Implicit Data-Parallelism in Kahn Process Networks: Bridging the MacQueen Gap

Robert Khasanov
Chair for Compiler Construction
TU Dresden
Dresden, Germany
robert.khasanov@tu-dresden.de

Andrés Goens
Chair for Compiler Construction
TU Dresden
Dresden, Germany
andres.goens@tu-dresden.de

Jeronimo Castrillon
Chair for Compiler Construction
TU Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

ABSTRACT

Modern embedded systems are rapidly increasing their complexity, both in terms of numbers of cores, as well as heterogeneity. To generate efficient code for these systems, it is common to leverage formal models of computation. Among these, the dataflow model of Kahn Process Networks (KPN) is widespread because it is expressive but guarantees a deterministic execution. However, the KPN model is ill-suited to expose data-level parallelism, since this has to be made explicit in the process network. This is aggravated by the fact that its most common execution model, Kahn-MacQueen, poses restrictive conditions on the scheduling of data-parallel processes, leading to an inefficient execution. In this paper we present a novel extension to the KPN model and a relaxed execution strategy that addresses this problem, while keeping the deterministic KPN semantics. It improves run-time adaptivity in malleable way and provides implicit parallelism. We evaluate our approach on two architectures, improving the performance of a benchmark by up to 25.6 % on an Intel chip with hyper-threading, and by up to 78.0 % on a heterogeneous embedded ARM big.LITTLE architecture.

CCS CONCEPTS

• **Theory of computation → Parallel computing models**; • **Computing methodologies** → *Parallel programming languages*;

KEYWORDS

Streaming applications, process networks, adaptivity, MPSoC, heterogeneous

ACM Reference Format:
Robert Khasanov, Andrés Goens, and Jeronimo Castrillon. 2018. Implicit Data-Parallelism in Kahn Process Networks: Bridging the MacQueen Gap. In *PARMA-DITAM '18: 9th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and 7th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms, January 23, 2018, Manchester, United Kingdom.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3183767.3183790

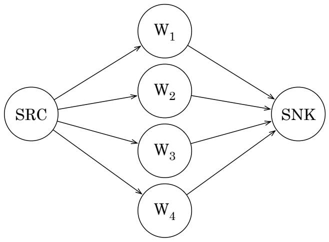

Figure 1: An example of a KPN Application: Calculating the Mandelbrot Set.

1 INTRODUCTION

Most modern embedded systems consist of several cores, and the number of cores in such systems only continues to grow. In fact, while commercially widespread systems today dwell in the lower ranges of four or eight cores, some recent many-core systems are already reaching numbers in the thousands [18]. Other platforms, like those based on ARM big.LITTLE™ [7], also consist of heterogeneous cores that offer better energy-effiency or in some cases feature specialized cores for efficient execution of particular types of code. These features offer high amounts of resources for applications to leverage, allowing them to use a high number of different cores, or share resources with other applications. It is therefore difficult to program these systems.

The current trend in compilers and programming languages is to address this problem by using specific application models that can help exploit the advantages of hardware platforms while hiding low-level details from a programmer.

A typical application model in the embedded domain is that of dataflow process networks [15], where continuously running processes communicate through FIFO channels. A particularly important model in this family is that of Kahn Process Networks (KPN) [9], which is the most general model which guarantees a deterministic execution [15], a property that is particularly desirable in embedded systems [12]. This model is well-suited for streaming applications, such as audio/video encoders or image recognition.

As an example, consider the KPN application in Figure 1. It depicts a network to calculate the Mandelbrot set. In it, a source (SRC) divides the complex plane in lines and sends those lines to four different workers (W1-W4). The workers calculate the convergence or divergence of points, and send them to a sink (SNK), which reassembles the lines to output the set.

Besides showcasing how the KPN model can be used to express parallelism, this example also shows two main difficulties with the

PARMA-DITAM '18, January 23, 2018, Manchester, United Kingdom

ACM ISBN 978-1-4503-6444-7/18/01…$15.00
https://doi.org/10.1145/3183767.3183790

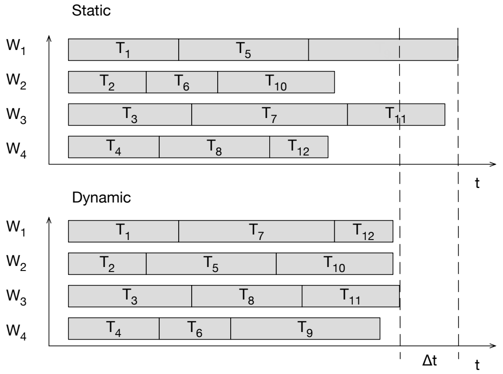

Figure 2: An Example of Sub-Optimal Scheduling in the Static KPN Model.

KPN model. The first is that KPN lacks implicit data-level parallelism. In order to express it, application developers have to explicitly specify a parallel topology of the process network, as well as the workload distribution, at compile-time (W1-W4 in the example). Similarly, another difficulty is that most KPN-based models use the Kahn-MacQueen blocking-read semantics, which restrict the order of execution. More precisely, when a process reads from various channels in this execution model, it has to have a static order to read from. Even if, at runtime, all other channels have data which could be processed, if the first channel to be read blocks because it does not have any tokens, the whole process will block and execution will stop. This also means that workers in explicit data parallelism have a static order for packages, and can lead to sub-optimal scheduling, as can be seen in the example in Figure 2. The figure shows a Gantt chart of the four workers calculating 12 different tokens. Because of the static, deterministic nature of the KPN model, the workload is distributed poorly between workers.

These limitations have several additional consequences: First, they hamper the portability of the application, since programmers have to generate the network topology for each platform they are targeting. Second, since the amount of available resources in a system may vary due to other applications, explicit network topologies cannot adapt to these variations. Such applications are also called *rigid* [4], i.e. they run on a certain number of processors specified by the user and this number does not change at run-time. At the same time, heterogeneous platforms and dynamic frequency-voltage scaling of cores make static strategies for workload distribution inefficient. All of these make it harder to design a holistic approach to determine an optimal amount of parallelism, because the model cannot capture the semantic information that a process can be executed in a data-parallel fashion. Thus, tuning data-parallelism cannot be integrated in the compiler. This task necessarily falls on the shoulders of the programmer.

Since data parallelism is a very common type of parallelism, these issues need to be addressed in order to fully leverage the capabilities of modern systems.

In this paper we present such an extension of implicit parallelism to the KPN execution model, which preserves the formal semantics of KPN [9]. We also show how we can relax the Kahn-MacQueen execution semantics [10], with blocking reads and writes, and return to the original semantics of a Kahn Process Network [9] which are strictly more expressive [13]. We call this gap in execution semantics, “the MacQueen Gap”. We present an implementation of these extensions as efficient channel libraries using SLX [22], a commercial spin-off of the MAPS framework [2]. We show how our extended model has a higher degree of adaptability, especially on heterogeneous platforms.

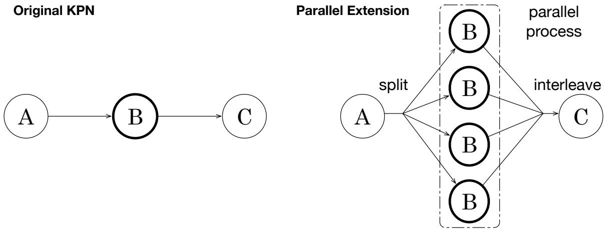

Figure 3: The Parallel Process KPN Extension.

The rest of the paper is structured as follows: Section 2 describes the extensions to the KPN model and their implementation. These are evaluated in Section 3. Then, related work is discussed in Section 4, and finally, Section 5 concludes the paper.

2 EXTENSION OF PARALLEL PROCESSES

In this section we present our extended execution model, based on KPN and the typical Kahn-MacQueen execution model. Our extension consists of several parts: a special type of process, a *parallel process*, with a corresponding type of *parallel channel*.

2.1 Parallel Processes

For our extended execution model we first introduce a new type of process, a *parallel process*. A parallel process is a process that can be duplicated and which workload can thus be distributed among diverse instances of the original process. Semantically, the duplicated processes behave as a regular process. Figure 3 depicts this, by showing an example where the process B is marked as parallel and duplicated into four instances which can be executed in parallel.

Not every process can be marked as parallel. We must introduce special requirements for a process to be annotated as parallel. Consider the formal semantics of KPN [9]. A KPN process can be seen as a continuous function

$$f : D_1^{\omega} \times \ldots \times D_k^{\omega} \to D_1'^{\omega} \times \ldots \times D_l'^{\omega},$$

for the sequence domains $D_i^{\omega}, D_j'^{\omega}, i = 1, \ldots, k, j = 1, \ldots, l$.

To mark a process as parallel, it needs to fulfill two conditions

(1) It must be possible to consider the process as an actor with constant rates, in the sense of dataflow with firing [15]. Formally, this means that f can be specified as a function

$$f : I^{\omega} \to O^{\omega},$$

for a subset $I \subset D_1^{*} \times \ldots \times D_k^{*}$ and a subset $O \subset D_1'^{*} \times \ldots \times D_l'^{*}$, where $*$ denotes the Kleenee closure, such that the length of each of the components for all elements in each of I and in O is constant and finite, i.e.

$$\text{For all } i = 1, \ldots, k : |\{\text{length}(w_i) \mid (w_1, \ldots, w_k) = w \in I\}| = 1,$$

$$\text{For all } j = 1, \ldots, l : |\{\text{length}(w_j) \mid (w_1, \ldots, w_l) = w \in O\}| = 1.$$

Note that the input and output rates themselves must not be equal, just constant (and finite) for every channel.

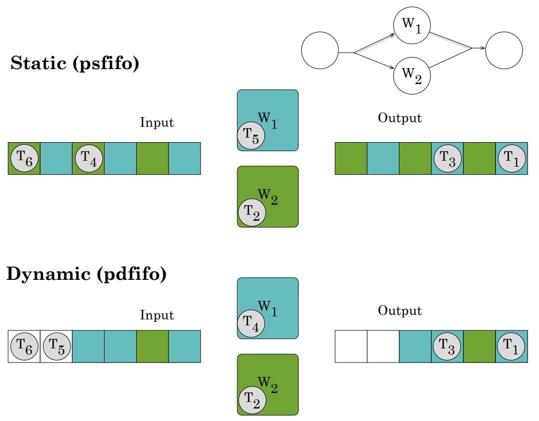

Figure 4: The Different Execution Strategies for Split-Interleave Primitives.

(2) The process must be *stateless*. Formally, this means that, for any two sequences $v, w \in I^\omega$, where v is a suffix of w, i.e. $w = w'v$ for some $w' \in I^\omega$, it holds that $f(w) = f(w')f(v)$. In other words, that the output only depends on the corresponding input in a firing, not on any previous firings.

By considering dataflow actors with firing as KPN [15], these conditions would be fulfilled by a stateless Synchronous Data Flow (SDF) actor [14]. Note that the conditions are sufficient but not necessary. For example, our approach could, in principle, be extended to CSDF [1] actors. The requirement of statelessness is important since in that case we ensure that there is no dependency between different iterations.

2.2 Parallel Channels

When a parallel process expands to multiple instances, these instances have to be connected with the predecessors and successors of the original process. Moreover, the order of the tokens must be the same, to respect the KPN semantics.

In this paper we introduce *split* and *interleave* primitives. Split primitives distribute tokens among instances of the parallel processes and interleave primitives gather all tokens from all channels coming from parallel workers into a single channel. This interleave functionality is similar to the well-known *deterministic merge*, albeit less strict, as will be seen in this section. Thus, they are necessary in order to implement data parallelism with parallel processes. The split-interleave functionality can be included in three ways, as an additional process, inside the predecessors and successors of parallel process, or as part of the channels themselves. Figure 3 depicts this last option.

If the primitives are included in processes, either as a standalone process or in the successors/predecessors, then they come with some additional restrictions. In order to keep tokens in the same order, all split and interleave processes must be coordinated with the same strategy. Additionally, a particular drawback of using split and interleave nodes is the usage of additional memory for each channel, as well as the overhead of copying tokens.

Thus, for our extended execution model, we introduce special split and interleave channels. In this special type of FIFO channels, all tokens in the channel are distributed among the parallel instances internally. When a parallel worker process reads data, a worker identifier is used to determine the first token assigned to this worker. When a parallel worker process writes data, its identifier is again used to determine the position in the output channel. Thus, the consumer of the output channel reads tokens in the same order as if it would have been a single worker. The top part of Figure 4 depicts this for a static assignment of tokens to workers, in a round robin fashion. We call this channel type `psfifo`, or parallel static FIFO. It is similar to the approach with special processes, but avoids copying data. In the figure, tokens T_1 and T_3 have already been processed by the worker W_1, which is now processing T_5. On the other hand, W_2 is still executing T_2. Thus, besides automatically adding data parallelism, `psfifo` channels improve execution by relaxing the blocking reads and writes.

Finally, the `psfifo` channel can be relaxed further, in an implementation of split and interleave channels which we call parallel dynamic FIFO, or `pdfifo`. As in a `psfifo`, here we have only a single channel. In this case, however, there is no static assignment of tokens. In order to read and write to channels in order, all tokens are associated with a firing identifier. At the beginning of the firing, a parallel process gets a firing id. Then, the process uses this identifier to determine an exact memory address of data for reading and writing. In Figure 4, since W_2 takes a long time executing T_2, tokens T_3 and T_4 are assigned dynamically to W_1. This will generally lead to a better dynamic scheduling as depicted in Figure 2.

While they allow a more flexible execution, as described, dynamic channels currently only work on a shared memory model. In further work we plan to study how to implement this on a distributed memory model as well.

2.3 Adaptivity

A regular KPN with static data-parallelism is called *rigid*, since it is specified by the user at compile-time. The `psfifo` version improves this by allowing the system to decide. This property is called *moldability*. The `pdfifo` channels are even more adaptable. This adaptivity comes at two distinct levels: workload distribution among parallel workers, and the possibility to change the number of parallel workers. This latter property is called *malleability*.

Since `pdfifo` channels do not statically distribute tokens among parallel workers, the distribution of the tokens may adapt and vary at run-time. Workers adapt the distribution of the workload implicitly: once a worker finishes a firing it executes the next token. It follows that a faster worker will execute more firings, and a slower worker will execute less number of firings. While this still does not ensure an optimal scheduling, it should improve it in most cases (see Figures 2 and 4). This is especially beneficial in scenarios where parallel workers are running on heterogeneous platforms or when one of the cores is shared with another application. Similarly, this is also beneficial when the workload depends on the data, and thus, the execution time of a single firing varies significantly between firings.

The other level of adaptivity is how the application may change the number of parallel workers without changing split-interleave strategies. In fact, `pdfifo` channels are agnostic to the number of parallel workers. Moreover, parallel workers may start or stop operating without interrupting the rest of the application, including other parallel workers. Note that stopping can only safely occur after finishing a firing, not during one.

2.4 Semantics

Having described the extension above it should be intuitively clear that the extended execution model preserves the semantics of the original KPN program, where a single process is placed instead of parallel workers. Both `psfifo` and `pdfifo` channels and parallel tokens are just an implementation of the same f as before, in the KPN semantics. By using the properties required in Section 2.1, we can ensure that f can be executed in different "firings" in parallel, and the channel implementations just ensure the order is preserved.

A particularly useful property of this is that all assumptions based solely on the KPN semantics still hold. This includes determinism, as demonstrated in the original KPN paper [9], as well as many assumptions that are implicit in many analysis flows [2, 19, 20].

The execution, on the other hand, effectively bridges the MacQueen gap between KPN semantics and the Kahn-MacQueen execution model with blocking reads and writes. To see why this is the case, consider Figure 4. In the Kahn-MacQueen execution model, the application would block in both cases after W_1 has finished executing T_1 and T_3, since from the blocking semantics the network would continue to wait for W_2 to finish with T_2, before being able to process the next token. Additionally, the `pdfifo` channels go even beyond this, since they allow dynamic changes to the workload distribution. In particular, the workload distribution in `pdfifo` channels is a controlled non-deterministic procedure, although the execution of the whole parallel process remains deterministic.

3 EVALUATION

In this section, we evaluate our extension of implicit data-parallism and the parallel channels, with a particular focus on the adaptivity of our approach.

3.1 Experimental setup

Hardware setup. We conducted the evaluation on two platforms: Intel Core i7-4790 and Hardkernel Odroid XU4, a modern off-the-shelf heterogeneous multicore system. Intel Core i7-4790 is a quad-core 64 bit CPU that runs 8 threads simultaneously using hyper-threading technology. The CPU is clocked at 3.60 GHz and has a cache of 8 Mb. Odroid XU4 features Exynos 5422 big.LITTLE chip with four Cortex-A15 cores and four Cortex-A7, with 2 Gb of LPDDR3 RAM. The frequency of the A7 (little) cluster ranges from 200 MHz to 1.4 GHz and that of the A15 (big) cluster ranges from 200 MHz to 2.0 GHz. During the experiments the *performance* governor was activated.

Benchmark description. In our experiments we used as benchmark a calculation of the Mandelbrot set. The process network is shown on Figure 1 and is explained in the introduction. The parallel workers calculate points in an iterative fashion, where the number of iterations depends on the point itself. For this reason, this benchmark is well-suited for investigating data-parallelism more thoroughly, since in contrast to many benchmarks based on linear algebra, the execution time of the algorithm is data-dependent. Figure 5 shows how many iterations it performs for each point, which incidentally shows the Mandelbrot set as well.

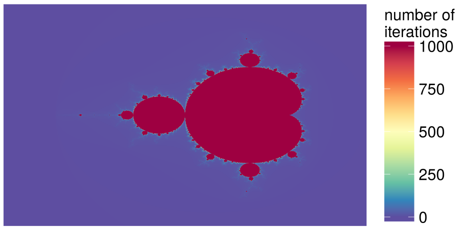

Figure 5: Number of iterations needed to calculate the Mandelbrot set

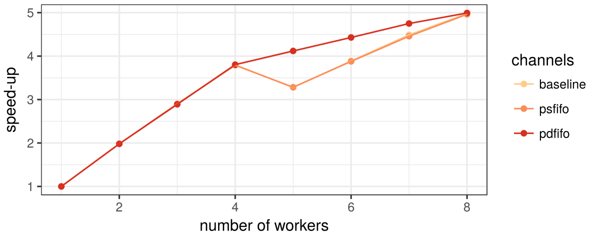

Figure 6: Speedup on Intel i7-4790

We tested our benchmark on three versions of data-parallelism in KPNs: `baseline`, `psfifo` and `pdfifo`. In the `baseline` version, data parallelism is inserted manually on the KPN at the source-code level by implementing the split-interleave functionality as part of the processes. The capacity of each channel in `baseline` version is 8 tokens, and the capacity of parallel channels is 8 × number of workers. In our experiments we calculate the Mandelbrot set for two areas: 4000 × 3000 and 8000 × 6000, for the experiments in Sections 3.2 and 3.3 respectively. In all scenarios, all parallel workers were fixed to individual cores/hardware threads, without sharing, while the source and sink processes were not fixed.

3.2 Evaluation of channels

In the first experiment we check the performance of all three versions by varying the number of parallel workers on both, Intel i7-4790 and Odroid XU-4.

To measure the performance on Intel i7-4790 we executed all three versions, with the number of parallel workers ranging from 1 to 8. In configurations of up to 4 parallel workers, the workers were fixed to different physical cores, and on the other configurations, additional workers were fixed to the second hardware thread of each core. The results are shown on Figure 6. It shows how the speed-up curves of `baseline` and `psfifo` are virtually identical, as well as all three curves are for up to 4 parallel workers. However, for more than 4 workers `pdfifo` outperforms the other versions by up to 25.6 %. This effect diminishes up to 8 cores, where the versions perform again virtually identically. The reason of the worse performance of the two static versions in the middle of the plot is that in these configurations some workers use a physical core exclusively, while the other workers share a core on different hardware threads. Thereby, they become stragglers, slowing down the rest of the process network. In the dynamic version, the faster workers take over part of the workload from the slower ones, improving performance.

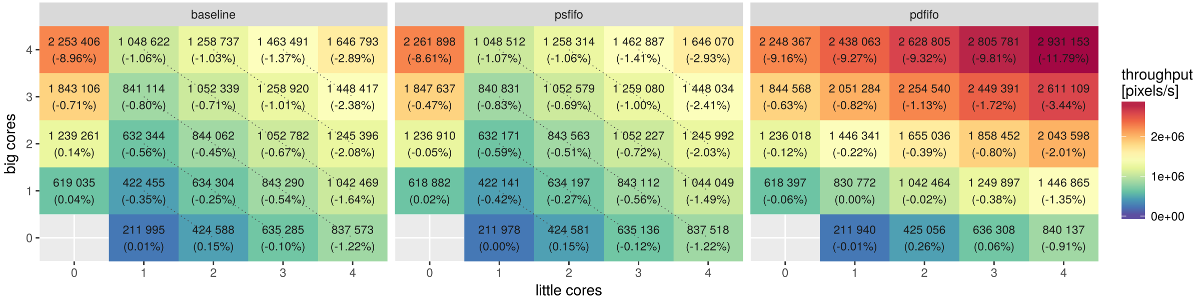

Figure 7: Throughput on Odroid XU-4 with different numbers of little and big cores

On Odroid XU-4 we executed all versions for all possible combinations of numbers of *little* and *big* workers, which we define as workers running on little and big cores respectively. The performance is reported in terms of throughput, in pixels/s. The results are depicted in Figure 7. Similar to the previous experiment, both `baseline` and `psfifo` show equal throughput on each configuration. All three versions show the same results when all workers allocated to homogeneous subsets of cores, either big or little. When the workers are fixed to a heterogeneous subset of cores, `pdfifo` shows a better performance, by up to 78.0 %. Perhaps surprisingly, the static versions with 4 big workers outperform the ones with 4 big and 4 little workers by 36.8 %.

This can again be explained by straggling workers slowing down the rest of the system. We can quantify this observation, by approximating the total throughput for the static and dynamic kernels as follows:

$$TH^{\text{stat}}_{\text{total}} \approx \begin{cases} TH_{\text{little}} \times (b + l) & \text{if } l > 0 \\ TH_{\text{big}} \times b & \text{if } l = 0 \end{cases}$$

$$TH^{\text{dyn}}_{\text{total}} \approx TH_{\text{little}} \times l + TH_{\text{big}} \times b,$$

where l and b are the number of little and big workers, respectively.

On Figure 7 the relative error of this approximation is shown in brackets. The dashed lines show equally-performing configurations of the static versions. On all configurations in which the big cluster is not fully utilized, the relative error of our approximation does not exceed 3.44 %. When the big cluster is fully utilized we see a significant increase of the relative error (11.79 %) because the big cluster is not able to sustain high frequency. This, in turn, is due to the inability of the stock fan to adequately cool the chip.

Both experiments show that, indeed, in the static versions the rigid execution model allows straggling workers to impair the execution in the whole system.

3.3 Adaptivity

In the previous experiments we only evaluated our approach on systems with fixed resources. In this section, we analyze the runtime adaptability of the different versions. We do this at two levels: workload distribution adaptivity, and parallelization adaptivity.

3.3.1 Workload distribution adaptivity. In this scenario we run the benchmark with four little workers on Odroid-XU4. At runtime, we start a second application (a sorting task), which creates contention by sharing one of the little cores with our benchmark. We start two instances of this contending application, one after 15 s and the other one 17 s later. Figure 8 shows the throughput of the benchmark as a function of time, for all three versions. For reference, the same graphs for all three versions, without an additional application contending for resources, are shown as the dotted lines in the plot. We see how the contending application dramatically reduces the throughput of the architecture on `baseline` and `psfifo`, whereas the reduction is much less dramatic for `pdfifo`. This is because the worker sharing a core with the other application becomes a straggler and slows down all the other workers in the static versions. This effect is avoided by the dynamic `pdfifo`. Note that the high throughput at the beginning and the end of the execution relates to the data-dependent behavior of the application, as at those points it is calculating the edges of the square area, far from the Mandelbrot set (see Figure 5).

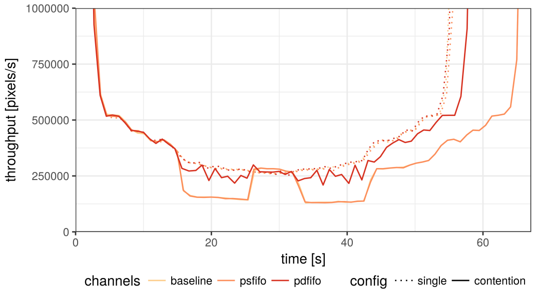

Figure 8: The Effect of Contention on Performance for the Different Variants.

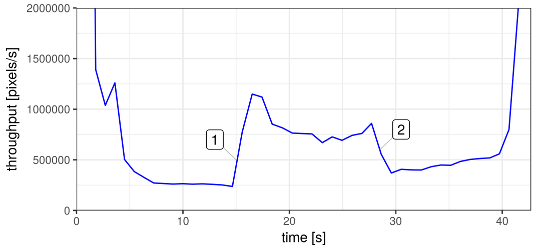

Figure 9: Dynamic Reallocation of Resources.

3.3.2 Parallelization adaptivity. In this scenario we change the amount of parallelism at run-time. This feature is only supported by the `pdfifo` library, since its dynamic nature allows to add and remove instances of the parallel process without affecting the currently-executing ones. Figure 9 shows the evolution of throughput over time throughout the experiment. It starts with two little workers. After 15 s, four big workers are added, marked by (1). Then, after an additional delay of 13 s, three big and one little worker are removed. This event is marked by (2). The plot shows how the pdfifo library can adapt to the resources available at run-time, utilizing them to the fullest.

4 RELATED WORK

The optimization of data-parallelism in dataflow applications has attracted much interest from the research community. In the literature it is called with different names, such as fission, partitioning or replication [8].

In StreamIt [5, 6], the authors apply fission tranformations along with other tranformation like fusion, reordering and load balancing at compile-time. Similarly, several authors use diverse approaches to leverage data-parallelism in SDF graphs at compile time [3, 11, 23–25]. Conversely, the authors of [16] introduced a dynamic scheduling approach for SDF graphs. However, our approach works with the far more expressive KPN model of computation.

More closely related to our work with KPN, in AdaPNet [21], a new application model called Expandable Process Networks (EPN) is introduced as an extension of KPN. In this model, a stateful process might be expanded to a refinement network and contracted back, resulting in a new process network that has the same functionality as the original network. Though this system addresses more general transformations on stateful processes, these transformations are not malleable nor does it address implicit data-parallelism. The transformations have to be specified explicitly. The authors of [17], on the other hand, offload the computation of stateless actors to the GPU units. In contrast to this, our approach works in more general heterogeneous systems through the abstract nature of the SLX flow. Moreover, this work uses a non-deterministic model of computation through RVC-CAL, whereas our approach still guarantees a deterministic execution.

To the best of our knowledge, our approach is the first solution that works on the highest possible level that guarantees determinism (KPN), while improving run-time adaptivity in malleable way and providing implicit parallelism, effectively briding the MacQueen gap.

5 CONCLUSION

In this paper we presented an extension of Kahn Process Networks with an alternative execution model. We showed how, by replicating certain stateless processes, we can leverage data-parallelism in KPNs in an implicit fashion. Additionally, we argued how we can remove the blocking reads and writes from the KahnMacQueen execution model while retaining the deterministic KPN semantics. We showed how this holds true, improving the performance of a benchmark on two architectures, by up to 25.6 % on commodity desktop hardware with hyperthreading, and by up to 78.0 % on the heterogeneous Odroid system. In particular, the dynamic nature of our execution model allows the execution to utilize all system resources, avoiding being blocked by a straggling process and rigid semantics.

In future work we will address questions of optimal energy-efficient mapping for such process networks and study an efficient implementation of our extension to platforms with a distributed memory model. Additionally, we plan to explore relaxing the semantics of a subgraph, violating the KPN semantics in a controlled fashion, to further improve performance when it is safe to do so.

ACKNOWLEDGMENTS

This work was supported in part by the German Research Foundation (DFG) within the Collaborative Research Center HAEC and the Center for Advancing Electronics Dresden (cfaed). We thank Silexica for making their SLX Tool Suite available to us.

REFERENCES

[1] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. 1996. Cycle-static dataflow. *IEEE Transactions on signal processing* 44, 2 (1996), 397–408.

[2] Jeronimo Castrillon Mazo and Rainer Leupers. 2014. *Programming heterogeneous MPSoCs: tool flows to close the software productivity gap.* Springer, Cham.

[3] Sardar M. Farhad, Yousun Ko, Bernd Burgstaller, and Bernhard Scholz. 2011. Orchestration by Approximation: Mapping Stream Programs Onto Multicore Architectures. *SIGPLAN Not.* 47, 4 (March 2011), 357–368.

[4] Dror G. Feitelson and Larry Rudolph. 1996. Towards Convergence in Job Schedulers for Parallel Supercomputers. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS '96).* Springer-Verlag, London, UK, UK, 1–26. http://dl.acm.org/citation.cfm?id=646377.689507

[5] Michael I. Gordon and et al. 2002. A Stream Compiler for Communication-exposed Architectures. *SIGARCH Comput. Archit. News* 30, 5 (Oct. 2002), 291–303. https://doi.org/10.1145/635506.605428

[6] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. *SIGPLAN Not.* 41, 11 (Oct. 2006), 151–162. https://doi.org/10.1145/1168918.1168877

[7] Peter Greenhalgh. 2011. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper* (2011), 1–8.

[8] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4, Article 46 (March 2014), 34 pages. https://doi.org/10.1145/2528412

[9] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. In *Information processing*, J. L. Rosenfeld (Ed.). North Holland, Amsterdam, Stockholm, Sweden, 471–475.

[10] Gilles Kahn and David MacQueen. 1976. Coroutines and networks of parallel processes. (1976).

[11] Manjunath Kudlur and Scott Mahlke. 2008. Orchestrating the Execution of Stream Programs on Multicore Platforms. *SIGPLAN Not.* 43, 6 (June 2008), 114–124. https://doi.org/10.1145/1379022.1375596

[12] Edward A Lee. 2015. The past, present and future of cyber-physical systems: A focus on models. *Sensors* 15, 3 (2015), 4837–4869.

[13] Edward A Lee and Eleftherios Matsikoudis. 2008. The semantics of dataflow with firing. *G. Huet, G. Plotkin, J.-J. Lévy, and Y. Bertot, editors, From Semantics to Computer Science: Essays in Honour of Gilles Kahn* (2008), 71–94.

[14] E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (Sept 1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

[15] Edward A Lee and Thomas M Parks. 1995. Dataflow process networks. *Proc. IEEE* 83, 5 (1995), 773–801.

[16] Haeseung Lee, Weijia Che, and Karam Chatha. 2012. Dynamic Scheduling of Stream Programs on Embedded Multi-core Processors *(CODES+ISSS '12).* ACM, New York, NY, USA, 93–102. https://doi.org/10.1145/2380445.2380465

[17] W. Lund, S. Kanur, J. Ersfolk, L. Tsiopoulos, J. Lilius, J. Haldin, and U. Falk. 2015. Execution of Dataflow Process Networks on OpenCL Platforms. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing.* 618–625. https://doi.org/10.1109/PDP.2015.29

[18] Andreas Olofsson. 2016. Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip. *arXiv preprint arXiv:1610.01832* (2016).

[19] Andy D Pimentel, Cagkan Erbas, and Simon Polstra. 2006. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Comput.* 55, 2 (2006), 99–112.

[20] Claudius Ptolemaeus. 2014. *System design, modeling, and simulation: using Ptolemy II.* Vol. 1. Ptolemy. org Berkeley.

[21] L. Schor, I. Bacivarov, H. Yang, and L. Thiele. 2014. AdaPNet: Adapting process networks in response to resource variations. In *CASES 2014.* 1–10. https://doi.org/10.1145/2656106.2656112

[22] Silexica. 2017. SLX. (2017). http://www.silexica.com

[23] J. Spasic, D. Liu, and T. Stefanov. 2016. Exploiting resource-constrained parallelism in hard real-time streaming applications. In *DATE 2016.* 954–959.

[24] A. Stulova, R. Leupers, and G. Ascheid. 2012. Throughput driven transformations of Synchronous Data Flows for mapping to heterogeneous MPSoCs. In *2012 International Conference on Embedded Computer Systems (SAMOS).* 144–151. https://doi.org/10.1109/SAMOS.2012.6404168

[25] J. T. Zhai, M. A. Bamakhrama, and T. Stefanov. 2013. Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC).* 1–8. https://doi.org/10.1145/2463209.2488944