Embedded Runtime for Reconfigurable Dataflow Graphs on Manycore Architectures

Hugo Miomandre, Julien Hascoët, Karol Desnos, Kevin Martin, Benoît Dupont de Dinechin, Jean-François Nezan

Embedded Runtime for Reconfigurable Dataflow Graphs on Manycore Architectures

Hugo Miomandre[1], Julien Hascoët[1,2], Karol Desnos[1], Kevin Martin[3],
Benoît Dupont de Dinechin[2], Jean-François Nezan[1]
[1] Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164, Rennes, France
[2] Kalray, Montbonnot-Saint-Martin, France
[3] Lab-STICC, Université Bretagne-Sud, CNRS UMR 6285, Lorient, France
email: first.last@{insa-rennes.fr, kalray.eu, univ-ubs.fr}

***Abstract*—Embedded manycore architectures offer energy-efficient super-computing capabilities but are notoriously difficult to program with traditional parallel Application Programming Interfaces (APIs). To address this challenge, dataflow Models of Computation (MoCs) are increasingly used as their high-level of abstraction eases the automation of computation mapping, memory allocation, and communication management. Reconfigurable dataflow is a class of dataflow MoC that fosters a unique trade-off between application dynamicity and predictability. This paper introduces the first embedded runtime manager enabling the execution of reconfigurable dataflow graphs on a Non-Uniform Memory Access (NUMA) architecture. The proposed runtime manager dynamically deploys reconfigurable dataflow graphs on clustered Processing Elements (PEs) through the Networks-on-Chips (NoCs) of the manycore architecture. An open-source implementation on the Kalray MPPA® processor demonstrates the feasibility and the great potential of such a runtime. The first results with an image processing application show a power efficiency 2.5 times better than on a multicore x86 architecture.**

I. Introduction

The ever-increasing performance of embedded systems is driven by the introduction of low-power massively parallel architectures. Unlike classic multicore architectures, which integrate tens of complex high-performance Processing Elements (PEs) in a single chip, the idea behind manycore architectures is to sacrifice the processing capability of individual PEs, in order to gain silicon area to integrate hundreds of PEs [1]. Hence, embedded manycore architectures are now commercialized, offering competitive energy-efficient processing [16], [4], [3].

At the same time, more than 80% of embedded software is still written using procedural languages such as C/C++. Procedural languages are based on control-dependent sequences of imperative instructions manipulating a pool of shared variables. These characteristics make procedural languages inherently ill-suited for programming manycore architectures where hundreds of PEs communicate through complex on-chip interconnects and distributed memory architectures. Hence, a widening *software productivity gap* exists between the developer productivity and the increasing code complexity required to fully exploit parallel computing resources [6].

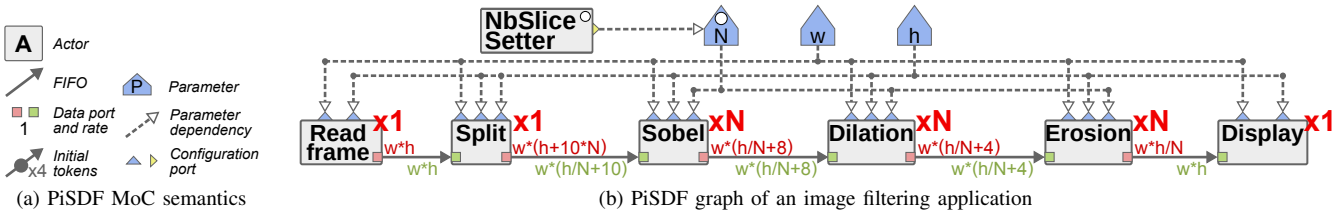
Dataflow Models of Computation (MoCs) have been introduced notably to bridge this software productivity gap. An application specified with a dataflow graph [13] consists of a set of processing entities, named actors, connected by First-In First-Out queues (FIFOs) transmitting data quanta, named data tokens, between actors. An actor starts its preemption-free execution when its input FIFOs contains the required data tokens. The number of data tokens consumed and produced during the execution of an actor is specified by a set of firing rules. The dataflow semantics naturally captures parallel and data-driven computations, which makes dataflow MoCs highly suitable for programming modern parallel architectures.

This paper focuses on reconfigurable dataflow MoCs which allow firing rules of actors to be reconfigured non-deterministically at restricted points in application execution [15]. The software component responsible for managing the graph reconfigurations is called a dataflow runtime. We propose the first implementation of an embedded dataflow runtime, namely the SPIDER runtime [10], for executing reconfigurable dataflow graphs on an off-the-shelf manycore processor. This paper details the new synchronization, memory allocation, and scheduling mechanisms that were designed to manage efficiently the key components of the manycore architecture. An experimental evaluation of the proposed runtime with an image processing application shows a power-efficiency up to 2.5 times better than on a multicore x86 architecture.

Related work on Multiprocessor System-on-Chip (MPSoC) and dataflow MoCs programming are presented in Section II. Section III details the new mechanisms of SPIDER for executing reconfigurable dataflow applications on manycore architectures. Finally, Section IV presents the experimental evaluation of the runtime performance on a commercial manycore processor, and Section V concludes this paper.

II. Context and Related work

Programming multi-/manycore architectures efficiently is a challenge. Although many Application Programming Interfaces (APIs) adopting various MoCs can be found in the literature, no universal parallel programming model fitting all architectures and all applications exist. Pthreads and OpenMP3 are multi-thread programming models for shared memory architectures where all the PEs access a common memory address space. OpenMP4, OpenCL and CUDA are acceleration programming models. The purpose of these models is to offload an application's heavy computations on external



(a) PiSDF MoC semantics (b) PiSDF graph of an image filtering application

Fig. 1: Parameterized and Interfaced Synchronous Dataflow (PiSDF) MoC semantics and example

computing resources, such as external CPUs, GPUs, FPGAs or hardware specific accelerators. The aforementioned programming models often propose dedicated features, like preprocessing directives or specific instructions via intrinsics, to target specific architectures. Thus, their programming requires a deep understanding of the application, the hardware and runtime libraries which take months to master.

Dataflow programming provides the application programmer with a higher level of abstraction. Related work on dataflow programming techniques for manycore architectures focuses on either dynamic [18], [2], [17] or static [7], [14], [9] classes of dataflow MoCs. Reconfigurable dataflow MoC offer a tradeoff between dynamicity and predictability that can be exploited by a runtime manager to verify application properties or to perform optimizations at runtime, like the mapping of actor computations [10].

A. PiSDF MoC

The reconfigurable dataflow MoC studied in this paper is the Parameterized and Interfaced Synchronous Dataflow (PiSDF) MoC [5] whose semantics is depicted in Figure 1. Reconfiguration in the PiSDF MoC is based on parameters, which are nodes of the graph associated to rate configuration parameters. These production and consumption rates of actors can be specified with expressions depending on these parameters. Following PiSDF execution rules [5], an actor may trigger a reconfiguration of the graph topology and intrinsic parallelism by setting a new parameter value at runtime.

Figure 1 depicts the graphical elements of the PiSDF semantics and gives an example of a graph implementing a video filtering algorithm. At each iteration of the graph, which corresponds to the processing of a new frame, the *SetNbSlice* actor triggers a reconfiguration of the data rates by assigning a new value to parameter N. Reconfigurations enable a dynamic variation of the number of parallel executions of the *Sobel*, *Dilation* and *Erosion* actors.

B. SPIDER Runtime

The Synchronous Parameterized and Interfaced Dataflow Embedded Runtime (SPIDER) was originally introduced in [10] as a runtime manager for the execution of PiSDF graphs on heterogeneous MPSoCs.

The internal structure and behavior of SPIDER are depicted in Figure 2. The depicted internal structure consists of two types of processes, each responsible for managing the cores which they are mapped on, and adopting a master/slaves model. The Global Runtime (GRT) is the master of the system: it manages the PiSDF graph topology and takes mapping and scheduling decisions. It is usually implemented over a general purpose core. The GRT can also process actors. The Local Runtimes (LRTs) are lightweight slave processes that execute actors. LRTs can be implemented over heterogeneous types of PEs: general purpose or specialized processors, accelerators.

The execution steps followed by SPIDER to run an actor are numbered in Figure 2. First, the GRT schedules an actor on a PE of the architecture, and sends the execution order through the dedicated *job queue* of the LRT of this PE. A *job* is a message that embeds all data required to execute one instance of an actor: a job ID, location of actor data and code, and which are the preceding actors in graph execution. When an LRT starts an actor execution, it waits for data tokens to be available in the input FIFOs specified in the job message, among a pool of data FIFOs. On actor completion, data tokens are written in output FIFOs, and the LRT sends new parameter values, if any, and execution traces back to the GRT for reconfiguration, monitoring and debugging purposes. Each LRT is associated with a *job counter* that stores the integer job ID of the last executed job. As the job IDs increase monotonically both with scheduling order and data dependencies between jobs, these job counters can be used for synchronization purposes between LRTs, to check whether an LRT already executed a given job.

Open-source implementations of the SPIDER runtime have been proposed for general purpose x86 architectures, Texas Instruments' Keystone digital signal processor architectures, and Xilinx's Zynq heterogeneous platforms [10]. To the best of our knowledge, this paper presents the first implementation of a reconfigurable dataflow runtime on a manycore architecture.

The objective of this paper is neither to advocate the performance of manycore architectures against other modern architectures (GPU, DSP, FPGA) nor to compare the performance of reconfigurable dataflow MoCs with decidable or dynamic MoCs. The objective of this paper is to demonstrate the feasibility and show the potential and flexibility of implementing a runtime for reconfigurable dataflow on a manycore architecture.

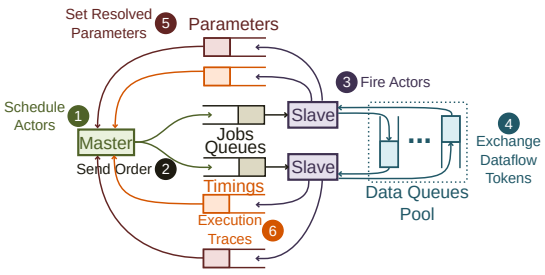


Fig. 2: SPIDER runtime internal structure.

III. SPIDER for Clustered Manycore

A. Architecture Overview

The original implementation of the SPIDER runtime is detailed in [10] targeting embedded shared-memory based MPSoCs. SPIDER operates as an offloading runtime similar to OpenCL or OpenMP4. The master process embeds a PiSDF graph description of the application and deploys at runtime the computation on the slaves processes (master/slaves organization). Such a model offers several advantages such as centralized control, the ability to trigger the offloading of any dataflow graphs depending on external events and manage error recovery on complex parallel systems. The master GRT, which handles graph reconfigurations, mapping and scheduling heuristics, and application profiling/monitoring, requires a substantially larger memory footprint than LRTs to handle these tasks. For instance, on the Kalray Multi-Purpose Processor Array (MPPA)® implementation (see Section IV), the GRT process is mapped on one *IO subsystem*, which is a multicore implementing four VLIW-cores with direct access to the external DDR memory of 4 GBytes. Slave LRTs are mapped on the PEs of the manycore with a dedicated job queue coming from the master.

All runtime mechanisms presented in this section are fully automated in SPIDER and transparent to the application developer. Hence, the application developer specifies the application PiSDF graph, and computation kernels associated to the PiSDF actors and accessing input and output buffers of the application through data pointers. In particular, all data movements are managed implicitly by the distributed SPIDER runtime which performs explicitly all the required communications, using the underlying low-level asynchronous one-sided communication API provided by the Kalray toolchain [8].

B. Software Explicit Network-on-Chip (NoC) Communications

1) Issue: Shared-Memory Based Communications: SPIDER was originally designed for shared memory MPSoCs. Shared memory models are easy to use thanks to the global address space and the provided hardware synchronization mechanisms (atomics). The original synchronization protocol used by LRTs to trigger the execution of actors implements the following sequences. First, when an LRT completes a job, it writes the produced data tokens into shared memory, then sets its job counter to the completed job ID. Second, the LRT can dequeue a new job to process from its job queue. Third, before firing the actor, it needs to synchronize with the completion of preceding actors executions. For that purpose, job messages contain the job ID of each preceding actor and the ID of each LRT that executed these jobs. Thus the LRT will compare the expected job counter values, given by the job IDs, to the actual job counter values of the specified remote LRTs. For convenience, job counter values of all LRTs are stored in a single array, accessible to all PEs. Such synchronization mechanisms are simple to implement for shared memory architectures where they consist of a comparison done by *Load/Store* instructions and management of the memory consistency using full memory barrier to prevent from data race. On previous SPIDER implementations [10], when targeting the Texas Instruments' Keystone II architecture, this synchronization mechanism used specific hardware queues to manage data dependencies. It means that the SPIDER architecture provides a proper partitioning of the key actions of the dataflow runtime, allowing them to be accelerated by hardware specific features of the targeted platform.

2) Solution: Distributed Synchronization Algorithm: The objective of the new synchronization algorithm is to both distribute the control of synchronizations and bound the number of NoC communications per data dependency of the firing of an actor. Therefore, the proposed algorithm builds on the observer design pattern, where the *observers* are the LRTs waiting for completion of a preceding actor, and the *notifier* is the LRT executing this actor. The operating principle of the algorithm consists of the three following actions:

Register: When an LRT pops a new job to execute in its queue, it scans the set of preceding actors (information coming from the master in the job message), and sends a *notification request* to each LRT executing the preceding actors. A *notification request* encapsulates both the ID of its sender LRT, and the ID of the awaited job.

Notify: On job completion, an LRT updates its *job counter* then answers to all LRTs with a pending *notification request* with an awaited ID lower than the new *job counter* value.

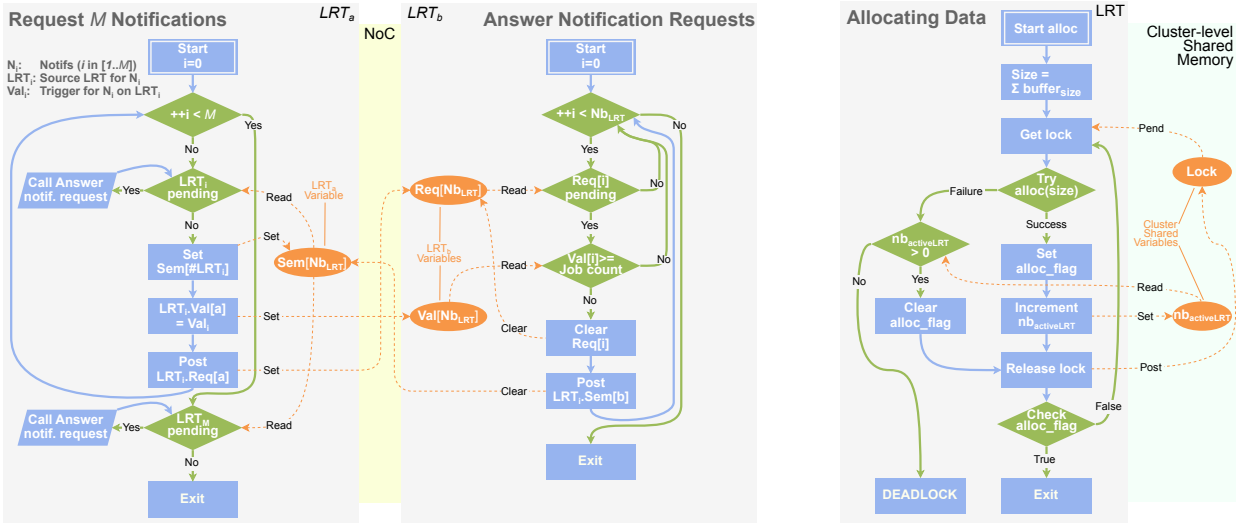
Peek: Optionally and for optimization purpose, after sending all its *notification requests*, an LRT can check, once and on its own, the *job counter* values of all LRTs that have not yet answered. The objective of a peek, which consists of a remote 8-byte load in a distant memory over the NoC, is to avoid waiting for a notification from a busy LRT whose *job counter* is already greater than the awaited value.

Considering the above description of actions, each data dependency between two actors requires at most five communications through the NoC: two to send a notification request, one to send a notification, and two for a peek. Hence, a finite number of NoC communications per dependency is needed, thus fulfilling the communication bounding objective.

The algorithm flow-chart diagram in Figure 3a details the distributed synchronization protocol. It implements an all-to-all (LRTs) synchronization mechanism. The left part of the diagram describes the observer LRT_a popping a new job from its *job queue* and the right part explains the notification sequence when an LRT_b processes pending notification requests. For simplification purposes the *peek* action was omitted. The protocol requires three synchronization vectors for each LRT, allocated in the local scratchpad memory of the PE: Sem contains LRTs IDs of sent unanswered *notification request*, Req registers LRTs IDs of received *notification requests*, and Val contains job counter values awaited by LRTs registered in Req. The size of each array corresponds to the total number of LRTs in the system Nb_{LRT}.

C. Runtime Scheduling for a Large Number of PEs

1) Issue: Prohibitive Complexity and Footprint: The original scheduler implemented in SPIDER is a LIST scheduling



(a) Actor synchronization. As many requests are sent as the number of input FIFO of the next actor.

(b) Cluster-level memory allocation.

Fig. 3: LRT algorithms for memory allocation and actor synchronization.

heuristic described in [11]. When parameters of a graph are dynamically configured, the GRT analyses the data exchange rates in the PiSDF graph and generates an equivalent Directed Acyclic Graph (DAG) graph, exposing explicitly all data parallelism. Actors of the DAG are obtained by duplicating actors of the PiSDF graph as many times as their number of firing, themselves obtained analytically from data consumption and production rates [12]. Then, the GRT handles the mapping and scheduling of each actor, taking into account the dependencies of the DAG and mapping constraints if any.

The issue with the LIST scheduler is that its complexity becomes prohibitively large when targeting a processor with hundreds of PEs. Indeed, its complexity is given by $O(A.log(A)+P.(A+E))$ [11], where A and E are the number of actors and dependencies in the DAG, and P is the number of PEs. Manycore architectures implement hundreds of PEs and require a lot of application parallelism. Therefore, the number of DAG actors to be scheduled in parallel increases roughly linearly with the number of PEs. Consequently, the complexity of the LIST scheduling tends to increase quadratically with the number of PEs, making it a bottleneck for runtime scheduling.

2) Solution: Lightweight Scheduling: We replaced the original LIST scheduler with a less complex scheduling algorithm based on a specialized Round Robin (RR) heuristics. First, the new algorithm was optimized to reduce the memory footprint and the latency of job scheduling decisions. The classical RR heuristic iterates circularly on a list of LRTs, and sends jobs to the first available LRT. This heuristic lowers the scheduling complexity down to $O(A + E)$, as a topological ordering of actors is required. However we find out that the evaluation of the actor execution time and the job fairness distribution to LRTs were no longer required, as a lot of LRTs are available and one is always is ready.

Second, optimization regarding memory usage consists of interleaving the PEs from different clusters in the list on which the RR algorithm iterates. In each cluster, several PEs share a local scratchpad memory and a NoC interface. The objective of this optimization is to prevent too many jobs from being sent simultaneously to PEs on the same cluster. Multiple jobs starting their execution will simultaneously try to synchronize themselves with their predecessor actors, and allocate cluster memory for their input and output buffers (see Section III-D). Therefore, scattering jobs among clusters will prevent/reduce the waiting time of PEs to access to shared cluster resources.

Third, the specialized RR algorithm features a flow control mechanism to avoid the corruption of LRT job queues. The GRT is authorized to send a job to an LRT only if the targeted remote job queue contains enough space to receive the job. For that purpose, LRTs send their job counter value to the GRT on job completion. The statically configurable size of *job queues* has to be large enough to prevent starvation of the LRTs, but as small as possible to limit its memory footprint.

Finally, the RR scheduler was enhanced to take into account the available memory in the compute cluster. When scheduling an actor, the required amount of memory for its execution is computed, and the scheduler avoids sending it to an LRT that will not be able to run it, thus creating a deadlock. The measurement of available space in cluster memory associated to each PE is done at compile time, but can be automated during the initialization boot sequence of SPIDER.

D. Distributed Scratchpad Memory Allocation

1) Issue: Scratchpad Memories instead of Caches: When an LRT pops a new job, it needs to allocate memory to accommodate the input and output buffers of the corresponding actor. As the original SPIDER implementation was implemented for shared-memory based architectures, where PEs (LRTs) access the main memory (usually DDR technology) through their data cache, SPIDER used single global memory allocator. Data pointers on globally allocated data tokens of the FIFOs are sent to the LRT job queues, and LRTs are able to access data tokens using simple *Load/Store* instructions, implicitly supported by their data cache, refilling from the main memory. However,

memory consistency operations at synchronization points are still required (full memory barrier).

On a scratchpad memory based clustered manycore architecture, the memory in the cluster needs to be allocated by the software. Furthermore, a scratchpad memory is limited and distributed, thus making the control software more complex and error-prone. In such a case, exhausting memory resources may happen frequently but does not necessarily requires terminating the application. For instance, a memory allocation may fail for an actor A when another actor B executed in the same cluster uses all the available scratchpad memory. On completion of the execution of actor B, its memory can be reused, possibly after sending output buffers back to the main external memory. Then actor A may successfully perform its memory allocation in the scratchpad memory.

2) Solution: Thread-safe Scratchpad Memory Allocator: The flow-chart in Figure 3b describes a new algorithm for managing the memory allocation in the scratchpad memory of a clustered manycore architecture. This allocation procedure ensures that all job scheduled on an LRT running in a cluster will succeed in allocating the required memory, as long as their required memory does not exceed the maximum capacity of the cluster memory space (see Section III-C). When all firing conditions of a mapped actor are fulfilled, the LRT attempts to allocate memory using this algorithm.

As multiple PEs may compete for scratchpad memory space, a cluster level $lock$, based on atomic instructions, is required to prevent the memory allocator from data structure corruption. This critical section of this algorithm also protects a shared counter $Nb_{ActiveLRT}$ that represents the number of actors currently executed in the cluster. If the number of active LRT is greater than zero when a memory allocation fails, then the LRT should release the $lock$, and try again later. If not, a deadlock is detected as no other LRT is currently using cluster memory, and there is no reason for more memory to be available during a future allocation attempt. The deadlock detection is an expendable safety feature if, as presented in Section III-C, the scheduling algorithm is aware of the maximum available space in cluster memories.

The deallocation procedure on actor completion consists of taking the $lock$, freeing the data, decrementing the $Nb_{ActiveLRT}$ counter, and releasing the $lock$, along with full memory barrier to manage the cache coherence in the cluster.

IV. Experimental Validation

A. Experimental Setup

The new mechanisms introduced in this paper, which allow the execution of the open-source SPIDER runtime for scratchpad memory based manycore architectures, were implemented on the Kalray MPPA® processor [4].

1) Kalray's MPPA® Architecture: MPPA® implements hierarchical computing resources featuring 18 Non-Uniform Memory Access (NUMA) nodes, 16 compute clusters and 2 IOs subsystems, interconnected with a NoC. Each compute cluster consists of 16 VLIW cores and 2 MB of *scratchpad-only memory*. The MPPA is a DMA-based manycore architecture. All computations are driven by DMA data transfers over the NoC and the software runtime is in charge of configuring the DMA NoC interface. Indeed the compute clusters can access the main memory (DDR) and the memory of other compute clusters **only** through the NoC and explicitly by software. Because of these features, programming efficiently the MPPA is a challenge as all communications have to be managed explicitly by the software, and thus written by the developer.

2) Image Filtering Application: The image processing PiSDF graph used to assess the functioning of the SPIDER manycore implementation is presented in Figure 1. The purpose of this application is to apply a commonly used *Sobel* image filter, and two morphological operators, an *Erosion* and a *Dilation*, in order to detect the edges of the processed 2D image. In Figure 1, notations *xN* next to actors denotes the number of (parallel) executions of each actor during a graph iteration. In this example, the *Sobel*, *Dilation* and *Erosion* actors are parallel and the other actors are sequential. The reconfigurable parameter *N* represents the number of slices the input image is divided into for parallel processing. In this example, actor *NbSliceSetter* automatically searches for the value of N that maximizes application performance by monitoring the system. The selected application allows to demonstrate the feasibility of a reconfigurable dataflow runtime on a manycore architecture, and evaluate the runtime overhead in comparison with a static execution.

B. Results

1) LRT Memory Footprint: A first porting of the SPIDER LRT runtime on MPPA® PEs resulted in a memory footprint of 82kB. Mapping an LRT process on each 16 PEs of a cluster requires 1.28MB out of the 2MB local memory. Considering that 620kB of memory is reserved for system services, only 116kB of memory would remain available to store processed data. 116kB allows executing the filtering application on image resolutions up to *720p* with 256 slices processed in parallel.

Thanks to the scheduling mechanism limiting the number of jobs sent to individual LRTs, the size of the *job queues* was reduced to three slots, enabling job buffering and leading to an LRT memory footprint of 6.5kB. In this configuration, 1.29MB of cluster memory remains available for storing processed data. This new configuration leaves enough space to allow the processing of ultra high-resolution (4K) images, equivalent to nine *720p* images, on the MPPA®.

2) Performance and SPIDER Overhead: Figure 4 shows the performance obtained when executing the image filtering PiSDF graph on the MPPA® for 4K images. Application performance, expressed as the number of processed frames per second (fps), is plotted for a varying number of active clusters, and a varying number of active PEs per cluster. The sequential performance on a single PE is 0.13 fps. When using the 256 PE of the compute clusters, a throughput of 2.81 fps is reached, which represents a speed-up of 22 compared to the sequential execution. During the processing of each

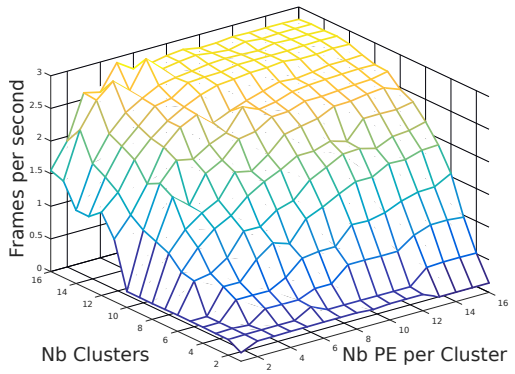


Fig. 4: Application performance on a 4K video

frame (0.36s), only 8% of this latency is due exclusively to GRT computations. Hence, actor computations and NoC communications are responsible for 92% of the latency. These results are decent considering that, according to Amdahl's law, the theoretical speedup for this application on 256 cores is 28, which is an optimistic prediction as the communications overhead is ignored. Calculation of Amdahl's law is based on a measurement of T_{par} the summed execution time of the image processing actors (Sobel, Dilation, Erosion) on the compute clusters, and T_{seq} the summed execution time of all other actors on the IO subsystem. The theoretical speedup S on 256 cores is given by $S = ((1 - \frac{T_{par}}{(T_{seq}+T_{par})}) + \frac{256 \times T_{par}}{T_{seq}+T_{par}})^{-1} = 28$.

In [9], the authors evaluate the performance of a static version of the PiSDF graph from Figure 1. In the static version, N is fixed and all mapping and scheduling are done at compile time for VGA videos (640x480). The top performance obtained for the static execution is 217 fps. For an identical video resolution, the reconfigurable PiSDF graph executed with SPIDER peaks at 47 fps. Besides the SPIDER runtime overhead, the difference between the performance of the static and reconfigurable executions is mostly due to the lack of memory optimization in the reconfigurable implementation. In the reconfigurable version, many `memcpy` calls are issued to create the image slices in the *Split* actor and to merge processed slices into a contiguous buffer before *Display*. Thanks to compile time optimizations, these `memcpy` calls are replaced with pointer operations in the static version reducing the memory bandwidth drastically by a factor of 3.

When using a standard thread-based implementation of SPIDER on an Intel Xeon E5-1650 with 6 hyper-threaded x86 cores clocked at 3.60GHz, the processing of a 4K video with the same PiSDF graph reaches 11.40 fps, using 95% of all CPU time. Although the performance on the Xeon processor is almost 4 times better, this processor dissipates on average 10 times more power than the MPPA®. Hence, the execution on the MPPA® is approximately 2.5 more energy efficient than on the Xeon.

V. Conclusion

This paper presents the first implementation of a runtime manager for reconfigurable dataflow graphs on embedded manycore architectures. The proposed runtime is based on new scheduling, synchronization, and memory allocation algorithms specifically designed for clustered architectures. Experiments on an MPPA® processor demonstrate the feasibility of such a runtime, and its potential in terms of application performance and energy efficiency. Future work includes the development of lightweight scheduling strategies improving the data locality of computations on clustered PEs.

Acknowledgements

This work was partially supported by the MORDRED Project, funded by the GdR ISIS of the CNRS.

References

[1] S. Borkar, "Thousand core chips: a technology perspective," in *Design Automation Conference (DAC)*. ACM, 2007, pp. 746–749.

[2] S. C. Brunet, C. Alberti, M. Mattavelli, and J. W. Janneck, "Design space exploration of high level stream programs on parallel architectures: a focus on the buffer size minimization and optimization problem," in *ISPA*. IEEE, 2013, pp. 738–743.

[3] F. Conti, D. Palossi, A. Marongiu, D. Rossi, and L. Benini, "Enabling the heterogeneous accelerator model on ultra-low power microcontroller platforms," in *Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2016, pp. 1201–1206.

[4] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications," in *High Performance Extreme Computing Conference (HPEC)*. IEEE, 2013, pp. 1–6.

[5] K. Desnos, M. Pelcat, J.-F. Nezan, S. Bhattacharyya, and S. Aridhi, "PiMM: Parameterized and interfaced dataflow meta-model for MPSoCs runtime reconfiguration," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. IEEE, 2013, pp. 41–48.

[6] W. Ecker, W. Müller, and R. Dömer, *Hardware-dependent Software*. Springer, 2009.

[7] T. Goubier, R. Sirdey, S. Louise, and V. David, "SigmaC: A programming model and language for embedded manycores," in *Algorithms and Architectures for Parallel Processing*, ser. Lecture Notes in Computer Science. Springer, 2011, pp. 385–394.

[8] J. Hascoët, B. D. de Dinechin, P. G. Massas, and M. Q. Ho, "Asynchronous one-sided communications and synchronizations for a clustered manycore processor," in *Embedded Systems for Real-Time Multimedia*. IEEE/ACM, 2017.

[9] J. Hascoët, K. Desnos, J.-F. Nezan, and B. D. de Dinechin, "Hierarchical dataflow model for efficient programming of clustered manycore processors," in *Application-specific Systems, Architectures and Processors (ASAP)*, 2017.

[10] J. Heulot, M. Pelcat, K. Desnos, J. F. Nezan, and S. Aridhi, "Spider: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps," in *Embedded Design in Education and Research Conference (EDERC)*, Sept 2014, pp. 167–171.

[11] Y.-K. Kwok, "High-performance algorithms for compile-time scheduling of parallel processors," Ph.D. dissertation, 1997.

[12] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, sept. 1987.

[13] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, 1995.

[14] Y. Lesparre, A. Munier-Kordon, and J. M. Delosme, "Evaluation of synchronous dataflow graph mappings onto distributed memory architectures," in *Digital System Design (DSD)*. Euromicro, Aug 2016, pp. 146–153.

[15] S. Neuendorffer and E. Lee, "Hierarchical reconfiguration of dataflow models," in *MEMOCODE*, 2004.

[16] A. Olofsson, "Epiphany-v: a 1024 processor 64-bit risc system-on-chip," *arXiv preprint arXiv:1610.01832*, 2016.

[17] S. Roloff, A. Pöppl, T. Schwarzer, S. Wildermann, M. Bader, M. Glaß, F. Hannig, and J. Teich, "Actorx10: An actor library for x10," in *ACM SIGPLAN Workshop on X10*. ACM, 2016, pp. 24–29.

[18] A. Stoutchinin and L. Benini, "Stream drive: A dynamic dataflow framework for clustered embedded architectures," in *Computing Frontiers*. ACM, 2017, pp. 1–8.