X-CEL: A Method to Estimate Near-Memory Acceleration Potential in Tile-Based MPSoCs

Sven Rheindt[1(✉)], Andreas Fried[2], Oliver Lenke[1], Lars Nolte[1], Temur Sabirov[1], Tim Twardzik[1], Thomas Wild[1], and Andreas Herkersdorf[1]

[1] Technical University of Munich (TUM), Munich, Germany
sven.rheindt@tum.de
[2] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

Abstract. Near-memory acceleration strives to tackle the data-to-task locality issue in MPSoCs in order to obtain higher performance and lower power consumption. However, it is not easy to determine whether the advantages arise from the near-memory integration or the hardware acceleration (versus software execution). We propose *X-CEL*, a method to accurately estimate the potential of near-memory acceleration using an easy-to-integrate *near-memory core*. We showcase *X-CEL's* benefits with three variants of graph copy mechanisms in a tile-based MPSoC. Evaluations reveal that the estimated speedup is in good accordance with the actual speedup achieved by the near-memory accelerator.

Keywords: Data-to-task locality · Near-memory acceleration · Design space exploration · Graph copy · Tile-based MPSoC

1 Introduction

The performance and power consumption of today's MPSoCs are dependent on data-to-task locality more than ever. A significant amount of energy and time is nowadays spent on data transfers between processor cores and the main memory, especially for memory-intensive applications, which are dominated by data access and movement [3,10]. Conventionally, sophisticated cache hierarchies are used to improve data-to-task locality by bringing data closer to the processor cores, thus lowering memory access latencies and the energy footprint. However, their benefit is decreasing due to the emergence of large, irregular and cache-unfriendly datasets, utilized by today's and future applications [10]. The locality challenge becomes worse when shifting towards tile-based manycore architectures, as on these the distance between physically distributed cores and memory grows.

Many recent approaches therefore leverage in- or near-memory computing to bridge the widening gap between processors and memory [1,16,22,25,27]. The majority of them use *near-memory accelerators* (NMAs), which perform

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 146371743 – TRR 89: Invasive Computing.

A. Brinkmann et al. (Eds.): ARCS 2020, LNCS 12155, pp. 109–123, 2020.
https://doi.org/10.1007/978-3-030-52794-5_9

their task both close to memory, as well as in a dedicated hardware module (either near the memory or as a specific accelerator layer in a 3D-stacked circuit). NMAs usually achieve a higher computational density and performance than a software solution while saving energy and resources at the same time. On the other hand, they sacrifice the flexibility of general-purpose computing and every new accelerator requires a significant hardware development effort.

However, it is not always clear which portion of the performance advantage of the NMA originates from the location (i.e., *near-memory* integration) or type of function implementation (i.e. software execution versus hardware *acceleration*). The impact of either one of the two effects is highly dependent on the application and the underlying system architecture. To determine the optimal design, it is therefore essential to analyze the influence of both effects on multiple important user- and case-specific decision criteria, such as: performance, power consumption, resource usage, design effort, flexibility (general- vs. fixed-purpose), system or accelerator utilization, etc. The analysis whether 1. a near-memory integration (near-memory core or near-memory accelerator) is beneficial at all, 2. a dedicated hardware accelerator can outperform a software-programmable core for the given task, or 3. whether only the combination of both achieves a speedup, is crucial to avoid unnecessary and costly development effort. However, it is not trivial to quantitatively predict the effect of the individual optimizations before implementing and measuring them. Further, it has to be determined if a near-memory core or accelerator can handle the workload which is outsourced to it by many cores.

Therefore, a method for speedup estimation which helps the developer to make early yet robust design choices would be of much benefit. Conventionally, a design space exploration (DSE) is mostly performed on a virtual prototype (i.e. simulation-based) or an FPGA-based prototype [8,19]. Both need at least an accurate model or an implementation of the NMA, which already requires a good amount of development effort if the DSE is expected to yield conclusive results. To avoid this effort, we envision an orthogonal approach that could be applied to both virtual and FPGA-based prototyping. We therefore

- propose *X-CEL*, an agile, measurement-based method to estimate the speedup potential of near-memory accelerators in a tile-based MPSoC,
- showcase *X-CEL* with a case study of three graph copy mechanisms (two of them are near-memory),
- and provide an in-depth evaluation of this case study.

This agile development method builds on actual measurements of an intermediate, easy-to-integrate near-memory core implementation. With the intermediate stage, we achieve a better estimation of the target design (near-memory accelerator) because in it the near-memory dimension (i.e. location) has been decoupled from the accelerator dimension (i.e. type of function implementation).

The rest of the paper is organized as follows: Sect. 2 describes the related work. In Sect. 3, we present *X-CEL* followed by the case study in Sect. 4. Section 4 is divided into a description of the showcase scenario (Sect. 4.1 and 4.2) and how we apply *X-CEL* to it (Sect. 4.3). We further perform an in-depth analysis of the evaluation results in Sect. 5, before concluding in Sect. 6.

2 Related Work

Our work is closely related to design space exploration (DSE) of heterogeneous systems. As Sangiovanni-Venticelli et al. strive to do in their *platform-based design* method [23], we place our approach early in the design phase.

During a DSE run, the DSE needs to be able to evaluate the performance of each considered design point. Conventionally, it follows either a *simulation-based* or *analytical* method as defined by Pimentel [19]. When a custom hardware unit is part of the system, both of these methods need a model of that unit to be developed beforehand. Reagen et al. [21] and Altaf et al. [2] demonstrate this approach for the simulation-based and analytical methods, respectively.

There is also the *measurement-based* evaluation method, but Pimentel associates this with a prohibitively high development overhead. This is because instead of a (simplified) model of the custom hardware, the evaluation now needs a full prototype.

Our approach, however, is orthogonal to conventional DSE and allows us to bypass the need to develop a model or prototype beforehand. As we target *near-memory acceleration* (NMA), we extrapolate its performance by leveraging measurements of an easy-to-integrate, software-programmable near-memory core, without the need for the actual accelerator.

Recently, there has been much interest in NMAs for numerical applications [16,25], graph processing [11], and system software [27]. For dealing with object graphs, Maas et al. presented an accelerator (albeit not an NMA) to speed up tracing garbage collection [13]. Rheindt et al. specifically targeted the problem of copying object graphs with an NMA [22], which is also the focus of our paper.

There are also sophisticated software-only approaches to efficiently copy object graphs without costly (de)serialization: Mohr et al. [14] presented Pegasus, which targets embedded MPSoCs, while Skyway by Nguyen et al. [17] optimized object graph transfers over networks.

3 X-CEL

X-CEL is a measurement-based method to estimate and analyze the speedup potential of near-memory accelerators in tile-based manycore architectures. To be able to conquer the complexity of this endeavor, we propose an agile two-stage approach, which separates the near-memory from the hardware accelerator dimension. This decoupling of both effects allows us to make a better estimation.

Our method categorizes the activity of a parallel application scenario running on an MPSoC into three parts: 1. the *task of interest* (TOI), which would benefit from near-memory computing and which is often memory-intensive, 2. all remaining tasks of the application, and 3. idle time of the cores. Figure 2 illustrates this for a parallel application running on N_{cpu} cores. As depicted and defined in Fig. 2, t_{toi} and t_{other} are the accumulated times over all application cores executing the task of interest and the remaining tasks, respectively. The TOI can either be given as a design choice to be explored/analyzed or it can

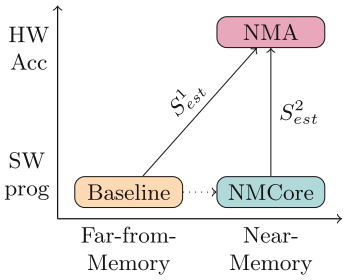

Fig. 1. *X-CEL* reduces the design space exploration complexity by one dimension.

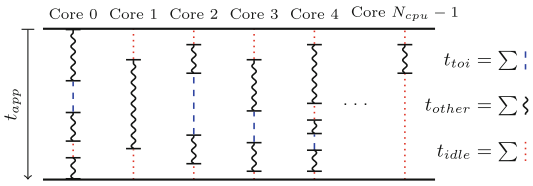

Fig. 2. Example manycore application scenario including definitions of t_{toi}, t_{other}, t_{idle} and t_{app}.

be determined through application profiling, e. g. last-level cache misses indicate which task(s) have the most DRAM accesses.

The idle times arise from sequential parts of the application, limited parallelism, data dependencies, as well as inter-thread communication and synchronization overhead. If there are several different tasks of interest, *X-CEL* could also be individually applied to them to analyze the speedup potential of each. In the following, we assume one task of interest which is executed multiple times throughout the application.

The tile-based manycore architecture we consider (an example is depicted in Fig. 4), contains a main memory, a two level cache hierarchy, many cores and potentially a software-programmable *near-memory core* (NMCore) or dedicated hardware *near-memory accelerator* (NMA). Thus, we can differentiate between three implementation variants: 1. **baseline** (far-from-memory & without accelerator): the task of interest (TOI) and all others tasks are executed parallelized on the *far-from-memory* cores, 2. **NMCore** (near-memory, but without accelerator): the task of interest is executed *near-memory* on the *near-memory core*, while all others tasks remain on the distributed cores, and 3. **NMA** (near-memory & accelerated): similar to NMCore, but the task of interest is offloaded to the *near-memory hardware accelerator*.

Beginning with the existing baseline variant, *X-CEL* introduces and leverages an agile development step via the NMCore variant. The near-memory core serves well as an intermediate step in the two-stage estimation since it has negligible development effort compared to the near-memory accelerator: The existing software algorithm of the TOI just needs to be executed on an additionally instantiated core. This offloading needs to be properly synchronized with the rest of the system. As depicted in Fig. 1, *X-CEL* decouples the near-memory from the hardware acceleration dimension. In contrast, an estimation of the NMA using the baseline measurements would incorporate a change of both dimensions at the same time. This would be a difficult endeavor in such a complex system with many superposed effects of MPSoCs and parallel programming. Therefore, a refined estimation based on the NMCore variant is more promising because the near-memory dimension is fixed due to the same location of the NMCore and the NMA in the architecture.

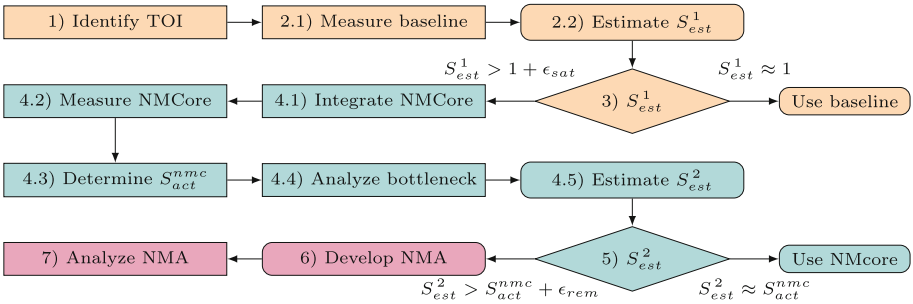

Fig. 3. Flowchart showing the steps of *X-CEL*

Our proposed *X-CEL* method thus follows the steps depicted in Fig. 3:

Step 1. Identify the task of interest (TOI) of the application scenario that could benefit from near-memory acceleration. In case of more than one TOI, apply *X-CEL* either individually or in a combined manner to them.
Step 2.1. Execute the baseline variant and measure the accumulated CPU time of all application cores taken by the task of interest t^{base}_{toi}, all other parts of the program t^{base}_{other}, as well as the overall runtime t^{base}_{app}.
Step 2.2. Determine a first speedup estimate S^1_{est} of the NMA variant using the baseline measurements. Given that only the TOI is accelerated, while the rest of the application remains untouched, an upper bound estimate is given by:

$$S^1_{est} = \frac{t^{base}_{other} + t^{base}_{toi}}{t^{base}_{other}} \tag{1}$$

Step 3. If $S^1_{est} \approx 1$, the TOI has a negligible fraction of the total execution time. There is thus no speedup potential through near-memory computing and the baseline variant can be used. If, however, $S^1_{est} > 1+\epsilon_{sat}$, where ϵ_{sat} expresses a user-defined satisfying margin, we consider it worthwhile to speedup the TOI with near-memory computing. However, the confidence of this first stage estimate S^1_{est} is not very high, as the estimation for the near-memory accelerator is based on the baseline variant which is neither near-memory integrated nor accelerated. Therefore, we refine the estimation in the next steps.
Step 4.1. Integrate the *near-memory core* (NMCore) variant.
Step 4.2. Execute this variant and measure the respective times of the different tasks t^{nmc}_{toi}, t^{nmc}_{other}, as well as the overall runtime t^{nmc}_{app}.
Step 4.3. Determine the actual speedup of the NMCore variant relative to the baseline implementation:

$$S^{nmc}_{act} = \frac{t^{base}_{app}}{t^{nmc}_{app}} \tag{2}$$

Step 4.4. Analyze whether the near-memory core becomes a bottleneck by monitoring its utilization. If $t^{nmc}_{toi} \approx t^{nmc}_{app}$, meaning the NMCore is utilized almost

during the whole execution time of the application, the use of a second near-memory core might be an option. However, as commonly known, interleaved accesses of several cores to the same DRAM memory bank can even deteriorate the performance due to row conflicts. We experienced this behavior and hence employ only one near-memory core.

Step 4.5. Based on the NMCore measurements, refine the speedup estimate for the NMA compared to the baseline variant:

$$S^2_{est} = \frac{t^{nmc}_{other} + t^{nmc}_{toi}}{t^{nmc}_{other}} \cdot S^{nmc}_{act} \tag{3}$$

As the NMCore and the NMA are located in the same position in the architecture, this second stage estimate is invariant to the near-memory dimension. It therefore promises a higher confidence.

Step 5. Compare the actual speedup achieved by the NMCore variant S^{nmc}_{act} (Step 4.3) with S^2_{est} (Step 4.5), which is the refined estimation for the NMA speedup potential. Both are relative to the baseline variant and thus directly comparable. If $S^2_{est} \approx S^{nmc}_{act}$, there is no remaining speedup potential for the hardware accelerator and the near-memory core is sufficient. If $S^2_{est} > S^{nmc}_{act} + \epsilon_{rem}$, where ϵ_{rem} expresses a big enough remaining speedup margin, the near-memory accelerator should be considered. However, the development effort and the required hardware resources of the NMA should not be neglected in this decision.

Step 6. Develop and implement the near-memory accelerator.

Step 7. Finally, measure the NMA variant and perform an analysis of how close both estimates S^1_{est} and S^2_{est} approach the NMA variant.

4 X-CEL Case Study

This section presents a case study of *X-CEL* applied to near-memory graph copy. We first motivate the choice of near-memory graph copy as a showcase scenario (Sect. 4.1) and describe the prototype and benchmark setup of our case study (Sect. 4.2), before applying *X-CEL* to it (Sect. 4.3).

4.1 Motivation for Near-Memory Graph Copy

As mentioned in Sect. 1, data-to-task locality and the reduction of data movement is especially challenging on tile-based manycore architectures. Although parallel applications and operating systems help to exploit the increased scalability, they often impose significant overhead for inter-tile communication, data transport and thread synchronization. Common communication patterns of parallel applications, libraries, and operating systems require the transfer of arbitrary data to remote tiles and its subsequent processing there. As tile-based architectures often omit hardware support for inter-tile cache coherence and consistency [4,5,12], inter-tile communication (data transfer and thread synchronization) has to be handled explicitly via e. g. message passing (e. g. MPI [15])

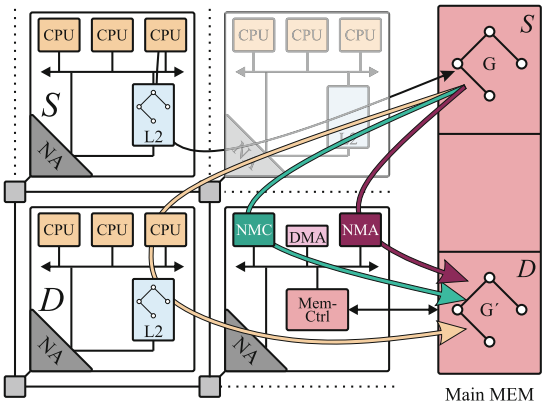

Fig. 4. Tile-based architecture.

or partitioned global address space (PGAS) programming (e. g. X10 [24] or Chapel [7]). These models have in common that they require data transfers between the memory partitions associated with each processor. These architectures therefore normally provide direct memory access (DMA) engines to support efficient transfer of data.

However, if object oriented programming (e. g. Java, X10, Chapel) is used, the data to be copied will be *object graphs* consisting of objects pointing to each other. These pointered data structures cannot be directly copied by a DMA engine since all copied pointers would become invalid. Since it is crucial for the performance of object-oriented applications on such architectures, many approaches optimize the transfer or handling of object graphs [13,14,17,22].

As one of them (Pegasus [14]) uses neither near-memory integration, nor hardware acceleration, it serves well as a baseline implementation in the case study. Another state-of-the art implementation of the same mechanism (NEMESYS [22]) on the other hand leverages full near-memory acceleration. Both approaches target a MPSoC architecture as well.

4.2 Prototype and Benchmark Setup of the Case Study

We use a tile-based manycore architecture synthesized on a multi-FPGA system consisting of four Xilinx Virtex-7 2000T FPGAs [20]. The 4×4 tile MPSoC prototype design consists of up to 15 compute tiles and one memory tile, which is located at grid position (1,1). Figure 4 depicts the top-left-most 2×2 part of the whole design.

Each compute tile contains 4 cores (Gaisler SPARC V8 LEON 3 [6,26] processors) with private L1 caches. They are configured in write-through mode and kept intra-tile coherent by a classical bus snooping coherence scheme. The LEON 3 cores further use branch prediction and a floating-point unit. Each compute tile is further equipped with an L2 cache, which caches accesses to the

Table 1. Cache and memory parameters.

Parameter	Value	Parameter	Value
L1-I cache sets	2	LEON 3 freq.	50 MHz
L1-I cache set size	16 kByte	L1 & L2 cache freq.	50 MHz
L1-I cache line size	32 Byte	TLM freq.	50 MHz
L1-D cache sets	2	MEM ctrl freq.	100 MHz
L1-D cache set size	16 kByte	NMCore freq.	50 MHz
L1-D cache line size	16 Byte	NMA freq.	100 MHz
L2 cache sets	4	Local-DMA freq.	100 MHz
L2 cache set size	128 kByte	L1 cache policy	Write-through
L2 cache line size	32 Byte	L1 hit time	1 cycle
L2 cache policy	Write-back	L2 hit time	20 cycles
Tile-local memory (TLM)	8 MByte	L2 miss time	90 cycles
Main MEM	2 GByte	TLM acc. time	20 cycles

remote main memory, and a *tile-local memory* (TLM), which holds the program text, OS data, and temporary user data.

The memory tile is additionally connected to the off-chip DDR-3 main memory and also contains the near-memory core (LEON 3 core with L1 cache) or accelerator (NMA) if present.

A network adapter (NA) connects the tiles to the NoC routers and carries out the remote load-store operations received from the L2 cache back-end. Besides that, the NA can forward remote task invocations and trigger commands to the NMA. Table 1 gives an overview of the core, cache, accelerator and memory configuration parameters.

A distributed operating system [18] which is able to exploit the described hardware features runs on the FPGA prototype. We use the X10 IMSuite benchmarks [9] – a collection of distributed parallel kernels using the PGAS model – in the same configuration as [22].

4.3 X-CEL Applied to Near-Memory Graph Copy

To demonstrate *X-CEL*, we now apply it to the above-mentioned graph copy problem on this tile-based manycore architecture. In this section, we pick one (*MinimumSpanningTree*, MST) out of the twelve IMSuite benchmarks and run it on 15 compute tiles (MST-15) to showcase the different steps of the framework. For a complete study of all twelve benchmarks and different number of compute tiles, refer to the evaluation in Sect. 5.

In **Step 1** of the framework, we identify the memory-intensive graph copy operation as the task of interest (TOI). This task is part of the inter-tile communication routine of the runtime system and therefore occurs during the execution of any kind of parallel application on our system. As outlined in Sect. 3, our goal is to decide

whether to perform this graph copy operation on a core in the receiving compute tile, on the near-memory core, or on a near-memory accelerator (see Fig. 4).

In every variant, the sending processor first needs to ensure that the latest version of the object graph G is in main memory. Since our architecture does not provide inter-tile cache coherence, the processor traverses G and explicitly writes back all necessary cache lines. After the write back on sender side and the invalidation on destination side, both the receiving processor and any near-memory processing elements now have a consistent view of G, and the copying operation can begin.

In the baseline variant, the receiving processor itself does the graph copying [14]. Here, the complete object graph needs to be cloned remotely via the cache hierarchy and the NoC from the source memory partition S to the processor and back to the destination memory partition D. The operation is indicated in Fig. 4 with the beige arrow. This limits performance and pollutes the receiver's caches with the source graph. On the other hand, this approach requires no additional hardware and the newly copied data is available in the receiver's cache right away.

Steps 2.1–2.2. We execute the baseline variant, which yields the following measurements:

	t_{toi}^{base}	t_{other}^{base}	t_{app}^{base}	S_{est}^{1}
MST-15	32.66 s	27.16 s	14.32 s	2.20×

Note, that t_{toi} and t_{other} are accumulated times over all cores, as defined in Fig. 2, while t_{app} is not.

Step 3. As the speedup potential $S_{est}^{1} = 2.20\times > 1 + \epsilon_{sat}$ is satisfyingly large, we go on to analyze the near-memory core variant.

Step 4.1. We implement the NMCore variant, where the memory-intensive graph copy is outsourced to the near-memory core. The near-memory core performs the same software graph copy algorithm as the baseline variant. A negligible effort is required to integrate the near-memory core in the system, schedule the existing graph copy software algorithm on it and maintain consistency with it. The near-memory core is assisted by a state-of-the-art DMA engine for copying larger amounts of consecutive, non-pointered data, if existent. Figure 4 shows the existing system architecture including the near-memory core in green.

Steps 4.2–4.4. The execution and measurement of the NMCore variant yielded the following times:

	t_{toi}^{nmc}	t_{other}^{nmc}	t_{app}^{nmc}	S_{act}^{nmc}	S_{est}^{2}
MST-15	3.09 s	23.00 s	8.94 s	1.60×	1.82×

The actual speedup of the this variant was measured as $S_{act}^{nmc} = 1.60\times$. However, since the NMCore is only utilized during roughly one third ($t_{toi}^{nmc} = 3.09\,s$)

of the total application runtime ($t_{app}^{nmc} = 8.94\,s$), it is far from becoming the bottleneck.

Step 4.5. Based on the measurement results of Step 4.2, we can now do a better estimation of the NMA variant. According to the numbers depicted above, the speedup estimate for the NMA variant compared to the baseline can be calculated to $S_{est}^2 = 1.82\times$.

Step 5. As $S_{est}^2 = 1.82\times > 1.60\times = S_{act}^{nmc}$, we still see potential to achieve a higher speedup by using the near-memory accelerator. However, before this decision is made, all different benchmarks and application scenarios should be evaluated, which is done in Sect. 5. Also the development effort and the required hardware resources (compared to the NMCore) should be considered in this decision.

Step 6. Develop a graph copy NMA as proposed by Rheindt et al. [22]. This implementation uses a near-memory accelerator to perform the graph copy operation which executes the same graph copy functionality as the processor core using a slightly different algorithm that can be performed by a hardware module [22]. The NMA is indicated in purple in Fig. 4. This speeds up the copy operation itself and leaves the processors free for other tasks. However, it requires a tremendous development effort, as well as additional hardware resources of approximately the size of one core. Furthermore, the functionality of the NMA is limited to the graph copy task.

Step 7. The execution and measurement of the NMA variant brought these final results:

	t_{toi}^{NMA}	t_{other}^{NMA}	t_{app}^{NMA}	S_{act}^{NMA}
MST-15	1.65 s	21.38 s	7.69 s	1.86×

The actual measured speedup of the NMA $S_{act}^{NMA} = 1.86\times$ is very close and even slightly larger than the estimate $S_{est}^2 = 1.82\times$. Under the assumption that t_{other} is not effected by the NMA implementation, S_{est}^2 was defined as an upper bound. However, t_{other} decreased to $t_{other}^{NMA} = 21.38\,s$ compared to the baseline implementation's $t_{other}^{base} = 27.16\,s$, which helps to explain the additional improvement compared to the estimate.

5 Evaluation

This section presents the full case study and in-depth analysis for all twelve IMSuite benchmarks and a varying number of compute tiles between one and 15.

We examine the performance predictions of X-CEL in more detail. To this end, we use all benchmarks from the X10-IMSuite, and run them each on differently sized systems (1, 2, 3, 4, 8, 12, and 15 compute tiles). We then compare the two stages of performance predictions made by X-CEL with the actual performance achieved by the NMA.

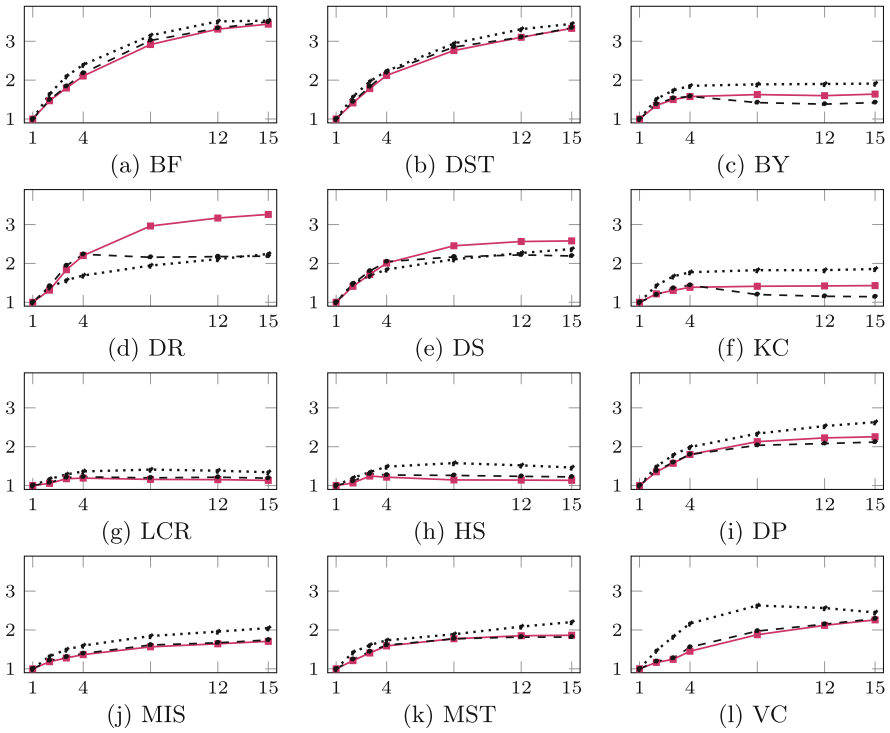

Fig. 5. Individual benchmark speedups of the NMA (—■—) normalized to the baseline for varying number of compute tiles, including S^1_{est}(···•···), S^2_{est}(- -•- -): x-axis: number of computes tiles with four cores each, y-axis: relative speedup.

Figure 5 shows the speedups achieved by the NMA in each benchmark with varying system size, relative to the baseline variant of the same system size. The solid line —■— shows the actual speedups, whereas the dashed lines ···•··· and - -•- - depict S^1_{est} and S^2_{est}, respectively.

For the systems with 3 and 15 compute tiles, we also show the run-times of each variant (Baseline, NMCore, and NMA) in Fig. 6. The dashed lines in these charts represent the run-times predicted by S^1_{est} and S^2_{est}.

The validity of X-CEL rests on two conditions: First, that S^1_{est} gives an indication whether near-memory computing could accelerate the given program at all, and second that S^2_{est} gives an accurate prediction of the run-time achieved by an NMA. We will now examine these two conditions in turn.

We first observe that S^1_{est} usually gives an upper bound on the achievable speedup. That is to say, if S^1_{est} is close to 1, the application will certainly not benefit from near-memory computing.

S^1_{est} only under-estimates the speedup in the DR and DS benchmarks. A closer analysis of the graph copy tasks performed shows a difference to the other benchmarks: DR and DS have many very small graph copy tasks to perform

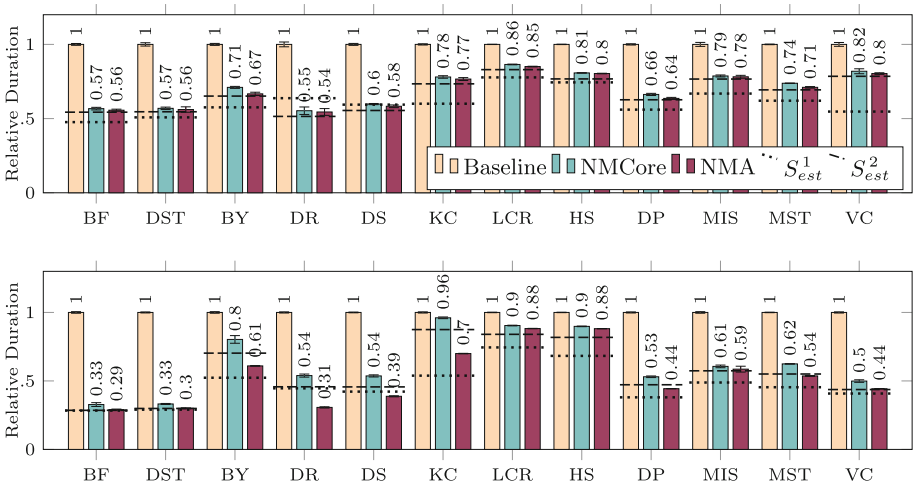

Fig. 6. Runtime measurements of the IMSuite benchmarks with three (Top) and 15 compute tiles (Bottom), respectively.

(e. g., DS transfers a single object of 24 bytes 17 856 times [22]). Thus, the offloading and synchronization overheads come to play a larger role, which our model does not handle as well. Still, we see that S^1_{est} fulfills its function well in most cases.

When examining S^2_{est}, we observe that S^2_{est} approximates the actual speedup well, with a root mean square error of 0.23. Out of all the 84 configurations we evaluated (12 benchmarks × 7 system sizes), in 60 configurations S^2_{est} deviated by less than 5% from the actual speedup.

The other 24 configurations warrant a closer analysis, because too low speedup estimates have a different impact from too high ones: X-CEL uses S^2_{est} as an indication of whether to develop a dedicated hardware accelerator (see Step 5 in Sect. 3). If S^2_{est} turns out to under-estimate the NMA's speedup, this is hardly a problem, because the NMA performs better than expected. On the other hand, if S^2_{est} over-estimates the speedup, the effort spent developing the NMA may have been wasted.

Out of the 24 configuration where S^2_{est} deviates by more than 5 %, it under-estimates the speedup in 14 cases, and over-estimates it in 10. The under-estimates are relatively large in places (up to 32.9 % for DR on 15 compute tiles), but as we have explained, this is not problematic. On the other hand, the over-estimates are at most 10.2 % (HS on 8 compute tiles), and indeed only 5 of the 10 over-estimates are larger than 6 %.

Considering that X-CEL does not need any information about the actual algorithm, the estimates it provides are quite accurate in most cases. Morevoer, if they deviate from the speedup achievable by the NMA, they usually err on the safe side from the developer's point of view.

6 Conclusion

We presented *X-CEL*, a measurement-based method to estimate the potential of near-memory acceleration. It helps to perform an early yet robust estimation whether the development effort of a near-memory accelerator is worthwhile. The two-stage method is based on measurements of an easy-to-integrate near-memory core (near-memory, but no accelerator) variant, which is closer to the target design than the existing baseline implementation (neither near-memory, nor hardware-accelerated). We showcased *X-CEL* with a (near-memory) graph copy problem in a tile-based MPSoC with a set of distributed graph algorithm kernels. An in-depth analysis revealed that the second stage estimate is within 5 % of the actual speedup in 70 % of the configurations. Moreover, it has 36 % higher accuracy than the original estimate.

Future work could refine the estimation model, as well as extend the framework to more case studies.

All in all, we envision *X-CEL* to become an *x-cel*-lent tool in the hand of developers to make sophisticated predictions on the near-memory acceleration potential and thereby avoid unnecessary development effort.

References

1. Ahn, J., Hong, S., Yoo, S., Mutlu, O., Choi, K.: A scalable processing-in-memory accelerator for parallel graph processing. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, 13–17 June 2015, pp. 105–117 (2015). https://doi.org/10.1145/2749469.2750386
2. Altaf, M.S.B., Wood, D.A.: LogCA: a high-level performance model for hardware accelerators. In: 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), pp. 375–388, June 2017. https://doi.org/10.1145/3079856.3080216
3. Arnold, O., Fettweis, G.: Power aware heterogeneous MPSoC with dynamic task scheduling and increased data locality for multiple applications. In: 2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, pp. 110–117, July 2010. https://doi.org/10.1109/ICSAMOS.2010.5642075
4. Carter, N.P., et al.: Runnemede: an architecture for ubiquitous high-performance computing. In: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), pp. 198–209, February 2013. https://doi.org/10.1109/HPCA.2013.6522319
5. Choi, B., et al.: DeNovo: rethinking the memory hierarchy for disciplined parallelism. In: 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT 2011), Galveston, TX, USA, 10–14 October 2011, pp. 155–166 (2011). https://doi.org/10.1109/PACT.2011.21
6. Cobham Gaisler: LEON 3 (2010). http://gaisler.com/index.php/products/processors/leon3
7. Cray Inc.: Chapel language specification (2019). https://chapel-lang.org/docs/_downloads/chapelLanguageSpec.pdf

8. Gries, M.: Methods for evaluating and covering the design space during early design development. Integr. VLSI J. **38**(2), 131–183 (2004). https://doi.org/10.1016/j.vlsi.2004.06.001
9. Gupta, S., Nandivada, V.K.: IMSuite: a benchmark suite for simulating distributed algorithms. J. Parallel Distrib. Comput. **75**, 1–19 (2015). https://doi.org/10.1016/j.jpdc.2014.10.010
10. Kogge, P.: Memory intensive computing, the 3rd wall, and the need for innovation in architecture (2017). https://memsys.io/wp-content/uploads/2017/12/The_Wall.pdf
11. Li, G., Dai, G., Li, S., Wang, Y., Xie, Y.: GraphIA: an in-situ accelerator for large-scale graph processing. In: Proceedings of the International Symposium on Memory Systems (MEMSYS 2018), Old Town Alexandria, VA, USA, 01–04 October 2018, pp. 79–84 (2018). https://doi.org/10.1145/3240302.3240312
12. Lyberis, S., et al.: Formic: cost-efficient and scalable prototyping of manycore architectures. In: 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, pp. 61–64, April 2012
13. Maas, M., Asanović, K., Kubiatowicz, J.: A hardware accelerator for tracing garbage collection. In: Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA 2018), pp. 138–151. IEEE Press, Piscataway (2018). https://doi.org/10.1109/ISCA.2018.00022
14. Mohr, M., Tradowsky, C.: Pegasus: efficient data transfers for PGAS languages on non-cache-coherent many-cores. In: Proceedings of the Conference on Design, Automation & Test in Europe, pp. 1785–1790. European Design and Automation Association (2017)
15. MPI Forum: MPI: a message passing interface standard version 3.1 (2015). https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
16. Neggaz, M.A., Yantir, H.E., Niar, S., Eltawil, A.M., Kurdahi, F.J.: Rapid in-memory matrix multiplication using associative processor. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE 2018), Dresden, Germany, 19–23 March 2018, pp. 985–990 (2018). https://doi.org/10.23919/DATE.2018.8342152
17. Nguyen, K., Fang, L., Navasca, C., Xu, G., Demsky, B., Lu, S.: Skyway: connecting managed heaps in distributed big data systems. In: ACM SIGPLAN Notices, vol. 53, pp. 56–69. ACM (2018)
18. Oechslein, B., et al.: OctoPOS: a parallel operating system for invasive computing. In: McIlroy, R., Sventek, J., Harris, T., Roscoe, T. (eds.) Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA). Sixth International ACM/EuroSys European Conference on Computer Systems (EuroSys), vol. USB Proceedings, pp. 9–14. EuroSys (2011)
19. Pimentel, A.D.: Exploring exploration: a tutorial introduction to embedded systems design space exploration. IEEE Des. Test **34**(1), 77–90 (2017). https://doi.org/10.1109/MDAT.2016.2626445
20. PRO DESIGN Electronic GmbH: FPGA module xc7v2000t (2019). https://www.profpga.com/products/fpga-modules-overview/virtex-7-based/profpga-xc7v2000t
21. Reagen, B., Shao, Y.S., Wei, G.Y., Brooks, D.: Quantifying acceleration: power/performance trade-offs of application kernels in hardware. In: International Symposium on Low Power Electronics and Design (ISLPED) (2013)
22. Rheindt, S., Fried, A., Lenke, O., Nolte, L., Wild, T., Herkersdorf, A.: NEMESYS: near-memory graph copy enhanced system-software. In: Proceedings of the International Symposium on Memory Systems (MEMSYS 2019), pp. 3–18. ACM, New York (2019). https://doi.org/10.1145/3357526.3357545

23. Sangiovanni-Vincentelli, A., Martin, G.: Platform-based design and software design methodology for embedded systems. IEEE Des. Test Comput. **18**(6), 23–33 (2001). https://doi.org/10.1109/54.970421
24. Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O., Grove, D.: X10 language specification (2019). http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf
25. Schuiki, F., Schaffner, M., Gürkaynak, F.K., Benini, L.: A scalable near-memory architecture for training deep neural networks on large in-memory datasets. IEEE Trans. Comput. **68**(4), 484–497 (2019). https://doi.org/10.1109/TC.2018.2876312
26. SPARC Inc.: The SPARC Architecture Manual, Version 8, sav080si9308 edn. (1992)
27. Yitbarek, S.F., Yang, T., Das, R., Austin, T.M.: Exploring specialized near-memory processing for data intensive operations. In: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE 2016), Dresden, Germany, 14–18 March 2016, pp. 1449–1452 (2016)