HAL
open science

DKPN: A Composite Dataflow/Kahn Process Networks Execution Model

Paul-Antoine Arras, Didier Fuin, Emmanuel Jeannot, Samuel Thibault

To cite this version:

Paul-Antoine Arras, Didier Fuin, Emmanuel Jeannot, Samuel Thibault. DKPN: A Composite Dataflow/Kahn Process Networks Execution Model. 24th Euromicro International Conference on Parallel, Distributed and Network-based processing, Feb 2016, Heraklion Crete, Greece. 10.1109/PDP.2016.34 . hal-01234333

HAL Id: hal-01234333

https://inria.hal.science/hal-01234333

Submitted on 12 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DKPN: A Composite Dataflow/Kahn Process Networks Execution Model

Paul-Antoine Arras*†‡, Didier Fuin†, Emmanuel Jeannot‡, Samuel Thibault§‡
* CEA, LIST, Gif-sur-Yvette, France, paulantoine.arras@cea.fr
‡ Inria Bordeaux Sud-Ouest, LaBRI, France, emmanuel.jeannot@inria.fr
† STMicroelectronics, Grenoble, France, didier.fuin@st.com
§ University of Bordeaux, LaBRI, France, samuel.thibault@u-bordeaux.fr

***Abstract*—To address the high level of dynamism and variability in modern streaming applications (e.g. video decoding) as well as the difficulties in programming heterogeneous MPSoCs, we propose a novel execution model based upon both dataflow and Kahn process networks. This paper presents the semantics and properties of this hierarchical and parametric model, called DKPN. Parameters are classified and it is shown that hints can be derived to improve the execution. A scheduler framework and policies to back the model are also exposed. Experiments illustrate the benefits of our approach.**

I. Introduction

Multiprocessor System-on-Chip (MPSoC) development is known to be a complex task due to the combination of software and hardware challenges. On the software side, target applications are increasingly complex and demanding in computation power. On the hardware side, modern architectures, however powerful they may be, often cannot be fully exploited due to a lack of well-suited programming models and tools. For instance, in the field of video decoding, the major algorithms, namely the ubiquitous H.264/AVC [1] and its successor HEVC [2], exhibit a very high level of complexity and dynamism (i.e. variability and unpredictability) which requires both *flexibility* and *computing power* on the part of the platform. The former can only be achieved by software, while the latter is clearly more amenable to hardware, hence the need for heterogeneous architectures. But such mixed hardware/software implementations are notoriously difficult to program efficiently [3]. To ease the development process, a number of paradigms have already been proposed, including dataflow and Kahn process networks which lend themselves well to streaming applications. The latter requires very little knowledge about the application but can be costly to execute, while the former needs more information in exchange for efficient execution. Therefore, in the context of mapping dynamic streaming applications onto heterogeneous MPSoCs, an important question remains: how to get the best of both worlds?

To tackle this problem, this paper presents DKPN, a parameterized dataflow/Kahn process networks execution model that aims at: (1) bridging the gap between high-level models and real implementations; (2) supporting heterogeneous platforms, including hybrid Von Neumann/dataflow architectures [4]; (3) capturing the highest level of application dynamism. In detail, the contributions of the paper are: (1) a composition semantics for dataflow and Kahn process networks; (2) parameters to address variability and unpredictability at run time; (3) a scalable, dynamic, distributed scheduler to back the execution model; (4) effective scheduling strategies.

The remainder of this paper is organized as follows. Section II briefly describes the existing models upon which DKPN is built. Section III details the proposed execution model and Section IV exposes how it can be scheduled efficiently. Section V presents some experimental results. Section VI discusses some related work. Section VII sums up our contributions and concludes.

II. Background

A. Dataflow

Historically, the very first research in dataflow programming dates back to Dennis' seminal work [5]. Numerous variants—called *models of computation* (MoCs)—have been proposed since then, either static or dynamic. All MoCs share a common set of definitions and properties. Every dataflow program is defined as a directed graph whose vertices are computational units called *actors* and edges are conceptually unbounded communication channels carrying atomic data units called *tokens* in a first-in-first-out (FIFO) fashion. The interface between an actor and a FIFO is called a *port*. The execution of an actor is a discrete process composed of a succession of *firings*. A firing is an atomic quantum of computation that consumes some number of input tokens to produce a certain amount of output tokens. Those quantities are called *rates* and, in the general case, can change at every firing. A necessary condition for an actor to fire is that a predefined set of *firing rules* should be satisfied. These rules concern the number and possibly the values of input tokens [6]. When an actor is not firing, i.e. not running, it is said to be *quiescent*. In dataflow models, reads (i.e. the attempts for an actor to consume input tokens) are non-blocking—meaning that they always return immediately and successfully—thanks to the very existence of firing rules. Due to the unboundedness of FIFOs, writes are also non-blocking in theory[1]. A practical implication is that, from a scheduling viewpoint, an actor firing can be seen as a run-to-completion task. In this paper, we focus on the Dataflow Process Networks (DPN) [6], which is the most expressive dataflow MoC: rates and firing rules are not constrained in any way.

[1] As we shall see later, FIFOs in real implementations are bounded due to limited memory capacity.

TABLE I. DIFFERENCES BETWEEN DPN AND KPN.

	DPN	KPN
Execution	Discrete	Continuous
Firing rules	Yes	No
Blocking reads	No	Yes

B. Kahn Process Networks

KPN [7] is a MoC very closely related to dataflow, and especially to DPN. The main difference is that in KPN the execution is continuous rather than discrete, in the sense that there is no well-defined quantum of computation, i.e. no notion of firing. Thus, input tokens are processed as soon as they become available and output tokens are pushed as soon as they have been produced. Due to the lack of firing rules and so as to guarantee determinism, reads are blocking: an attempt to consume tokens from an empty channel will result in suspending the reading process. Hence, from an execution viewpoint, the need for context switches between processes to avoid deadlocks as long as there are more of them than execution resources.

Besides, KPN and DPN graphs can be classified according to two properties [8]:

- termination: a complete execution consists of a finite number of operations;
- boundedness: the execution requires a finite amount of memory.

Table I sums up the main differences between these two MoCs.

C. PEDF

Predicated Execution Dataflow [9, sec. 2.1.5.1] is the programming model developed and used by STMicroelectronics for STHORM (formerly P2012) [10], an embedded hybrid platform for MPSoCs comprising programmable cores and hardware accelerators. A PEDF application is a hierarchical graph whose computational units are called *filters*. Several filters can be clustered into a *module* with a controller aimed at reconfiguring them and driving their execution. This execution is decomposed into firings but without firing rules. The idea of separating control and processing into distinct entities lays the ground for the proposed model.

Still, PEDF suffers from a number of shortcomings. First, it is not supported by a well-defined execution model. Second, it does not feature a scheduler. Third, software filters are not supported, which greatly hampers heterogeneous flexibility. Thus, given a heterogeneous platform and a dynamic application, we propose the DKPN execution model to solve the mapping problem and fill the gap left by existing approaches.

III. THE DKPN EXECUTION MODEL

The *execution model* defines a set of actions and rules involved in executing the application. It can be seen as the intermediate layer between the compiler, aimed at generating the application's code, and the runtime software supporting the target platform, as depicted by Fig. 1. Moreover, it provides guidelines for the scheduler's action.

DKPN is an execution model in which a program is described as a graph with the structural properties stated in

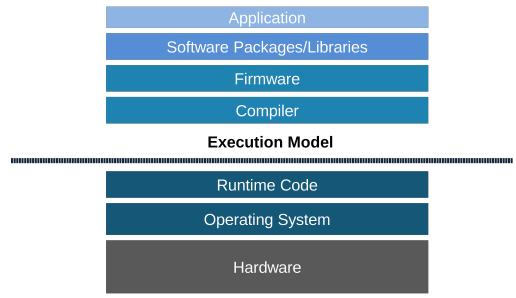


Fig. 1. Place of the execution model within the software stack. Source: STMicroelectronics.

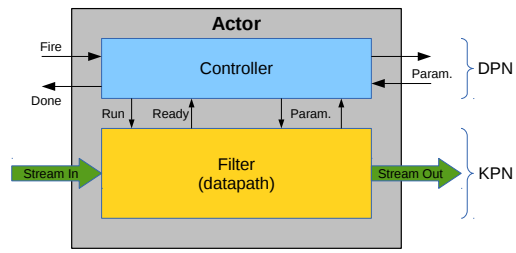


Fig. 2. Decomposition of a DKPN actor into its controller and its filter. The controller exchanges commands (`fire`/`done`) with the scheduler and parameters with other actors. It drives the filter through `run` and `ready` commands and reconfigures it through internal parameters. The filter exchanges data with other actors via streaming interfaces and processes them. Controller and filter respectively belong to DPN and KPN spaces.

Section II-A, similarly to dataflow MoCs. The building block is still the actor, here defined as a composite entity consisting of a *controller* and a *filter*, as shown in Fig. 2. As in PEDF, the filter processes the data while the controller is responsible for its reconfiguration.

Formally, a DKPN graph $\mathcal{G}$ is a pair $(\mathcal{A}, \mathcal{E})$ where $\mathcal{A}$ is the set of actors and $\mathcal{E}$ the set of FIFOs. A DKPN actor $A \in \mathcal{A}$ is a pair $(A_C, A_F) \in \mathcal{C} \times \mathcal{F}$ where $\mathcal{C}$ and $\mathcal{F}$ are respectively the sets of controllers and filters. A FIFO may carry either data or parameter tokens, but not both: $\mathcal{E} = \mathcal{E}_D \uplus \mathcal{E}_P$ where $\mathcal{E}_D$ and $\mathcal{E}_P$ respectively denote the sets of data and parameter FIFOs, and $\uplus$ denotes disjoint union. A DKPN graph is split into two *spaces* with well-defined semantics: the DPN space, denoted $\mathcal{D}$, encompasses controllers and parameter FIFOs, while the KPN space, denoted $\mathcal{K}$, comprises filters and data FIFOs. Thus:

$$\mathcal{D} = (\mathcal{C}, \mathcal{E}_P), \qquad \mathcal{K} = (\mathcal{F}, \mathcal{E}_D) \ .$$

In dataflow, an important notion is that of an *iteration*, defined very generally as a minimal sequence of firings after which a graph or a subgraph is back into its initial state. In dynamic models such as DKPN, it is not possible to derive analytically an iteration from the graph's specification—which is why dynamic scheduling is required in the general case (see Section IV)—since it can be data dependent[2]. Therefore, the precise definition of an iteration largely depends on the application. For instance, in an H.264 video decoder, an iteration may correspond to the decoding of one video frame. In DKPN, each actor has its own concept of iteration. When an actor has ended its iteration, it is put into a special state. When all actors

[2]Thus, the application's designer has to ensure that his or her design will not end up inevitably in an infinite accumulation of tokens.

are in end-of-iteration state, the graph is also done with its iteration. This is a quiescent point for the whole graph that may serve for synchronization or structural reconfiguration (see Section III-B). DKPN supports non-terminating applications and thus allows infinite iterations.

A. Composition semantics

This paragraph describes how the DPN and KPN spaces of a DKPN graph interact. The composition rules at actor level are the following. During quiescent states, the filter has no conceptual existence: the actor is solely constituted of its controller. The firing of the actor is thus equivalent to that of its controller, and consequently abides by DPN semantics: it has to follow its firing rule. In other words, a DKPN actor firing shall happen only if the parameters required are present in input FIFOs. This firing is then internally split into three subfirings, but from an external viewpoint the firing is still atomic. The first subfiring stands for the reconfiguration of the actor: it enables the controller to gather input parameters and spawn a filter instance configured accordingly through internal parameters defined by the application. The second subfiring corresponds to the execution of the filter under KPN semantics: data is read, processed and written continuously until completion of the filter's work. During the second subfiring, the controller conceptually disappears and the actor as a whole behaves as a KPN process. The last subfiring happens when the filter is done: it then yields to the controller which gathers internal parameters, renders some post-processing and sends resulting output parameters. After the end of firing, which corresponds to the end of the last subfiring, the actor goes back to its initial quiescent state. Thus, when quiescent, a DKPN graph can be treated as a dataflow graph.

B. Parameters

We propose to use parameters to drive reconfigurations both at actor and at graph levels. This is in contrast to Neuendorffer's approach [11], which distinguishes parametric (actor-level) and structural (graph-level) reconfigurations, and uses parameters only for the former. We define parameters as a class of tokens distinct from data tokens. Parameter tokens have their own ports, FIFOs and semantics. Conceptually, data tokens undergo some processing from filters duly reconfigured under the control of parameter tokens. A parameter has one the following types depending on its effects:

- functional: modify the internal behavior of the target actor;
- data-rate: change the rates of the data ports of the target actor;
- parameter-rate (or meta-): change the rates of the parameter ports of the target actor;
- structural: alters the graph structure by enabling or disabling some set of actors and FIFOs.

Functional, data-rate and meta- parameters trigger actor-level reconfigurations, whereas structural parameters drive graph-level reconfigurations. Actor-level reconfigurations can happen at each actor firing, while graph-level ones are restricted to hierarchical quiescent points. Data-rate parameters do not have

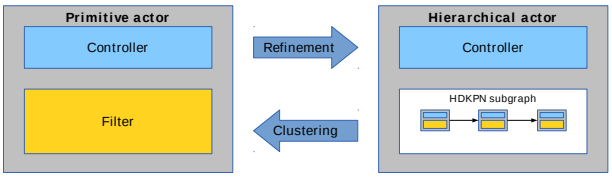


Fig. 3. Refinement and clustering between a primitive actor and its corresponding hierarchical actor.

to be taken into account since they belong to the KPN space of the DKPN graph, but they might be of interest for the scheduler as shown in Section IV-A.

C. Hierarchical composition

The baseline DKPN execution model can be extended to support hierarchy, it is then called HDKPN. As formalized by Neuendorffer [11], a hierarchical dataflow graph can be represented as a tree of actors. In an HDKPN graph, a leaf node is called a *primitive* actor and a non-leaf node a *hierarchical* actor[3]. A primitive HDKPN actor is a DKPN actor. A hierarchical HDKPN actor consists of one controller and any number and combination of actors, which can themselves be hierarchical, similarly to a PEDF module. Thus, a primitive actor can easily be *refined* into a hierarchical one by decomposing its filter into an HDKPN subgraph. Conversely, a hierarchical actor can be *clustered* into a primitive one by merging its subgraph into a single filter. To ensure compositionality, both refinement and clustering transformations shall preserve the actor's external interface and controller.

From an external viewpoint, the firing of a hierarchical actor has exactly the same semantics as a primitive actor. Furthermore, a firing is still internally split into three subfirings. The main difference lies in the second subfiring: the execution of the filter is replaced by the execution of the subgraph, called subiteration. For instance, in an H.264 video decoder, a subiteration may correspond to the decoding of one macroblock[4]. Subiterations are defined analogously to DKPN iterations: all actors in the subgraph have to be in end-of-iteration state for the subiteration to terminate. To guarantee compositionality, transformations shall not impact the application's termination property. In particular, refining an actor in a terminating graph shall not make it non-terminating by introducing an infinite subiteration.

As regards hierarchical reconfigurations, they shall happen only during hierarchical quiescent points [11]: all actors (primitive or hierarchical) involved in the reconfiguration must be quiescent simultaneously. As a matter of fact, hierarchical reactivity demands that a nested actor should be quiescent whenever its encompassing parent is. For the special case of structural reconfigurations, the nearest ancestor of all involved actors in the tree must be quiescent.

Tokens are passed across the hierarchy in the following way. When an actor is refined, the data FIFOs previously connected to the filter are extended to enter the resulting

[3]The terms *atomic* and *composite* are not used in this context since leaf nodes are already composite due to their decomposition into controller and filter entities.

[4]A macroblock is a block of pixels, typically of size 16×16.

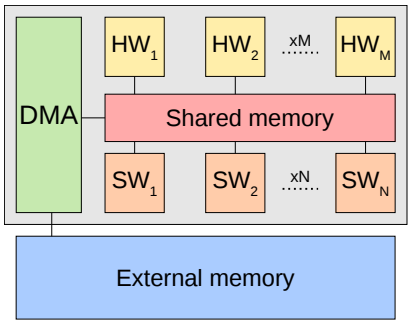


Fig. 4. Model of the target computing environment.

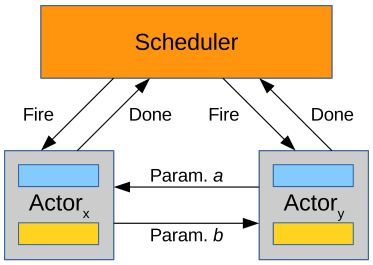


Fig. 5. Relations between scheduler and actors. Parameters a and b are exchanged directly between actors x and y. The scheduler drives the actors through `fire` and `done` signals.

subgraph and reach the relevant actor. As regards parameters, new FIFOs must be added to route the internal parameters issued by the subgraph controller to their respective recipient actors[5] during the first subfiring. Symmetrically, additional FIFOs are required for the subgraph controller to gather internal parameters resulting from the subgraph's computation during the last subfiring.

IV. Scheduling of DKPN

In this section, the target computing environment is supposed to be a heterogeneous platform featuring both general-purpose software processing elements (SWPEs) and specialized hardware accelerators (HWPEs) with shared memory, as depicted by Fig. 4. Each HWPE supports only a single actor but runs it faster than a SWPE. Given such a heterogeneous platform and an application described as a DKPN graph, the scheduling problem consists in deciding on the assignment and ordering of actor firings onto the available PEs. The scheduler's duties are: arbitrating between actors sharing the same resources, enforcing the semantics of the execution model and preventing deadlocks. The proposed scheduler structure has the following properties:

- symmetrically distributed on all SWPEs;
- cooperative: a task[6] has to yield explicitly to the scheduler, otherwise it cannot be preempted;
- policy agnostic: any user-defined scheduling strategy can be used, Section IV-B provides examples of some them.

As shown in Fig. 5, parameters are passed directly across actors. Nevertheless, as the scheduler is in charge of checking the firing rules, it has to constantly monitor the availability of parameters thanks to counters. When an actor has its firing rule asserted, i.e. required input parameters are available, it is flagged as ready, then the scheduler emits the `fire` signal. At the end of its firing, an actor sends the `done` signal to the scheduler—meaning that output parameters have been produced and that it is waiting for the next command—and is put into idle state.

A scheduling step is a sequence of operations performed by the scheduler and comprising three phases:

1) find ready actors and select the next to be scheduled;
2) map and order them using the selected scheduling policy;
3) fire an actor.

Algorithm 1 describes in more detail these operations.

During phase 2, tasks are pushed to one of the global[7] task queues. The number and the nature of such queues depends on the strategy. Increasing this number may be necessary for scalability reasons. It may also improve the scheduler's performance by allowing some specialization, e.g. one queue for high-priority tasks and one for low-priority ones, or one for SWPE-mapped tasks and one for HWPE-mapped ones. In phase 3, the scheduler has to select a task queue according to the strategy and the available PEs.

A. Hint parameters

The parameter types presented in Section III-B do not reflect the full expressiveness of parameters. In fact, it is possible to devise a new class of parameters aimed at providing hints to the scheduler so as to improve its efficiency; therefore we refer to them as *hint parameters*. This class is orthogonal to the previous classification in the sense that a hint parameter can be of any of the defined types. Thus, a parameter is a hint parameter if its value can be taken as input and interpreted by the scheduler for optimization purposes. The rationale is that application designers often have useful additional knowledge about individual firings that cannot be expressed through classical models: computation or memory boundedness, amount of data transferred, complexity, etc. This information can be encoded using *ad hoc* or existing parameters (functional, dataflow, etc.) and leveraged at run time, for instance to predict the execution time of the next firing, with a view to improving scheduling decisions. The exact nature of the information typically depends on each application's specificities, but a simple scenario would be to map a parameter value modeling computation complexity to a proportional duration indicator.

B. Scheduling policies

To implement the second phase of the presented scheduler, we propose five scheduling strategies. To ensure fairness, all policies schedule every ready actor at each step; only the order differs. Also, they require only a single task queue.

[5]Note that internal parameters emitted by the subgraph controller are still viewed as regular input parameters by the receiving actors.

[6]In the sequel, the term *task* refers to an actor firing whose mapping and ordering have already been decided.

[7]Global in the sense that it is shared by all SWPEs.

```
begin scheduling step
    /* Phase 1                                  */
    foreach pending done signal do
        A: emitting actor ;
        Acknowledge signal from A ;
        Update available parameters ;
        Put A into idle state ;
        if no input parameter required or all available
        for A then
            Flag A as ready
        end
        foreach produced parameter do
            F: associated FIFO ;
            B: target actor ;
            if exactly 1 parameter token in F then
                Flag parameter as available ;
                if all parameters are available and B is
                idle then
                    Flag B as ready
                end
            end
        end
    end
    /* Phase 2                                  */
    scheduling_policy() ;
    /* Phase 3                                  */
    Pop from task queue ;
    Run task ;
end
```

Algorithm 1: Description of a scheduling step, applicable to all policies detailed in Section IV-B.

a) Round-robin (RR).: Ready actors are sorted in topological order. At each step, the list is traversed to find and schedule ready actors in this order. The last scheduled actor in step n is memorized and the scan is resumed in step $n+1$ with the next actor in the list.

b) Short first (SF).: This scheduling strategy uses hint parameters to predict the duration of each firing. At each step, actors are sorted in increasing duration order. The list is traversed once, starting by the shortest actor. Ties are broken using topological order.

c) Long first (LF).: This scheduling policy is similar to the previous one, except that actors are sorted in decreasing duration order. Then longer actors are scheduled beforehand.

d) Dynamic priorities (DP).: At each step, actors are assigned a priority that reflects the filling of their data FIFOs. Parameter FIFOs are not considered when setting priorities because they pertain to the DPN space and thus token transfers across them are already regulated by firing rules. The idea is to prioritize actors with fuller input and emptier output FIFOs so as to prevent token accumulation and thus avoid context switches. Let $c(e)$ denote the capacity of FIFO $e \in \mathcal{E}_D$ and $n(e,s)$ the number of tokens in e at step s. Then, the priority P of actor $A \in \mathcal{A}$ at step s is given by:

$$P(A,s) = \sum_{e^i \in \mathcal{E}_D^i(A)} \frac{n(e^i,s)}{c(e^i)} - \sum_{e^o \in \mathcal{E}_D^o(A)} \frac{n(e^o,s)}{c(e^o)} \ ,$$

where $\mathcal{E}_D^i(A)$ and $\mathcal{E}_D^o(A)$ are the sets of input and output data FIFOs of actor A. The sums are normalized over the capacities since they may differ even for a single actor and we are only interested in FIFO fullness. Moreover, normalizing over the number of FIFOs rather than capacities would not be helpful since a higher number of FIFOs generally means more sources of deadlock so actors with more asymmetrical inputs/outputs should have higher priorities in absolute value. Ties are broken using topological order.

e) Dynamic priorities with hint parameters (DP+H).: This policy is the same as the previous one, except that ties are broken using hint parameters. In this case, shorter actors are scheduled first.

C. Discussion on deadlocks

In process networks, deadlocks can result either from an inconsistent application design or from the execution model. On the one hand, the DPN part of the model is guaranteed to be free from deadlocks thanks to firing rules. On the other hand, the KPN part can yield deadlocks, except if data token rates are provided through hint parameters. The rationale for not including deadlock-detection and -resumption mechanisms is threefold. First, in most cases, resolving a deadlock implies a context switch between processes, which incurs a non-negligible run-time cost. Second, it can be avoided in a number of cases like in all our experiments. Third, since the issue of deadlocks in KPN has already been extensively studied [12], [13], [14], existing techniques can easily be adapted to our model. Thus, in this article, we chose to focus on deadlock prevention through adequate scheduling rather than resumption.

V. Experiments

A. Overhead evaluation

As the presented execution model is geared toward dynamic applications, we assess its overhead by applying it to a static one. The evaluation consists in running a real-world Temporal Noise Reduction (TNR) application from STMicroelectronics on the TLM simulator[8] of the STHORM platform [10]. Controllers are supposed to be mapped to SWPEs and filters to HWPEs. We have reworked the original PEDF implementation of TNR to comply with DKPN. Also, the whole PEDF/STHORM toolchain has been modified to adapt to DKPN. We compare a *reference* implementation with a *scheduled* one. The reference PEDF is a monolithic version very similar to the original, with a hand-written optimized schedule. In the scheduled PEDF, DKPN is used in conjunction with a simple dynamic scheduler that fires actors repeatedly in topological order. As the simulator only provides a functional model for hardware filters, their execution times cannot be derived directly. Instead, the variability is simulated by random durations of actor firings generated as follows.

Each actor is assigned a—supposedly mean—reference computation time rct that serves as a basis for the computation of a delay added to the simulated time. This delay includes a

[8]The platform is modeled as loosely timed and SWPEs as instruction-accurate.

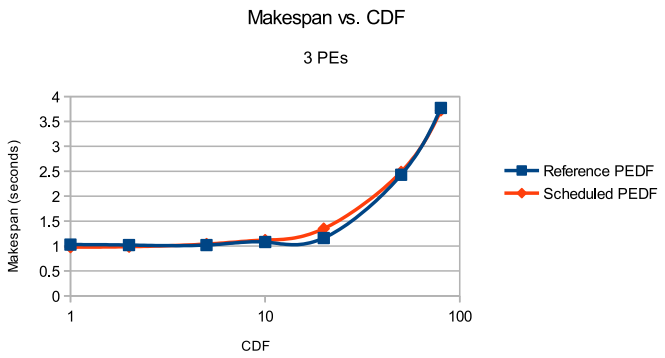


Fig. 6. Comparison of execution times of the TNR application in reference and scheduled PEDF with different variability levels for overhead evaluation.

random factor rdf sampled from a Beta law and a constant factor cdf; both factors are common to all actors. cdf models the average speedup introduced by the hardware implementation over the reference software, while rdf accounts for variability and unpredictability. The resulting total delay td is computed as: $td = cdf \times rdf \times rct$. td is then reevaluated at every firing and added to the SystemC time during the simulation. In these experiments, we measured the makespans of several TNR runs with both reference and scheduled PEDF versions against different cdf values. The simulated platform comprises three SWPEs for the scheduler and controllers and one HWPE for each filter. Figure 6 shows the results.

For lower cdf values, the performances of both versions are equivalent, with a slight advantage of about 4% for the scheduler. When the delay factor increases, the gap widens in favor of the reference up to 16% for $cdf = 20$. For $cdf \geq 50$, the difference becomes insignificant: only about 1%. Therefore, three distinct behaviors can be observed for different variability levels. For a small delay factor ($cdf \in [1, 5]$), computation times are of the same order of magnitude as the time spent by the SWPEs to control and schedule actors. For a medium delay factor, computation times begin to dominate, hence a sharp increase in makespans: the reference is better than the scheduler since the raw cost of its decisions prevails over their benefits in terms of schedule. Lastly, for longer hardware-filter delays, all the control is negligible compared to the computation: both versions have comparable performances. Ultimately, in the worst case, DKPN entails a 16% overhead in execution time compared to an *ad hoc*, optimized scheme.

Moreover, the memory cost of the execution model and the scheduler has also been evaluated. The runtime code size increased from 4.7 KB to 7.6 KB, which remains reasonable compared to the application's (23 KB) and the capacity of program caches in STHORM (32 KB). The data size is still less than 100 B.

B. Design space exploration

1) Experimental conditions: To assess the benefits of the proposed execution model and scheduler under different circumstances, we built a custom simulator. The execution model can thus be simulated on any number of PEs, and various other parameters can be adjusted: actor mapping, FIFO sizes, scheduling policy, etc. This allows to circumvent the limitations of PEDF/STHORM's simulation environment, especially its inability to tweak FIFOs and architectural parameters as needed, as well as its low performance and limited debugging tools. However, for the sake of simplicity, the following restriction is enforced: context switches are not allowed; thus the scheduler has to find execution paths that do not yield local deadlocks. We simulated the execution of an H.264 video decoder based on an STMicroelectronics implementation.

To evaluate the impact of hint parameters, we used a specific feature of modern video coding algorithms: the type of prediction. Basically, each macroblock (MB) can be coded from either a neighboring MB in the same picture (INTRA prediction) or an MB from a previously decoded picture (INTER prediction). The latter is supposed to be longer since it requires a memory access to the buffer where the previously decoded picture is stored and the computation involved is more complex. Therefore, hint parameters were used to let the scheduler know that firing an actor on an INTER-predicted MB is on average 4 times longer than on an INTRA-predicted one[9].

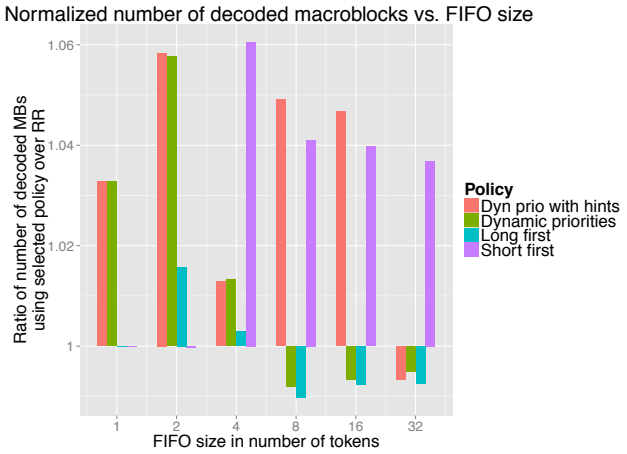
The experiments we carried out consisted in feeding our application model with a hypothetical sequence of MBs alternating INTER and INTRA prediction, and measuring the throughput expressed in total number of decoded MBs during a fixed amount of scheduling steps[10]. We compared the performances of the five scheduling strategies presented in Section IV-B using round-robin as reference. The execution times of the filters were generated using the scheme described in Section V-A, where $cdf = 1$ for INTRA prediction and $cdf = 4$ for INTER prediction, as discussed above. The parameters of the experiments are the size of the FIFOs and the variance of the Beta law from which rdf is sampled. The results are represented as bar charts where each bar stands for a complete execution. The absence of a bar means the corresponding result is 1 (same performance as round-robin).

2) Interpretation: The first set of experiments consisted in setting the variance and varying the size of the FIFOs. Figure 7 depicts the results. On average, all strategies are better than RR except LF in the lower-variance case. In detail, when the variance is set to 1.5, both DP and DP+H give a higher throughput for lower FIFO sizes, but only SF is consistently better for greater sizes with up to 6% speedups. When the variance is set to 3, on the other hand, both DP and DP+H surpass all other policies except for unit FIFO size. DP is slightly better for shorter FIFOs while DP+H outperforms it for bigger ones and can achieve 9% speedups. On average, all proposed scheduling strategies except SF increase their advantages compared to RR when the variance is set to a higher value.

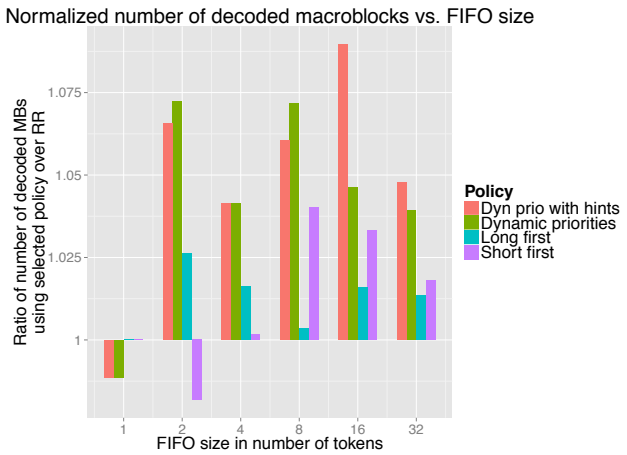
In a second set of experiments, we set FIFO sizes and varied the variance of the execution times. Figure 8 illustrates the results. When FIFO sizes are set to 2, both DP and DP+H reach 7% speedups and outperform all other policies whatever the variance. LF is also consistently better than RR. When FIFO sizes are set to 16, only SF has consistent performance.

[9]This is based upon measures we carried out on real executions of STMicroelectronics' H.264 decoder on the STHORM simulator.

[10]The simulator features no other notion of time than the number of scheduling steps elapsed.

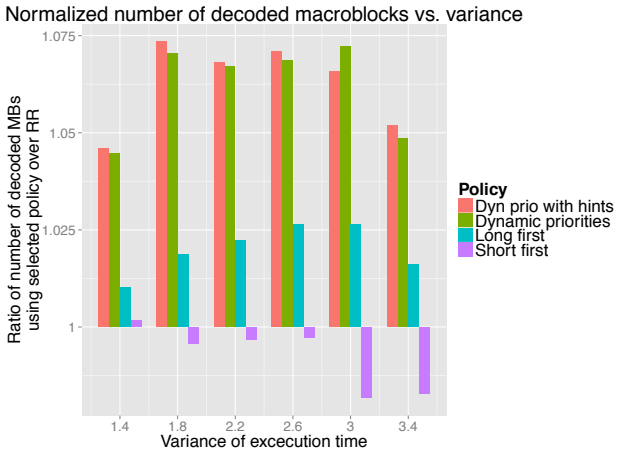


(a) Variance of random execution times = 1.5.

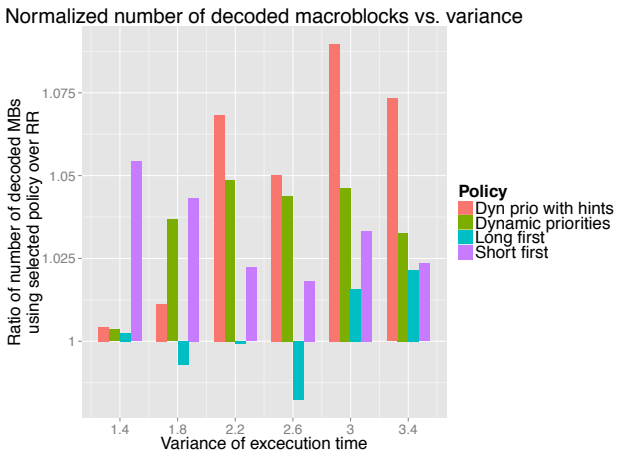


(b) Variance of random execution times = 3.

Fig. 7. Comparison of throughputs of H.264 under various scheduling policies with different FIFO sizes and fixed variances of execution times.



(a) FIFO size = 2.



(b) FIFO size = 16.

Fig. 8. Comparison of throughputs of H.264 under various scheduling policies with different variances of execution times and fixed FIFO sizes.

It is even the best solution for lower variances. On the other hand, for greater values DP+H exhibits higher speedups (up to 9%).

3) Discussion: Overall, DP+H is the most effective scheduling strategy since it leverages both dynamic priorities and hint parameters. The former allow to schedule actors according to FIFO states, thus augmenting the throughput, while the latter bring information on firing durations. Nevertheless, for applications exhibiting low variances in execution times and mapped with large FIFOs—which corresponds to the simplest case—, SF should be preferred. As a matter of fact, it can be shown that scheduling shorter firings earlier allows to optimize resource usage [9, sec. 6.1.2]. Moreover, when dynamic-priority– and hint-parameter–based strategies compare equal in simulation, the latter should be favored since in an actual implementation it will usually be slightly cheaper at run time. This is because dynamic priorities have to be recomputed at each scheduling step, whereas a hint parameter only requires a simple lookup and comparison.

VI. Related Work

Surprising though it may sound, contrary to static MoCs such as SDF [15] and its newer, parameterized derivatives like BPDF [16], the dynamic scheduling of more expressive MoCs has not been well studied. The most advanced research has come from the RVC-CAL [17] and Orcc [18] communities. Their first contribution, as regards dynamic scheduling of dataflow programs [19], is a couple of scheduling strategies: a simple round-robin policy and a combined round-robin/demand-driven/data-driven policy. They also present a distributed scheduler with static mapping. In subsequent work [20], they enable automated mapping of actors through profiling, but still not dynamically at run time. In their latest paper [21], they target specifically MPSoCs but the scheduling and mapping techniques remain the same. Our approach distinguishes itself by targeting heterogeneous platforms and allowing fully dynamic run-time mapping of actors.

As regards Kahn process networks, prior studies [22] advocate data-driven execution and deadlock resolution by increasing FIFO sizes selectively. In terms of practical implementation, lightweight threads and cooperative scheduling can be used to run efficiently a KPN application [23]. To this end,

the execution of the processes is also discretized into separate firings. Dynamic mapping is not considered though.

Finally, the Ptolemy project [24] addresses the general problem of heterogeneous composition of MoCs [25], [26], including dataflow and KPN. The theoretical framework is interesting but Ptolemy is mainly geared towards modeling and simulation rather than software synthesis for real-world implementations.

VII. Conclusion

In this paper, we have presented DKPN, a mixed DPN/KPN execution model for mapping streaming applications on heterogeneous MPSoCs. DKPN enables elegant description and efficient execution of complex programs through a number of properties: strict separation between control (DPN space) and computation (KPN space), hierarchical composition, and high expressiveness thanks to various classes of parameters. In particular, we introduced hint parameters aimed at capturing the dynamics arising from data-dependent behaviors that cannot be predicted statically. Moreover, a cooperative, distributed scheduler framework and five scheduling policies have been sketched to support this execution model. The use of hint parameters by the scheduler has been illustrated. Finally, experiments have shown that the temporal and spatial costs of this scheme are very low, even for static applications, and that hint parameters associated with our advanced scheduling strategies can bring substantial benefits compared to a baseline round-robin approach, especially in the constrained (small FIFOs) and dynamic (high variability) cases.

Future work include testing our contributions on real hardware and with a larger spectrum of application types (e.g. coding algorithms for mobile phones). An interesting extension would be to incorporate timing annotations within DKPN to address real-time issues. In the longer term, integrating DKPN with some high-level synthesis tool would ease the generation of hardware filters with proper interfaces.

Acknowledgments

Thanks are due to Arthur Stoutchinin from STMicroelectronics for helpful discussions and numerous insights, as well as Alain Girault for his thorough review of the model, especially the composition aspect.

References

[1] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, Jul. 2003.

[2] G. Sullivan, J. Ohm, W.-J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 12, pp. 1649–1668, 2012.

[3] A. Jerraya, "Long term trends for embedded system design," in *Digital System Design, 2004. DSD 2004. Euromicro Symposium on*, Aug 2004, pp. 20–26.

[4] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion, "Hybrid Dataflow/von-Neumann Architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, Jun. 2014.

[5] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*, ser. Lecture Notes in Computer Science, P. B. Robinet, Ed. Springer Berlin Heidelberg, Jan. 1974, no. 19, pp. 362–376.

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

[7] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," in *Information Processing '74: Proceedings of the IFIP Congress*, J. Rosenfeld, Ed. North-Holland, 1974, pp. 471–475.

[8] T. M. Parks, "Bounded scheduling of process networks," Ph.D. dissertation, University of California, 1995.

[9] P.-A. Arras, "Ordonnancement d'applications dynamiques à flux de données pour les MPSoC embarqués hybrides comprenant des unités de calcul programmables et des accélérateurs matériels," Ph.D. dissertation, University of Bordeaux, France, Feb. 2015. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01118687/document

[10] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded SoCs: Performance evaluation of visual analytics applications," in *2012 49th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2012, pp. 1137–1142.

[11] S. A. Neuendorffer, "Actor-oriented metaprogramming," Ph.D. dissertation, University of California, Berkeley, 2005.

[12] G. Allen, P. Zucknick, and B. Evans, "A distributed deadlock detection and resolution algorithm for process networks," in *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, vol. 2, April 2007, pp. II–33–II–36.

[13] B. Jiang, E. Deprettere, and B. Kienhuis, "Hierarchical run time deadlock detection in process networks," in *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, Oct 2008, pp. 239–244.

[14] A. Olson and B. Evans, "Deadlock detection for distributed process networks," in *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on*, vol. 5, March 2005, pp. v/73–v/76 Vol. 5.

[15] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *Computers, IEEE Transactions on*, vol. 100, no. 1, pp. 24–35, 1987.

[16] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur, "Bpdf: A statically analyzable dataflow model with integer and boolean parameters," in *Proceedings of the Eleventh ACM International Conference on Embedded Software*, ser. EMSOFT '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 3:1–3:10.

[17] M. Mattavelli, I. Amer, and M. Raulet, "The Reconfigurable Video Coding Standard [Standards in a Nutshell," *IEEE Signal Processing Magazine*, vol. 27, no. 3, pp. 159–167, May 2010.

[18] H. Yviquel, A. Lorence, K. Jerbi, G. Cocherel, A. Sanchez, and M. Raulet, "Orcc: Multimedia Development Made Easy," in *Proceedings of the 21st ACM International Conference on Multimedia*, ser. MM '13. New York, NY, USA: ACM, 2013, pp. 863–866.

[19] H. Yviquel, E. Casseau, M. Wipliez, and M. Raulet, "Efficient multicore scheduling of dataflow process networks," in *Signal Processing Systems (SiPS), 2011 IEEE Workshop on*, 2011, pp. 198–203.

[20] H. Yviquel, E. Casseau, M. Raulet, P. Jaaskelainen, and J. Takala, "Towards run-time actor mapping of dynamic dataflow programs onto multi-core platforms," in *2013 8th International Symposium on Image and Signal Processing and Analysis (ISPA)*, Sep. 2013, pp. 732–737.

[21] H. Yviquel, A. Sanchez, P. Jääskeläinen, J. Takala, M. Raulet, and E. Casseau, "Embedded Multi-Core Systems Dedicated to Dynamic Dataflow Programs," *Journal of Signal Processing Systems*, pp. 1–16, Oct. 2014.

[22] T. Basten and J. Hoogerbrugge, "Efficient execution of process networks," *Proc. of Communicating Process Architectures*, pp. 1–14, 2001.

[23] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele, "Efficient execution of Kahn process networks on multi-processor systems using protothreads and windowed FIFOs," in *IEEE/ACM/IFIP 7th Workshop on Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009*, Oct. 2009, pp. 35–44.

[24] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[25] A. Goderis, C. Brooks, I. Altintas, E. A. Lee, and C. Goble, "Heterogeneous composition of models of computation," *Future Generation Computer Systems*, vol. 25, no. 5, pp. 552–560, 2009.

[26] E. A. Lee, "Disciplined Heterogeneous Modeling," in *Proceedings of the 13th International Conference on Model Driven Engineering Languages*

and Systems: Part II, ser. MODELS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 273–287.