edge Detector"*, HiPEAC 2016.
- L. Fiorin, **E. Vermij**, J. van Lunteren, R. Jongerius, and C. Hagleitner, *"An energy-efficient custom architecture for the SKA1-low central signal processor"*, International Conference on Computing Frontiers 2015, **Best paper award**.

- R. Jongerius, G. Mariani, A. Anghel, G. Dittmann, **E. Vermij**, and H. Corporaal, *"Analytic processor model for fast design-space exploration"*, International Conference on Computer Design 2015.
- **E. Vermij**, L. Fiorin, C. Hagleitner and K. Bertels, *"Exascale radio-astronomy: can we ride the technology wave?"*, International Supercomputing Conference 2014, **Best paper award**.

REFERENCES

[1] IBM, “Watson health,” "https://www.ibm.com/watson/health/oncology-and-genomics/".

[2] G. E. Moore, “Cramming More Components Onto Integrated Circuits,” in *Electronics magazine*, 1965.

[3] TOP500, “TOP500 website,” "http://www.top500.org/".

[4] G. Taylor, “Energy Efficient Circuit Design and the Future of Power Delivery,” "https://pdfs.semanticscholar.org/ab2f/b4206fe87d9a16a5b0c8de381c93510d8ff5.pdf".

[5] NRDC, “Data Center Efficiency Assessment,” "https://www.nrdc.org/sites/default/files/data-center-efficiency-assessment-IP.pdf".

[6] S. W. Keckler, W. J. Dally, B. Khailany *et al.*, “GPUs and the Future of Parallel Computing,” *IEEE Micro*, vol. 31, no. 5, 2011.

[7] G. Kestor, R. Gioiosa, D. J. Kerbyson *et al.*, “Quantifying the energy cost of data movement in scientific applications,” in *IEEE International Symposium on Workload Characterization*, Sept 2013.

[8] Z. Stephens, S. Lee, F. Faghri *et al.*, “Big Data: Astronomical or Genomical?” *PLOS Biology*, 2015.

[9] M. van Haarlem, M. Wise, A. Gunst *et al.*, “LOFAR: The LOw-Frequency ARray,” *Astronomy & Astrophysics*, vol. May, 2013.

[10] SKA organisation, “Square Kilometer Array,” "http://www.skatelescope.org/".

[11] T. Engbersen, A. Boonstra, A. Anghel *et al.*, “SKA a bridge too far, or not?” in *Exascale Radio Astronomy*, 2014.

[12] IBM, “Investor briefing 2013,” "http://www.zdnet.com/article/ibm-eyes-china-south-america-africa-and-big-data-for-2015-growth/".

[13] ITRS, “ITRS Past, Present and Future,” "https://www.dropbox.com/s/6eskh6bwdcuzpsa/1507_11_Paolo%20Overview_Out.pdf?dl=0".

[14] SemiAccurate, “Analysis: ASML stops 450mm dead,” "http://semiaccurate.com/2013/12/18/analysis-asml-stops-450mm-dead/".

[15] R. Dennard, F. Gaensslen, H.-N. Yu *et al.*, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, 2007.

[16] H. Esmaeilzadeh, E. Blem, R. St. Amant *et al.*, "Dark silicon and the end of multicore scaling," *SIGARCH Computer Architure News*, vol. 39, no. 3, Jun. 2011.

[17] K. Rupp, "40 Years of Microprocessor Trend Data," "https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/".

[18] M. B. Taylor, "A Landscape of the New Dark Silicon Design Regime," *IEEE Micro*, vol. 33, no. 5, 2013.

[19] V. Zyuban, J. Friedrich, D. M. Dreps *et al.*, "IBM POWER8 circuit design and energy optimization," *IBM Journal of Research and Development*, vol. 59, 2015.

[20] G. Venkatesh, J. Sampson, N. Goulding *et al.*, "Conservation cores: reducing the energy of mature computations," *SIGARCH Computer Architure News*, vol. 38, no. 1, Mar. 2010.

[21] N. Clark, A. Hormati, and S. Mahlke, "VEAL: Virtualized Execution Accelerator for Loops," in *35th International Symposium on Computer Architecture*, 2008.

[22] J. Brown, S. Woodward, B. Bass *et al.*, "IBM Power Edge of Network Processor: A Wire-Speed System on a Chip," *IEEE Micro*, vol. 31, no. 2, 2011.

[23] J. Stuecheli, "POWER8," in *IEEE Hot Chips Symposium*, Aug 2013.

[24] Intel, "Intel random number generator," "http://software.intel.com/sites/default/files/m/d/4/1/d/8/441_Intel_R__DRNG_Software_Implementation_Guide_final_Aug7.pdf".

[25] J. A. Kahle, M. N. Day, H. P. Hofstee *et al.*, "Introduction to the Cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4.5, July 2005.

[26] ARM, "big.LITTLE," "http://www.arm.com/products/processors/technologies/biglittleprocessing.php".

[27] Xilinx, "Zynq," "https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html".

[28] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Computer Architure News*, vol. 23, no. 1, Mar. 1995.

[29] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[30] P. D. Patterson, "Latency Lags Bandwidth," in *International Conference on Computer Design*. IEEE Computer Society, 2005.

[31] P. Stanley-Marbell, V. C. Cabezas, and R. P. Luijten, “Pinned to the walls - Impact of packaging and application properties on the memory and power walls,” in *IEEE/ACM International Symposium on Low Power Electronics and Design*, 2011.

[32] MoSys, “Serial Memories Fill a Need,” "http://www.memcon.com/pdfs/proceedings2015/TDT101_Mosys.pdf".

[33] Intel, “Intel product catalog,” "https://ark.intel.com/".

[34] W. J. Starke, J. Stuecheli, D. M. Daly *et al.*, “The cache and memory subsystems of the IBM POWER8 processor,” *IBM Journal of Research and Development*, 2015.

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

[36] AMD, “High Bandwidth Memory,” "https://www.jedec.org/standards-documents/docs/jesd235a".

[37] NVIDIA, “Tesla Pascal P100,” "https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf".

[38] R. Nair, “Evolution of Memory Architecture,” *Proceedings of the IEEE*, vol. 103, no. 8, Aug 2015.

[39] NVIDIA, “Nvlink,” "http://www.nvidia.com/object/nvlink.html".

[40] IBM, “POWER9 CPU,” "http://www.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.921-.POWER9-Thompto-IBM-final.pdf".

[41] A. Heinecke, “Accelerators in scientific computing is it worth the effort?” in *International Conference on High Performance Computing and Simulation*, 2013.

[42] OpenMP, “OpenMP specifications,” "http://www.openmp.org/specifications/".

[43] NVIDIA, “Targeting GPUs with OpenMP4.5 Device Directives,” "http://on-demand.gputechconf.com/gtc/2016/presentation/s6510-jeff-larkin-targeting-gpus-openmp.pdf".

[44] L. Grinberg, “Performance portable single source-code implemention of sparse linear algebra operations on CPUs and GPUs,” "https://asc.llnl.gov/DOE-COE-Mtg-2016/talks/2-13_Grinberg.pdf".

[45] OpenCAPI, "http://opencapi.org/about/".

[46] S. Wallace, V. Vishwanath, S. Coghlan *et al.*, “Measuring Power Consumption on IBM Blue Gene/Q,” in *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, May 2013.

[47] A. Putnam, A. M. Caulfield, E. S. Chung *et al.*, “A reconfigurable fabric for accelerating large-scale datacenter services,” in *ACM/IEEE International Symposium on Computer Architecture*, June 2014.

[48] Google, “Custom motherboard announcement,” "https://plus.google.com/111282580643669107165/posts/Uwh9W3XiZTQ".

[49] Google, “Tensor Processing Unit,” "https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html".

[50] N. P. Jouppi, C. Young, N. Patil *et al.*, “In-Datacenter Performance Analysis of a Tensor Processing Unit,” "https://arxiv.org/abs/1704.04760".

[51] J. Stuecheli, B. Blaner, C. R. Johns *et al.*, “CAPI: A Coherent Accelerator Processor Interface,” *IBM Journal of Research and Development*, 2015.

[52] S. Kamil, J. Shalf, and E. Strohmaier, “Power efficiency in high performance computing,” in *IEEE International Symposium on Parallel and Distributed Processing*, 2008.

[53] US Department of Energy, “Exascale Computing Project,” "https://exascaleproject.org/2016/11/04/ecp_background/".

[54] Green500, "http://www.green500.org/".

[55] J. J. Dongarra, P. Luszczek, and A. Petitet, “The LINPACK benchmark: Past, present, and future.” *Concurrency Computat.: Pract. Exper.*, no. 15, 2003.

[56] J. J. Dongarra and M. Heroux, “Toward a New Metric for Ranking High Performance Computing Systems,” "http://www.sandia.gov/~maherou/docs/HPCG-Benchmark.pdf".

[57] Innovative computing laboratory - University of Tennesee, “HPCG handout,” http://www.hpcg-benchmark.org/overview/index.html.

[58] Graph500, “Graph500,” "http://www.graph500.org/".

[59] V. W. Lee, C. Kim, J. Chhugani *et al.*, “Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU,” *SIGARCH Computer Architure News*, vol. 38, no. 3, Jun. 2010.

[60] H. S. Stone, “A logic-in-memory computer,” *IEEE Transactions on Computers*, 1970.

[61] D. G. Elliott, W. M. Snelgrove, and M. Stumm, “Computational Ram: A Memory-simd Hybrid And Its Application To Dsp,” in *IEEE Custom Integrated Circuits Conference*, 1992.

[62] P. M. Kogge, “EXECUBE-A New Architecture for Scaleable MPPs,” in *International Conference on Parallel Processing*. IEEE Computer Society, 1994.

[63] M. Gokhale, B. Holmes, and K. Iobst, “Processing in memory: The terasys massively parallel pim array,” *Computer*, vol. 28, 1995.

[64] R. Fromm, S. Perissakis, and N. Cardwell, "The Energy Efficiency of IRAM Architectures," in *International Symposium on Computer Architecture*, 1997.

[65] C. Kozyrakis, S. Perissakis, D. Patterson *et al.*, "Scalable processors in the billion-transistor era: IRAM," *Computer*, 1997.

[66] A. Nowatzyk, F. Pong, and A. Saulsbury, "Missing the Memory Wall: The Case for Processor/Memory Integration," in *International Symposium on Computer Architecture*, 1996.

[67] D. Patterson, T. Anderson, N. Cardwell *et al.*, "A Case for Intelligent RAM," *IEEE Micro*, 1997.

[68] M. Oskin, F. T. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory," *SIGARCH Computer Architure News*, 1998.

[69] Y. Kang, W. Huang, S.-M. Yoo *et al.*, "FlexRAM: toward an advanced intelligent memory system," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1999.

[70] D. G. Elliott, M. Stumm, W. M. Snelgrove *et al.*, "Computational RAM: implementing processors in memory," *IEEE Design Test of Computers*, 1999.

[71] B. B. Fraguela, J. Renau, P. Feautrier *et al.*, "Programming the FlexRAM Parallel Intelligent Memory System," *SIGPLAN Not.*, 2003.

[72] M. Wei, M. Wei, M. Snir *et al.*, "A near-memory processor for vector, streaming and bit manipulation workloads," in *In The Second Watson Conference on Interaction between Architecture, Circuits, and Compilers*, 2005.

[73] R. Balasubramonian, J. Chang, T. Manning *et al.*, "Near-Data Processing: Insights from a MICRO-46 Workshop," *IEEE Micro*, 2014.

[74] IBM, "IBM PureData System for Analytics," "https://www.ibm.com/us-en/marketplace/puredata-system-for-analytics".

[75] IBM, "IBM Netezza Data Warehouse Appliances," "https://www-01.ibm.com/software/data/netezza/".

[76] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: An Improved Architecture for Data-Intensive Applications," *IEEE Micro*, 2010, IEEE Micro Top Picks.

[77] E. Doller, A. Akel, J. Wang *et al.*, "DataCenter 2020: Near-memory acceleration for data-oriented applications," in *Symposium on VLSI Circuits Digest of Technical Papers*, 2014.

[78] D. O. Benjamin Y. Cho, Won Seob Jeong and W. W. Ro, "XSD: Accelerating MapReduce by Harnessing the GPU inside an SSD," "https://pdfs.semanticscholar.org/adcd/426b1235d9c68ab0432867bcc02a661b8be1.pdf".

[79] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *ACM Commun*, vol. 51, 2008.

[80] G. H. Loh, N. Jayasena, M. H. Oskin *et al.*, "A Processing-in-memory Taxonomy and a Case for Studying Fixed-function Pim," 2013.

[81] J. Ahn, S. Yoo, O. Mutlu *et al.*, "PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture," in *International Symposium on Computer Architecture*, 2015.

[82] B. Akin, J. C. Hoe, and F. Franchetti, "HAMLeT: Hardware accelerated memory layout transform within 3D-stacked DRAM," in *IEEE High Performance Extreme Computing Conference*, Sept 2014.

[83] J. Lee, J. H. Ahn, and K. Choi, "Buffered compares: Excavating the hidden parallelism inside DRAM architectures with lightweight logic," in *Design, Automation Test in Europe Conference Exhibition*, 2016.

[84] V. Seshadri, Y. Kim, C. Fallin *et al.*, "Rowclone: Fast and energy-efficient in-dram bulk data copy and initialization," in *IEEE/ACM International Symposium on Microarchitecture*, 2013.

[85] V. Seshadri, K. Hsieh, A. Boroum *et al.*, "Fast Bulk Bitwise AND and OR in DRAM," *IEEE Computer Architecture Letters*, vol. 14, 2015.

[86] V. Seshadri and O. Mutlu, "The Processing Using Memory Paradigm:In-DRAM Bulk Copy, Initialization, Bitwise AND and OR," "arXiv:1610.09603".

[87] V. Seshadri, D. Lee, T. Mullins *et al.*, "Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM," "arXiv:1611.09988, 2016.

[88] M. Gao, G. Ayers, and C. Kozyrakis, "Practical Near-Data Processing for In-Memory Analytics Frameworks," in *Conference on Parallel Architectures and Compilation Techniques*, 2015.

[89] H. Asghari-Moghaddam, A. Farmahini-Farahani, K. Morrow *et al.*, "Near-DRAM Acceleration with Single-ISA Heterogeneous Processing in Standard Memory Modules," *IEEE Micro*, 2016.

[90] M. Gao and C. Kozyrakis, "HRL: Efficient and flexible reconfigurable logic for near-data processing," in *IEEE International Symposium on High Performance Computer Architecture*, 2016.

[91] R. Nair, S. F. Antao, C. Bertolli *et al.*, "Active Memory Cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, 2015.

[92] T. E. Carlson, W. Heirman, O. Allam *et al.*, "The Load Slice Core Microarchitecture," *SIGARCH Computer Architure News*, vol. 43, no. 3, Jun. 2015.

[93] M. Hashemi, Khubaib, E. Ebrahimi *et al.*, "Accelerating dependent cache misses with an enhanced memory controller," *SIGARCH Comput. Archit. News*, 2016.

[94] S. Khoram, Y. Zha, J. Zhang *et al.*, "Challenges and Opportunities: From Near-memory Computing to In-memory Computing (INVITED)," in *International Symposium on Physical Design*, 2017.

[95] S. Hamdioui, L. Xie, H. A. D. Nguyen *et al.*, "Memristor Based Computation-in-memory Architecture for Data-intensive Applications," in *Design, Automation & Test in Europe Conference & Exhibition*, 2015.

[96] Y. Zha and J. Li, "Reconfigurable in-memory computing with resistive memory crossbar," in *IEEE/ACM International Conference on Computer-Aided Design*, 2016.

[97] P. Dlugosch, D. Brown, P. Glendenning *et al.*, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, 2014.

[98] J. Ahn, S. Hong, S. Yoo *et al.*, "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in *International Symposium on Computer Architecture*, 2015.

[99] A. Farmahini-Farahani, J. H. Ahn, K. Morrow *et al.*, "DRAMA: An Architecture for Accelerated Processing Near Memory," *Computer Architecture Letters*, 2015.

[100] D. H. Kim, K. Athikulwongse, M. Healy *et al.*, "3D-MAPS: 3D Massively parallel processor with stacked memory," in *IEEE International Solid-State Circuits Conference*, 2012.

[101] D. Zhang, N. Jayasena, A. Lyashevsky *et al.*, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *International Symposium on High-performance Parallel and Distributed Computing*, 2014.

[102] Z. Sura, A. Jacob, T. Chen *et al.*, "Data Access Optimization in a Processing-in-memory System," in *International Conference on Computing Frontiers*. ACM, 2015.

[103] K. Hsieh, E. Ebrahim, G. Kim *et al.*, "Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent Near-Data Processing in GPU Systems," in *ACM/IEEE International Symposium on Computer Architecture*, 2016.

[104] SKA organisation, "SKA1 Baseline design," "https://www.skatelescope.org/wp-content/uploads/2012/07/SKA-TEL-SKO-DD-001-1_BaselineDesign1.pdf".

[105] R. A. Perley and W. C. Erickson, "A proposal for a large, low frequency array located at the VLA site," *VLA Scientific Memorandum*, 1984.

[106] SKA organisation, "SKA phase 1 science (level 0) requirements specification," "https://www.skatelescope.org/wp-content/uploads/2014/03/SKA1-Level0-Requirements.pdf".

[107] B. Jeffs, “Beamforming,” "http://ens.ewi.tudelft.nl/Education/courses/et4235/Beamforming.pdf".

[108] A. R. Thompson, J. M. Moran, and G. W. Swenson, *Interferometry and Synthesis in Radio Astronomy; 2nd ed.* Weinheim: Wiley-VCH, 2001.

[109] A. H. Bridle and F. R. Schwab, “Wide Field Imaging I: Bandwidth and Time-Average Smearing,” *Synthesis imaging in radio astronomy*, vol. 6, 1989.

[110] C. Tasse, B. van der Tol, J. van Zwieten *et al.*, “Applying full polarization A-Projection to very wide field of view instruments: An imager for LOFAR,” *Instrumentation and Methods for Astrophysics*, vol. Dec, 12 2012.

[111] T. J. Cornwell, K. Golap, and S. Bhatnagar, “The non-coplanar baselines effect in radio interferometry: The W-Projection algorithm,” *IEEE journal of selected topics in signal processing*, vol. 2, 2008.

[112] T. J. Cornwell, M. A. Voronkov, and B. Humphreys, “Wide field imaging for the square kilometre array,” *Proceedings of SPIE 8500, Image Reconstruction from Incomplete Data VII*, vol. 8500L, 2012.

[113] J. A. Högbom, “Aperture Synthesis with a Non-Regular Distribution of Interferometer Baselines,” *Astronomy and Astrophysics*, vol. 15, June 1974.

[114] R. Jongerius, S. Wijnholds, R. Nijboer *et al.*, “An End-to-End Computing Model for the Square Kilometre Array,” *IEEE Computer*, vol. 9, 2014.

[115] A. R. Offringa, A. G. de Bruyn, S. Zaroubi *et al.*, “A LOFAR RFI detection pipeline and its first results,” in *RFI mitigation workshop, Groningen*, ser. Proceedings of Science, 2010.

[116] F. R. Schwab, “Relaxing the isoplanatism assumption in self-calibration; applications to low-frequency radio interferometry,” *Astronomical Journal*, vol. 89, July 1984.

[117] G. B. Taylor, C. L. Carilli, and R. A. Perley, Eds., *Synthesis Imaging in Radio Astronomy II*, ser. Astronomical Society of the Pacific Conference Series. Astronomical Society of the Pacific, 1999, vol. 180.

[118] S. Vassiliadis, S. Wong, G. Gaydadjiev *et al.*, “The MOLEN polymorphic processor,” *IEEE Transactions on Computers*, vol. 53, no. 11, 2004.

[119] Convey Computer, "http://www.conveycomputer.com".

[120] Intel, “SSE and AVX ISA extensions,” "http://software.intel.com/en-us/intel-isa-extensions".

[121] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, “Efficient vectorization of the FIR filter,” in *Workshop on Circuits, Systems and Signal Processing*, 2005.

[122] R. Jongerius, H. Corporaal, C. Broekema *et al.*, "Analyzing LOFAR station processing on multi-core platforms." ICT Open, 2012.

[123] J. Romein, "Signal Processing on GPUs for Radio Telescopes." GPU Technology Conference, 2013.

[124] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *IEEE International Conference on Acoustics Speech and Signal Processing*, 1998.

[125] W. Xu, Z. Yan, and D. Shunying, "A high performance FFT library with single instruction multiple data (SIMD) architecture," in *International Conference on Electronics, Communications and Control*, 2011.

[126] J. Lobeiras, M. Amor, and R. Doallo, "FFT Implementation on a Streaming Architecture," in *Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, 2011.

[127] A. Szomoru, "The UniBoard: A multi-purpose scalable high-performance computing platform for radio-astronomical applications," in *URSI General Assembly and Scientific Symposium*, 2011.

[128] R. van Nieuwpoort and J. Romein, "Correlating radio astronomy signals with many-core hardware," *International Journal of Parallel Programming*, 2011.

[129] J. W. Romein, P. C. Broekema, J. D. Mol *et al.*, "The LOFAR correlator: implementation and performance analysis," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[130] M. A. Clark, P. C. L. Plante, and L. J. Greenhill, "Accelerating Radio Astronomy Cross-Correlation with Graphics Processing Units," *International Journal of High Performance Computing Applications*, 2013.

[131] A. Woods, "Accelerating Software Radio Astronomy FX Correlation with GPU and FPGA Co-processors," Master's thesis, University of Cape Town, 2010.

[132] L. de Souza, J. Bunton, D. Campbell-Wilson *et al.*, "A Radio Astronomy Correlator Optimized for the Xilinx Virtex-4 SX FPGA," in *International Conference on Field Programmable Logic and Applications, 2007*, Aug 2007.

[133] Intel, "Signal Processing on Intel Architecture: Performance Analysis using Intel Performance Primitives," "http://www.intel.nl/content/dam/doc/white-paper/signal-processing-on-intel-architecture.pdf".

[134] A. S. van Amesfoort, A. L. Varbanescu, H. J. Sips *et al.*, "Evaluating Multi-core Platforms for HPC Data-intensive Kernels," in *ACM Conference on Computing Frontiers*, 2009.

[135] B. Humphreys and T. Cornwell, "Analysis of Convolutional Resampling Algorithm Performance," "http://www.skatelescope.org/uploaded/59116_132_Memo_Humphreys.pdf", 2011.

[136] A. L. Varbanescu, A. S. van Amesfoort, T. Cornwell *et al.*, "Building high-resolution sky images using the Cell/B.E." *Sci. Program.*, vol. 17, no. 1-2, Jan. 2009.

[137] J. W. Romein, "An Efficient Work-Distribution Strategy for Gridding Radio-Telescope Data on GPUs," in *ACM International Conference on Supercomputing*, June 2012.

[138] N. Binkert, B. Beckmann, G. Black *et al.*, "The Gem5 Simulator," *SIGARCH Computer Architure News*, vol. 39, no. 2, Aug. 2011.

[139] S. Li, J. H. Ahn, R. Strong *et al.*, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO-42*, 2009.

[140] J. Leng, T. Hetherington, A. ElTantawy *et al.*, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *International Symposium on Computer Architecture*, 2013.

[141] A. Bakhoda, G. L. Yuan, W. W. L. Fung *et al.*, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *IEEE International Symposium on Performance Analysis of Systems and Software*, April 2009.

[142] E. Vermij, L. Fiorin, C. Hagleitner *et al.*, "Exascale Radio Astronomy: Can We Ride the Technology Wave?" in *Supercomputing*, ser. Lecture Notes in Computer Science, 2014.

[143] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Symposium on VLSI Technology*, June 2012.

[144] S. H. Pugsley, J. Jestes, R. Balasubramonian *et al.*, "Comparing Implementations of Near-Data Computing with In-Memory MapReduce Workloads," *IEEE Micro*, 2014.

[145] J. van Lunteren, "Memory-Driven Near-Data Acceleration and its application to DOME/SKA," HPC User Forum, September 2014.

[146] A. Pedram, J. McCalpin, and A. Gerstlauer, "Transforming a linear algebra core to an FFT accelerator," in *International Conference on Application-Specific Systems, Architectures and Processors*, 2013.

[147] H. Karner, M. Auer, and C. W. Ueberhuber, "Top Speed FFTs for FMA Architectures," 1998.

[148] S. Galal and M. Horowitz, "Energy-efficient floating-point unit design," *IEEE Transactions on Computers*, vol. 60, no. 7, July 2011.

[149] J. Van Lunteren, "High-Performance Pattern-Matching for Intrusion Detection," in *IEEE International Conference on Computer Communications*, 2006.

[150] J. van Lunteren, "Towards memory centric computing: a flexible address mapping scheme," in *IEEE Canadian Conference on Electrical and Computer Engineering*, May 1999.

[151] L. R. D'Addario, "Low-Power Correlator Architecture for the Mid-Frequency SKA, Memo 133," Jet Propulsion Laboratory, California Institute of Technology, Tech. Rep., 2011.

[152] J. R. Geraci and S. M. Sacco, "A Transpose-free In-place SIMD Optimized FFT," *ACM Transactions on Architecture and Code Optimizations*, vol. 9, no. 3, Oct. 2012.

[153] S. Thoziyoor, J. H. Ahn, M. Monchiero *et al.*, "A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies," in *International Symposium on Computer Architecture*, 2008.

[154] G. Chen, M. Anders, H. Kaul *et al.*, "A 340mv-to-0.9v 20.2tb/s Source-Synchronous Hybrid Packet/Circuit-Switched 16 x 16 Network-on-Chip in 22nm Tri-Gate CMOS," in *International Solid-State Circuits Conference*, Feb 2014.

[155] G. Balamurugan, J. Kennedy, G. Banerjee *et al.*, "A Scalable 5-15 Gbps, 14-75 mW Low-Power I/O Transceiver in 65 nm CMOS," *IEEE Journal of Solid-State Circuits*, April 2008.

[156] "International Technology Roadmap for Semiconductors 2012 Update," "http://www.itrs.net".

[157] R. Borkar, M. Bohr, and S. Jourdan, "Advancing Moore's Law in 2014 - The Road to 14 nm," Intel Presentation, August 2014.

[158] B. Giridhar, M. Cieslak, D. Duggal *et al.*, "Exploring DRAM organizations for energy-efficient and resilient exascale memories," in *Supercompute*, Nov 2013.

[159] T. Lippert, N. Petkov, P. Palazzari *et al.*, "Hyper-systolic matrix multiplication," *Parallel Computing*, 2001.

[160] S. Vangal, J. Howard, G. Ruhl *et al.*, "An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, Jan 2008.

[161] McKinsey Global Institute, "Big data: The next frontier for innovation, competition and productivity," 2011.

[162] M. Zaharia, M. Chowdhury, M. J. Franklin *et al.*, "Spark: Cluster Computing with Working Sets," in *USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[163] H. Zhang, G. Chen, B. C. Ooi *et al.*, "In-Memory Big Data Management and Processing: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, 2015.

[164] A. McLaughlin and D. Bader, "Revisiting edge and node parallelism for dynamic gpu graph analytics," in *Parallel Distributed Processing Symposium Workshops*, 2014.

[165] Intel, “A revolutionary breakthrough in memory technology,” "http://www.intel.com/newsroom/kits/nvm/3dxpoint/pdfs/Launch_Keynote.pdf".

[166] J. Jung and Y. Won, “nvramdisk: A Transactional Block Device Driver for Non-Volatile RAM,” *IEEE Transactions on Computers*, 2016.

[167] H. P. Hofstee, G. C. Chen, F. H. Gebara *et al.*, “Understanding system design for Big Data workloads,” *IBM Journal of Research and Development*, 2013.

[168] A. De, M. Gokhale, R. Gupta *et al.*, “Minerva: Accelerating Data Analysis in Next-Generation SSDs,” in *International Symposium on Field-Programmable Custom Computing Machines*, 2013.

[169] E. Azarkhish, D. Rossi, I. Loi *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*, 2015.

[170] IBM, “Summit and Sierra Supercomputers,” "http://www.teratec.eu/actu/calcul/Nvidia_Coral_White_Paper_Final_3_1.pdf".

[171] B. Falsafi, M. Stan, K. Skadron *et al.*, “Near-Memory Data Services,” *IEEE Micro*, 2016.

[172] G. C. Harvey, *Memory Systems and Pipelined Processors*. Jones and Bartlett Publishers, 1996, page 178.

[173] E. Azarkhish, D. Rossi, I. Loi *et al.*, *Design and Evaluation of a Processing-in-Memory Architecture for the Smart Memory Cube*. Springer, 2016.

[174] S. F. Yitbarek, T. Yang, R. Das *et al.*, “Exploring specialized near-memory processing for data intensive operations,” in *Design, Automation Test in Europe Conference Exhibition*, 2016.

[175] M. Y. Thompson, J. M. Barton, T. A. Jermoluk *et al.*, “Translation Lookaside Buffer Synchronization in a Multiprocessor System,” in *In USENIX Winter Conference*, 1988.

[176] K. Hsieh, S. Khan, N. Vijaykumar *et al.*, “Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation,” in *IEEE International Conference on Computer Design*, Oct 2016.

[177] J. Navarro, S. Iyer, P. Druschel *et al.*, “Practical, Transparent Operating System Support for Superpages,” *SIGOPS Oper. Syst. Rev.*, 2002.

[178] M. Gorman, “Understanding the Linux Virtual Memory Manager,” "https://www.kernel.org/doc/gorman/pdf/understand.pdf".

[179] A. Farmahini-Farahani, J. H. Ahn, K. Morrow *et al.*, “NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules,” in *High Performance Computer Architecture*, 2015.

[180] B. Sinharoy, J. A. V. Norstrand, R. J. Eickemeyer *et al.*, "IBM POWER8 processor core microarchitecture," *IBM Journal of Research and Development*, 2015.

[181] A. Boroumand, S. Ghose, B. Lucia *et al.*, "LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory," *IEEE Computer Architecture Letters*, 2016.

[182] M. Dobson, P. Gaughe, M. Hohnbaum *et al.*, "Linux Support for NUMA Hardware," in *Proceedings of the Linux Symposium*, 2003.

[183] H. Franke, J. Xenidis, C. Basso *et al.*, "Introduction to the Wire-speed Processor and Architecture," *IBM Journal of Research and Development*, 2010.

[184] R. Raghavan, R. J. Eickemeyer, A. C. Sawdey *et al.*, "IBM POWER8 performance and energy modeling," *IBM Journal of Research and Development*, vol. 59, no. 1, Jan 2015.

[185] B. Sinharoy, R. Kalla, W. J. Starke *et al.*, "IBM POWER7 multicore server processor," *IBM Journal of Research and Development*, May 2011.

[186] S. Sarkar, P. Sewell, J. Alglave *et al.*, "Understanding POWER Multiprocessors," *SIGPLAN Not.*, 2011.

[187] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.

[188] A. Basu, J. Gandhi, J. Chang *et al.*, "Efficient Virtual Memory for Big Memory Servers," *SIGARCH Computer Architure News*, vol. 41, no. 3, Jun. 2013.

[189] IBM, "Openpower," "http://open-power.org/".

[190] C. Luk, R. Cohn, R. Muth *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *SIGPLAN Not.*, 2005.

[191] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems," *SIGARCH Computer Architure News*, 2013.

[192] Z. Guz, M. Awasthi, M. Balakrishnan *et al.*, "Real-Time Analytics as the Killer Application for Processing-In-Memory," *2nd Workshop on Near-Data Processing*, 2014.

[193] A. Buluç and K. Madduri, "Parallel Breadth-First Search on Distributed Memory Systems," *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[194] F. Checconi and F. Petrini, "Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines," in *IEEE International Parallel & Distributed Processing Symposium*, 2014.

[195] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing Breadth-First Search," in *Supercomputing*, 2012.

[196] C. Johnson, D. H. Allen, J. Brown *et al.*, "A wire-speed power processor: 2.3GHz 45nm SOI with 16 cores and 64 threads," in *IEEE international Solid-State Circuits Conference*, 2010.

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

[198] H. Giefers, R. Polig, and C. Hagleitner, "Accelerating Arithmetic Kernels with Coherent Attached FPGA Coprocessors," in *Design, Automation & Test in Europe Conference & Exhibition*, 2015.

[199] IBM, "Contutto," "https://openpowerfoundation.org/presentations/contutto/".

[200] IBM, "Making Unforgettable MRAM Memory with OpenPOWER," "https://openpowerfoundation.org/blogs/making-unforgettable-mram-memory-openpower/".

[201] Altera, "Avalon Interface Specifications," "https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/mnl_avalon_spec.pdf.

[202] E. Panainte, K. Bertels, and S. Vassiliadis, "Compiling for the Molen Programming Paradigm," in *Field Programmable Logic and Application*, 2003, vol. 2778.

[203] Linux community, "Linux process pagemap," "https://www.kernel.org/doc/Documentation/vm/pagemap.txt".

[204] Radio-astronomy community, "CASA - Common Astronomy Software Applications," "https://casa.nrao.edu/".

[205] J. J. Dongarra, "HPCG benchmarking," "http://www.sandia.gov/~maherou/docs/HPCG-Benchmark.pdf".

[206] V. Marjanovic, J. Gracia, and C. W. Glass, "Performance Modeling of the HPCG Benchmark," in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems Workshop*, 2014.

[207] J. Park, M. Smelyanskiy, K. Vaidyanathan *et al.*, "Efficient Shared-Memory Implementation of HPCG Benchmark and its Application to Unstructured Matrices," "http://pcl.intel-research.net/publications/sc14_hpcg.pdf", 2014.

[208] J. Park, M. Smelyanskiy, K. Vaidyanathan *et al.*, "HPCG on Intel Architecture Update Nov 2015," "http://www.hpcg-benchmark.org/downloads/sc15/sc15-hpcg-bof-intel.pdf".

[209] Q. Zhu, T. Graf, H. E. Sumbul *et al.*, "Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware," in *IEEE High Performance Extreme Computing Conference*, 2013.

[210] Y. Yasui and K. Fujisawa, "Fast and scalable NUMA-based thread parallel breadth-first search," in *International Conference on High Performance Computing Simulation*, 2015.

[211] Z. Cui, L. Chen, M. Chen *et al.*, "Evaluation and Optimization of Breadth-First Search on NUMA Cluster," in *IEEE Cluster*, 2012.

[212] M. A. Z. Alves, M. Diener, P. C. Santos *et al.*, "Large vector extensions inside the HMC," in *Design, Automation and Test in Europe Conference*, 2016.

[213] E. Vermij, C. Hagleitner, L. Fiorin *et al.*, "An Architecture for Near-data Processing Systems," in *Computing Frontiers*, 2016.

[214] X. Liu, D. Buono, F. Checconi *et al.*, "An Early Performance Study of Large-Scale POWER8 SMP Systems," in *IEEE International Parallel and Distributed Processing Symposium*, May 2016.

[215] T. Iwashita, H. Nakashima, and Y. Takahashi, "Algebraic Block Multi-Color Ordering Method for Parallel Multi-Threaded Sparse Triangular Solver in ICCG Method," in *IEEE International Parallel & Distributed Processing Symposium*, 2012.

[216] Intel, "General Memory," "http://www.intel.com/content/www/us/en/intel-developer-forum-idf/san-francisco/2016/idf-2016-san-francisco.html".

[217] NVIDIA, "K80," "https://images.nvidia.com/content/pdf/kepler/Tesla-K80-BoardSpec-07317-001-v05.pdf".

[218] F. Checconi, F. Petrini, J. Willcock *et al.*, "Breaking the speed and scalability barriers for graph exploration on distributed-memory machines," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2012.

[219] Y. Yasui, K. Fujisawa, and K. Goto, "NUMA-optimized parallel breadth-first search on multicore single-node system," in *IEEE BigData*, 2013.

[220] T. Honjo and K. Oikawa, "Hardware acceleration of Hadoop MapReduce," in *IEEE International Conference on Big Data*, Oct 2013.

[221] P. Trancoso, "Moving to Memoryland: In-memory Computation for Existing Applications," in *ACM International Conference on Computing Frontiers*, 2015.

[222] D. Diamantopoulos and C. Kachris, "High-level synthesizable dataflow MapReduce accelerator for FPGA-coupled data centers," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, July 2015.

[223] Y. Chen, H. Zhu, H. Jin *et al.*, "Algorithm-level Feedback-controlled Adaptive data prefetcher: Accelerating data access for high-performance processors," *Parallel Computing*, vol. 38, 2012.

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

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

[226] A. Venkat and D. M. Tullsen, “Harnessing ISA Diversity: Design of a heterogeneous-ISA Chip Multiprocessor,” in *International Symposium on Computer Architecuture*. IEEE Press, 2014.

[227] D. Unat, A. Dubey, T. Hoefler *et al.*, “Trends in Data Locality Abstractions for HPC Systems,” *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[228] G. Piccoli, H. N. Santos, R. E. Rodrigues *et al.*, “Compiler support for selective page migration in NUMA architectures,” in *International Conference on Parallel Architecture and Compilation Techniques*, 2014.

[229] Barcelona supercomputing center, “OMPSS,” "https://pm.bsc.es/ompss".

[230] IBM, “AIX Operating System,” "http://www-03.ibm.com/systems/power/software/aix/".

[231] T. Do, B. Olszewski, and B. Mealey, “IBM AIX Dynamic System Optimizer - A New Approach to Autonomous System Performance,” IBM, Tech. Rep., 2013.

[232] J. Fang, A. L. Varbanescu, H. Sips *et al.*, “An Empirical Study of Intel Xeon Phi,” "https://arxiv.org/abs/1310.5842".

ACKNOWLEDGEMENTS

Many phd-thesis acknowledgments start by saying that this is one of the last parts of the thesis you write. This turns out to be true.

It has been almost four years by now, and I would not have gotten this far without the many excellent people around me, and acknowledgements are in place.

Ceciel, thanks for the wonderful years and supporting me while I was never there, either away physically in Groningen, or away mentally thinking about computers. But it is painful that you are not sitting next to me while writing this section. With this book out of the way there is hopefully time and energy again for a beer.

Doing a phd at IBM is a fantastic experience, with ample opportunities to learn interesting things. To get a solid understanding of the subject matter, I had to reach out to IBMers all around the world. Although difficult at first, these contacts have been a privilege and many times an inspiration. Sometimes a single email, sometimes many phone calls, but most importantly, I never got a 'no'. Everyone was always happy to help out, also the people known to be very busy, and I will take this away as a one of the most positive experiences of last years. Some I will name personally, as they have been essential for my work: Charlie Johns, Ravi Nair, Fabio Checconi, Leopold Grinberg, and Anton Blanchard. To you, and to all the IBMers I do not name personally: thanks.

Christoph, thanks for being my supervisor and copromotor, and helping me out and supporting me many times on a variety of topics. I always enjoyed our interactions, both the technical ones as well as the ones during our drinks in Zurich and the Netherlands. You never backed off from some late night work or driving a social event to make me (us) part of the Zurich accelerator group, and I appreciate that a lot.

Peter, you have been invaluable for many aspects of my work. The discussions with you have been an inspiration and motivation, and I owe you a lot of my contacts within IBM. Thanks for having the trust in me and my work and help me get to Austin to talk with the big guys, and of course thanks for helping with the last months to get the thesis finalized in time and in good shape.

Koen, it is strange to write something here. It is fantastic that you seem to be doing really well, but while writing this you most likely do not know I actually finalized my phd. Thanks for being my promotor, supporting and supervising me throughout the years, and accepting this sort of part-time phd arrangement. I always enjoyed our contacts and your enthusiasm, and your high-level professor insights have been a great help. It would be fantastic to see you again soon and I hope you can, in time, lead the department again with the same energy and drive.

And of course thanks to my doctoral committee: Kees Vuik, Henk Corporaal, Lieven Eeckhout, Onur Mutlu, and Alle-Jan van der Veen. Thanks for taking the time and inter-

est to review my work, and contribute to the quality of this thesis.

Leandro, you have been my P5 buddy, and together we solved the compute part of the SKA. Thanks for all the help (especially the paper writing...), and the extra support the last months to get all the submissions out in time. Rik, although you were not part of the coolest DOME workpackage, you were my phd buddy, helped me out many times, and the (technical) discussions were always useful, many thanks for that. With the two of you many beers have been drunk as well, which was always a welcome change of scenery from looking at computer screens. Bram, thanks for the eight o'clock cups of coffee, your enthusiasm about GPUs, and helping me out with the delicacies of Linux and all the other computer things I do not understand. Yusik, Yan, Chris, Przemek, Giovanni, Matthias, and Liying, thanks for all the lunches, coffees, Christmas dinners, and many discussions about the range of interesting aspects of the project.

At ASTRON I had the opportunity to learn about radio-astronomy, radio-astronomers, and all related research. I would like to thank John, Albert-Jan, Ronald, and Bas for explaining me the things I had to learn. The times in the IBM Zurich lab, as well as the occasional evening activities in Zurich and group sledging events, were made useful and enjoyable by the many people there: Gero, Andreea, Jan, Silvio, Francois, Christoph, and many more: thanks.

From my colleagues at the university in Delft I would like to thank Valery, Imran, and Razvan for the nice collaborations on a variety of subjects, and Johan and Joost for the many lunches and the occasional drink. And Zaid and Said, thanks for helping me with finalizing this thesis.

My work would not have been possible without the excellent support from ASTRON, IBM Amsterdam, and the university. Marja, Ina, Alexander, Anh Bui, Gabriella, Erik, Lidwina, and Joyce, thanks for all the support regarding the practicalities of doing this job.

As I enjoyed the luxury of having two houses, I also enjoyed the company of various housemates. Joris, Michiel, and Inge, thanks for making the time in Groningen enjoyable with the many dinners, drinks, and discussions.

Erik Vermij
May 2017

SAMENVATTING

De snelheid van supercomputers groeit niet meer zo snel als het vroeger deed. Enkele jaren geleden werd een breuk met historische trends zichtbaar. Eerst bij de 'langzaamste' van de wereldwijd geïnstalleerde systemen, maar tegenwoordig ook voor de gemiddelde snelheid van een grote groep supercomputers. Energie verbruik is het grootste probleem aan het worden voor het ontwerpen van computersystemen. De traditionele trends waarmee energie verbruik afnam gelden niet meer voor de huidige generatie halfgeleider technologie, en de snelheid van algemeen bruikbare componenten is gelimiteerd door hun energie verbruik. Server en systeem ontwerp is vervolgens gelimiteerd door hun absolute energie verbruik, wat gelimiteerd is door kost aspecten en praktische koel methoden. Om computers sneller te maken is het steeds populairder om gespecialiseerde componenten en gespecialiseerde server ontwerpen te gebruiken. Het verplaatsen van data is herkend als grote energie verbruiker en is daarnaast ook een prestatie flessenhals wanneer het door een smalle verbinding wordt verplaatst. Nu data een steeds belangrijker eigendom wordt voor overheden, bedrijven en individuen, is het realiseren van computers geoptimaliseerd voor data-intensieve taken noodzaak. In dit werk onderzoeken we de fundamentele aspecten voor zo'n systeem en kijken naar enkele belangrijke taken.

Om de relevantie van het werk duidelijk te maken, analyseren we de realiseerbaarheid van een volgende-generatie radio-telescoop, de Square Kilometre Array (SKA). We analyseerde de vereisten met betrekking tot berekeningen, opslag en bandbreedtes, en onderzoeken het gedrag van verschillende belangrijke algoritmen op bestaande producten. De SKA kan beschouwd worden als ultieme big-data uitdaging, en zijn vereisten en karakteristieken passen niet op de bestaande producten. Door de SKA vereisten naast historische trends te zetten, zien we dat de SKA niet op zijn volledige capaciteit kan worden gerealiseerd zonder extra moeite in het ontwikkelen van geoptimaliseerde systemen.

Om een stap te maken naar een succesvolle realisatie van de SKA hebben we een specifieke hardware architectuur ontworpen voor het Central Signal Processor (CSP) subonderdeel van de SKA. De CSP vereist hoge bandbreedtes, grote lokale geheugens, en significante hoeveelheden rekenkracht. Door middel van een speciale chip, verbonden met snel en nieuw geheugen, kan de voorgestelde oplossing een hoge energie efficiëntie halen van 208 GFLOPS/W, terwijl alle CSP taken worden ondersteund. Dit is een duidelijk voorbeeld van hoe geoptimaliseerde systemen het energieverbruik kunnen verlagen, en daarbij de realisatie van niet-conventionele projecten kan helpen.

Om de efficiëntie van een verscheidenheid aan taken te vergroten, hebben we een hardware architectuur ontwikkeld welke arbitraire verwerk capaciteiten heeft vlakbij het geheugen van een CPU. Dit volgt het thema 'vlakbij-data verwerken', welke hoge bandbreedtes en gereduceerde data verplaatsingen aanbied. De inspanning beslaat de twee hoofd observaties dat 1) verwerk capaciteiten moeten taak-geoptimaliseerd zijn, en 2) een focus op data en geheugen is belangrijk voor moderne taken. De architecturale be-

schrijving beslaat data allocatie en plaatsing, coherentie tussen CPU en de vlakbij-data verwerker (VDV), virtueel geheugen management, en het gebruiken van afgelegen data. Alle data management aspecten zijn gerealiseerd door middel van standaard functionaliteiten in het besturingssysteem, en vereist alleen aanpassingen in de firmware van het systeem. De andere drie aspecten zijn gerealiseerd door middel van een nieuw component vlakbij het geheugen (VDV-Manager, VDV-M), en een nieuw component vastgemaakt an de systeem bus van de CPU (VDV Toegangs Punt, VDV-TP). De VDV-M realiseert coherentie tussen de CPU en de VDV door middel van een fijn- en grof-mazig grootboek mechanisme, terwijl het vermijdt dat onbelangrijke VDV-TP berichten naar de VDV worden verstuurd. Adres vertaling is geïmplementeerd in de VDV-M, waar de vertaal buffer wordt gevuld en gesynchroniseerd door middel van een verbindingen met de VDV-TP. De VDV-TP is verder het punt waar aanvragen voor afgelegen data de globale coherente adres ruimte inkomen. Synthetische benchmarks laten samen met een graafverwerkings taak zien dat de voorgestelde methodes werken.

De evaluatie van de architectuur alsmede de evaluatie van verschillende types VDVs vereiste de ontwikkeling van een nieuwe systeem simulator. De ontwikkelde simulator kan hardware VDVs evalueren in een gesimuleerd geheugen systeem. Willekeurige applicaties welke gebruik maken van de simulator voeden het gesimuleerde geheugen systeem met laad en opsla instructies, en kunnen de VDVs besturen. Het is ook mogelijk algemeen bruikbare VDVs welke software draaien te simuleren. Complexe systeem interacties zoals coherentie en het gebruik van afgelegen data zijn in detail gemodelleerd en geven waardevolle inzichten.

Twee relevante benchmarks voor data intensieve computersystemen zijn de Graph500, en de High-Performance Conjugate Gradient (HPCG) benchmark. Ze implementeren respectievelijk een gedistribueerde graaf breedte-eerst zoektocht en een gedistribueerde multi-raster conjugaat gradiënt oplosser. Beide benchmarks zijn geïmplementeerd op de architectuur welke tot acht VDVs bevat. Door het onderzoeken van verschillende hardware parameters alsmede verscheidene software optimalisaties kunnen we de snelheid van beide benchmarks met een factor 3 verhogen ten opzichte van CPUs. Een belangrijk aspect is de hoge bandbreedte en late vertraging in het netwerk tussen de VDVs, gerealiseerd door het VDV-TP. De cacheerbaarheid van afgelegen data in de VDV-TP maakt snelle toegang to gedeelde afgelegen data mogelijk en is belangrijk voor Graph500. Het gebruik van gebruiker-gespecificeerde coherentie is handig om twee redenen. Ten eerste verlaagt het sturen van de grofmazige coherentie methode het aantal fijnmazige coherentie grootboek verzoeken. Ten tweede versnelt de cacheerbaarheid van afgelegen data in VDV caches de applicaties, maar vereist meer programmeer moeite om manueel coherentie te verzekeren.

Een typische big-data taak is het sorteren van data. Data sorteren heeft fases met veel data lokaliteit en fases met weinig data lokaliteit. Dit opent de interessante mogelijkheid om heterogeen gebruik te maken van de CPU en de VDV, waar de twee componenten respectievelijk de fases met veel, en met weinig, data lokaliteit verwerken. De CPU maakt optimaal gebruik van zijn caches, terwijl de VDV optimaal gebruikt maakt van zijn hoge bandbreedte naar het geheugen. Dit is geëvalueerd voor een hardware VDV, en we zien snelheidsverbeteringen van 2.6x ten opzichten van enkel een CPU. Door het lage energie verbruik van een hardware VDV is de complete energie verbetering 2.5x.

CURRICULUM VITÆ

Erik Vermij was born in Gouda, the Netherlands, on August 16, 1985. He attended secondary school in Alphen aan den Rijn, after which he studied at the Delft University of Technology. In Delft, he obtained his B.Sc. and M.Sc. in Electrical Engineering. During his Master's studies he did an internship at Fox-IT, and his Master's thesis was in collaboration with Convey Computer. After graduation he worked for two years as a software engineering consultant at Alten Nederland, doing projects at Honeywell, Rijkswaterstaat, and Royal Dutch Shell.

He joined IBM Research in the Netherlands in June 2013. He works in the DOME project, a joint research program between IBM Netherlands, IBM Research – Zurich, and ASTRON, the Netherlands institute for radio astronomy. His research focuses on workload-optimized coprocessor design and the system-level integration of coprocessors. Furthermore, while working on the DOME research project, he pursued his PhD at the Delft University of Technology. During his PhD project he (co-)authored several journal articles and publications in international conferences. He won two best paper awards, one as leading author, and one as second author.