Deterministic Memory Sharing in Kahn Process Networks: Ultrasound Imaging as a Case Study

Andreas Tretter, Harshavardhan Pandit, Pratyush Kumar and Lothar Thiele
Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
firstname.lastname@tik.ee.ethz.ch

***Abstract*—Kahn process networks are a popular programming model for programming multi-core systems. They ensure determinacy of applications by restricting processes to separate memory regions, only allowing communication over FIFO channels. However, many modern multi-core platforms concentrate on shared memory as a means of communication and data exchange. In this work, we present a concept for deterministic memory sharing in Kahn process networks. It allows to take advantage of shared memory data exchange mechanisms on such platforms while still preserving determinacy. We show how any Kahn process network can be transformed to use deterministic memory sharing by giving a set of transformations that can be applied selectively, only looking at one process at a time. We demonstrate how these techniques can be applied to an ultrasound image reconstruction algorithm. For an implementation on a test system, our technique yields significantly better performance combined with a drastically smaller memory footprint.**

I. Introduction

Multi-processor systems have been widely accepted as the only possibility of keeping up with the increasing demands for computation power, especially coming from multimedia applications. Yet, programming these systems is an intricate task. The danger of races, glitches, deadlocks or other synchronisation problems is always present.

The Kahn Process Network (KPN) programming model [1] avoids these issues by limiting the communication possibilities of each core. Essentially, a KPN is a directed graph, the nodes of which are called *processes* and the edges of which are called *channels*. A process is an independently working thread, which is only allowed to access its local memory. Communication between processes has to be done through the channels. A channel is a First-In-First-Out (FIFO) buffer that takes *tokens* from its source and and releases them to its destination. While a write operation (putting tokens on a channel) is always possible, a read operation (receiving tokens from an incoming channel) will block the process until tokens are present in the channel. The process is not allowed to check if a channel is filled with tokens or not. It is known, due to Kahn [1], that KPNs are always *determinate*, i.e., the tokens sent over the channels only depend on the functionality of the processes, but not on scheduling or other execution parameters. This makes them suitable for modelling highly parallelisable applications on multi-core architectures.

While KPNs can be thought of as advocating message passing to correctly manage concurrency, the current trend in hardware platforms, especially in the embedded domain, goes into another direction. Many of the modern multi-core platforms (e.g. STHORM, Kalray MPPA, TI Freescale or ARM Multicore) count on shared memory architectures for data exchange. These platforms minimise the latency of accessing shared memory using hardware features such as crossbar communication, multi-banked memory modules, and hierarchical cache architectures. Applications with high communication demands need to make use of these features in order to attain maximum efficiency on such platforms. Traditional KPNs, however, explicitly do not allow this. It would thus be desirable to combine the determinacy of KPNs and the improved performance of memory sharing on modern multi-core architectures.

In this paper, we propose a set of transformations that enable use of shared memory communication patterns without affecting the determinacy of KPNs. We call this Deterministic Memory Sharing (DMS). There are four primary features of DMS. First, we propose transformations to convert any standard channel of a KPN to allow a memory block to be shared between the producer and consumer processes of the channel. Second, we propose transformations to allow multiple processes to concurrently read from a shared memory block. We formalise the intricate conditions under which such concurrent reads can be allowed. Third, we propose using memory blocks for in-place modifications and direct re-transmissions. Fourth, motivated by streaming applications which expose data parallelism, we propose dividing a memory block into smaller sub-blocks, which can be concurrently read from and written to by different processes. In addition, we propose the insertion of *recycling* channels which significantly reduce the cost of allocation and deallocation of the shared memory blocks by allowing reuse of memory blocks once they are not used by any process. All the transformations mentioned above have been conceived such that they can be employed selectively to transform a standard KPN into a DMS-enabled KPN, sequentially and only looking at one process at a time. We show that applying each set of transformations preserves the determinacy of the KPN, by construction.

We illustrate DMS with an ultrasound image reconstruction algorithm, which is representative of most streaming applications: There is a large communication overhead, and explicit data and task parallelism. We show that a subset of our proposed transformations can be employed to correctly transform the KPN model of the algorithm. We implement the original KPN, the transformed DMS-enabled KPN, and a windowed-FIFO-based KPN [2], using the DAL framework [3], on the Intel Xeon Phi processor. With extensive experimental tests we conclude that sharing memory enables a higher throughput of the application, while using a smaller memory footprint.

The remainder of the paper is structured as follows: In the next section, related work is reviewed. In Section III, the ultrasound image reconstruction algorithm is presented in detail. In Section IV, the basic ideas of memory sharing in KPNs and how this can be done deterministically will be explained. In Section V, these ideas are formalised. In Section VI, transformations will be given for existing KPNs to apply these ideas and to optimise the resulting networks. In Section VII, the correctness of these transformations will be shown. In Section VIII, the ultrasound imaging algorithm is revisited and it is demonstrated how the transformations from the section before can be applied to it. Section IX will show how the concept can be implemented in C code. Finally, experimental results are presented in Section X.

II. Related Work

The memory related publications in process networks can be divided into three groups. The first group regards channel capacities (i.e. the amount of tokens the actual implementations

978-1-4799-6307-2/14/$31.00 ©2014 IEEE

of the individual KPN channels can hold), usually trying to minimise the memory footprint via these capacities. The second group tries to further reduce the memory footprint by reusing the same memory for multiple channels. A third group finally tries to avoid unnecessary copying overhead rather than looking at the memory consumption.

In the first group, which regards channel capacities, one idea is to start with small channel capacities, dynamically increasing them as required [4], [5]. Another approach is to entirely eliminate certain channels by automatically merging processes where appropriate [6]. For networks with regular patterns, such as synchronous dataflow [7] or cyclo-static dataflow [8], the minimal channel capacities can be calculated at design time. There are a number of approaches which try to further reduce these minimal capacities for special cases of these dataflow graphs [9], [10] or performing special analyses [11].

All the methods mentioned so far are complementary to our work; we assume these optimisations, if required, to have been carried out prior to applying the transformations presented here.

Another work on channel capacities, however focusing on their relation to application performance, is [12]. While this relation is regarded as a trade-off there (more memory for better performance), we can reduce the memory footprint of an application and achieve a better performance at the same time.

In the second group, which reuses buffers for multiple channels, [13] tries to minimise the memory footprint of synchronous dataflow graphs. While greatly reducing memory requirements, this only works for single-core systems and with pre-determined schedules. It is not discussed if this technique could be applied to multi-core systems; however, that endeavour would yield the danger of massively inhibiting parallelism, thereby losing performance.

[14] uses a global memory manager. Processes can obtain buffers through the memory manager from so-called pools. The programming model used there is not compatible to KPNs, though; in fact, it is non-deterministic. Also, it is the task of the programmer to decide which pool to obtain buffers from or how many buffers these pools should be provided with. In this work, we show how a deterministic, DMS-enabled application can be obtained from any KPN by applying simple transformations. No complicated synchronisation or buffer allocation decisions have to be taken care of by the programmer.

In [15], the SDF model is extended with a sort of global buffers for keeping track of common global states such as sampling frequency or gain in a multimedia stream. The motivation of this work are synchronisation purposes; memory footprint or performance are not taken into account.

The papers from the third group do, although often achieving it, not primarily target a low memory footprint. Their main idea is rather to boost the application performance by avoiding unnecessary copying overhead. In [2], this is done by using so-called *windowed FIFOs*, which can replace the standard FIFOs in KPN channels. Instead of copying the tokens from a sender or to a receiver process, a windowed FIFO provides these processes with a shared memory region (a *window*) they can directly write to or read from. Managing the access to this window, appropriately, ensures that processes do not overwrite unread data or do not read stale data. This saves a certain amount of copying overhead between two processes; still, when the data has to be passed on to a third process, copying cannot be avoided.

A more general approach is discussed in [16]. The idea there is that processes can allocate memory blocks, and then send tokens representing these blocks over the channels. The token gives a process the permission to access the corresponding memory block. Further, processes can send read-only copies of tokens to multiple receivers. However, it is not clearly mentioned whether the data in these memory blocks can be edited *in-place* and then sent on to another process. Also, the notion of block allocation and deallocation is only abstractly defined; using memory allocation functions provided by an operating system would be rather slow for regular allocation as part of a data streaming process network.

In this paper, we generalise the concepts from the third group of works mentioned here and we discuss several implementation details. We show simple transformations allowing to turn a traditional KPN into one using shared memory, still staying deterministic. Also, we introduce new techniques, such as recycling channels and memory sub-blocks. Furthermore, formalise the concept of in-place editing in KPNs and show how it can be implemented.

III. The Image Reconstruction Algorithm

In this section, the ultrasound image reconstruction algorithm we use for our experiments will be described in detail. It will be shown what it does and how it can be implemented as a KPN.

The different hardware variants, methods, reconstruction algorithms and parameters of ultrasound imaging are as manifold as the applications in medicine and in other domains. In this paper, we will limit ourselves to one single configuration, which we implement in different ways. First, we will explain the general principle of ultrasound imaging. Then, we will give details about the individual steps to be performed during image reconstruction. Finally, we will show how the whole algorithm can be implemented efficiently.

A. Principles behind ultrasound imaging

In the medical domain, one typically uses sound waves with a frequency of 1 to 50 MHz, which are simplifyingly assumed to travel at constant speed through human tissue. At every boundary of materials with different physical properties, transmission, absorption and reflection occur. The latter effect is taken advantage of for ultrasound imaging.

The tool for obtaining the images is called a *probe* and, in our case, is an array of linearly arranged piezoelectric crystals called *transducers*. A transducer changes its shape when subjected to an external voltage and can therefore be used to generate sound waves. Conversely, when changing its shape due to mechanical pressure (like sound waves), it produces a voltage which can be measured.

The image capturing process now works as follows. First, a plain wave signal is sent out from the transducers; this signal is e.g. a short window of a sine wave. As the wave travels through the tissue, it gets reflected varyingly strongly at the different locations. After sending out the signal, no more voltage is applied to the transducers and instead, the voltage generated by the transducers is measured over time. For n transducers, this gives n individual traces of recorded sound waves. From these traces, a two-dimensional image is reconstructed. This can be done using the algorithm which we implement and optimise in this paper, and which is described in the next section.

B. Individual steps of the algorithm

The image reconstruction can be decomposed into multiple independent steps, which are explained in the following.

Attenuation compensation: The longer a wave travels through the tissue, the more it gets attenuated. This attenuation can be calculated and reversed on each trace by multiplying the samples with an exponentially growing function.

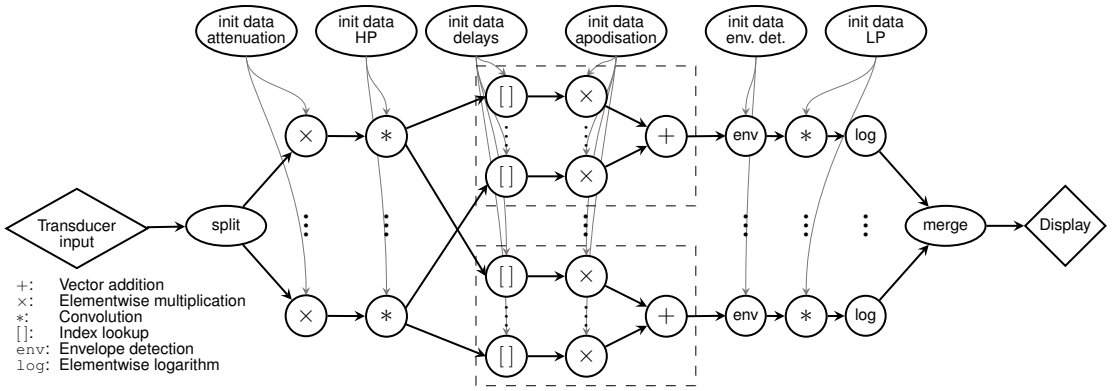


Figure 1: KPN implementation of the ultrasound image reconstruction algorithm. The dashed rectangles group all the processes involved in the beamforming of one image column each.

High pass filter: Each trace is convolved with a high-pass filter to eliminate DC biases.

Beamforming and Apodisation: This is the most important reconstruction step. When the signal is reflected, the reflection arrives at all transducers, however at different points in time due to the different geometrical distances between the reflection origin and the transducers. For every geometrical position, one can calculate at which times a reflection from there arrives at the individual transducers. The different samples at these times are summed up for all positions considered, thus creating a first image. The image quality can be improved by weighting the different samples according to the angle in which the reflection hits a transducer. This is called apodisation. In practice, for each transducer, one image column is calculated such that all the samples from the transducers trace can be used. For each column, this can be achieved by taking the prepared traces from all transducers, extracting samples at precalculated indices, multiplying the extracted samples with precalculated factors and finally summing the results up. All these operations are element-wise vector operations.

Demodulation: The beamformed image still contains the sine waves of the echoed signal. These are removed by applying an envelope detection and a low-pass filter. Both can essentially be implemented as a convolution.

Log compression: To stress differences at weaker reflections, the logarithm is taken of each point in the image.

C. KPN implementation of the algorithm

Fig. 1 shows how the ultrasound image reconstruction can be implemented as a KPN. The processes in the uppermost row precalculate certain vectors and tables like the filter kernels, the delay indices for the beamforming process or the apodisation tables. This precalculated working data is then sent to all the processes which need it. The processes will read the data once and keep it stored; afterwards, they will no longer read from those channels and instead keep working on the incoming samples. An input process obtains the data from the transducers, which is then split into the individual transducer traces and sent through the channels. The working processes themselves accomplish rather simple work like convolutions, element-wise multiplications or index-lookups. At the end, the final image is merged together.

IV. Deterministic memory sharing

It can be easily seen from the last section that the ultrasound image reconstruction algorithm works on large amounts of data, which need to be exchanged between different cores on multi-core systems. Also, multiple processes have to work on the same data. In a traditional KPN, all this data has to be sent over the channels, and there has to be a separate copy of it for each process. Clearly, this leads to a considerable overhead. We try to avoid this overhead using a different method of data exchange that is based on memory sharing.

This section explains how memory blocks can be shared by multiple KPN processes while still preserving the advantages of KPN, in particular its determinacy. We will introduce the basic ideas of KPN memory sharing and that of an efficient memory management technique.

A. Basic idea of the model

As previously mentioned, our approach is to have multiple KPN processes share certain memory blocks. In general, however, when multiple processes share one memory block, they can communicate through it, thereby circumventing the actual KPN communication mechanism (which uses channels). This would not only destroy the determinacy of the process network, it would also reintroduce races, glitches and all the other multi-processor issues KPN originally set out to avoid. Therefore, it is essential to have a synchronisation mechanism which regulates the accesses to shared memory blocks.

This can be done by the concept of *access tokens*. An access token gives a process the right to access (i.e. read from and write to) a certain memory block. There is only one access token for each memory block, and a process is not allowed to create copies of it. This ensures that only one process can access a memory block at a time. Access tokens can be sent to other processes over the conventional KPN channels; the sending process has to destroy its local instance of the token once it has sent it (s.t. there are no two copies of the token).

In summary, instead of sending data directly over a channel, the data is stored in a memory block, the access token to which is then sent over the channel. This is illustrated in Fig. 2. We will show in Section VII that the determinacy property of KPN still holds when sharing memory through access tokens.

B. Relaxations and additions to the model

The mechanism presented above already brings clear improvements, but there are more possibilities of eliminating overhead. In this section, we will discuss how it can be relaxed in order to allow further useful optimisations.

1) Multiple memory copies: One desirable feature would be to avoid multiple copies of the same data. This can be achieved by allowing multiple processes to simultaneously share one memory block. However, that leads to problems when these processes simultaneously write (or read and write) the same memory locations. As it was shown above, this would violate the determinacy of the KPN. On the other hand, it is not a problem if multiple processes concurrently *read* from the same memory locations. Thus, it is possible to relax the uniqueness constraint for the access tokens in a way that access

Figure 2: Two KPN approaches: a) Classic process network; b) Process network with shared memory blocks. The spade represents an access token linked to a memory block with the "classic" token from before.

tokens can be duplicated if it can be guaranteed that no write accesses are performed to the memory blocks they are linked to. Conversely, one can say that memory blocks may not be written to if multiple access tokens are linked to them. This relaxation does not compromise the determinacy of the KPN for the simple reason that no communication can be established by only reading.

A typical use case for duplicating access tokens could be as follows. Some data is produced and written to a memory block. Once the writing is finished, the producing process duplicates the access token multiple times and distributes the access tokens linked to the memory block to multiple receivers, which can then read it simultaneously.

2) Different levels of process granularity: A second relaxation that can be made to the access-token principle helps to deal with different levels of granularity of different processes. Depending, e.g., on the workload of different tasks, it may be advantageous if one process works on a large amount of data and afterwards multiple processes work on distinct subsets of this data. After that, it may be desirable again to have one more process working on the entire set of data. This can be allowed if it is ensured that these subsets are distinct, i.e. that in the memory block, the locations accessed by the different processes do not overlap. Again, it must be ensured that these processes cannot communicate through shared memory blocks.

To this end, we introduce the notion of *memory sub-blocks*. A memory block can be split up into multiple sub-blocks, which denote distinct, non-overlapping memory regions in the original memory block. Every sub-block can now have its own access token. The different access tokens can be sent to different processes, which can then only access different memory regions. Thus, no communication between them through the memory block is possible.

It is also possible to introduce a reverse operation to the *split* mechanism, which we will call *merge*. The merge operation can join two (or multiple) adjacent memory sub-blocks to one single (sub-)block, reducing the set of access tokens provided to it to one single access token.

3) Memory recycling: Until now, memory blocks have been discussed as something that exists, but without mentioning where they actually come from. The conventional way of obtaining them is through dynamic allocation [16]. Similarly, they are deallocated when no more access token is linked to them. This, however, may be rather slow on many systems or even not supported by the underlying software stack. Thus it appears sensible to look for alternatives to dynamic memory allocation.

One such alternative would be to introduce a *recycling channel* which goes from a consuming to a producing process. Instead of deallocating memory blocks, the consuming process sends the access tokens linked to them back to the producing process for later use. Initial access tokens (with corresponding memory blocks) are placed on the recycling channel such that the producing process can always use this channel as a source for obtaining (access tokens to) memory blocks. We call this technique memory block *recycling*.

In general, this change to the process network also changes the semantics of the program, especially when there is no more access token on the recycling channel and the consuming process blocks trying to obtain one. However, we will show that under certain conditions, the same change in semantics is also induced by using channels with limited capacity (which is anyway necessary when actually implementing a KPN).

V. Formalisation of the model

In the previous section, we have informally described the different ideas DMS is based on. Now, we are going to give a formal definition of all the mechanisms included. This will help in the next sections, when we discuss KPN transformation methods and show their correctness. For this purpose, we will first define the data structures involved and the properties they have. Afterwards, the different operations on these data structures will be defined.

The model works on *memory blocks*, which are regions of memory that can be shared between processes. B is the set of memory blocks. For each block $b \in \mathrm{B}$, there is

- $\mathrm{size}(b) \in \mathbb{N}$, the size of the memory block
- $b[n]$, $n \in \{0..\mathrm{size}(b)-1\}$, the access operator for the memory block. It returns a memory location which can be read or written.

The processes do not have direct access to the memory blocks. They can only access the blocks through *access tokens*. An access token is an abstract entity with the following properties:

1) It is linked to a memory block.
2) It allows read accesses to the block.
3) It only allows write accesses to the block when it is the only access token linked to the block.
4) It can be sent over KPN channels.
5) Send operations are destructive, i.e. the sending process does not retain a copy of the token sent.

T is the set of *access tokens* which currently exist in the application. For each $t \in \mathrm{T}$, there is

- $\mathrm{link}(t) \in \mathrm{B}$, the memory block the token is linked to.
- $t[n] := \mathrm{link}(t)[n]$, the access operator for the access token.

The subject of write accesses being allowed or not needs some more discussion. This is a global property, which we call *ownership*: A process *owns* a memory block if it has the only access token linked to that block. One could have mechanisms to check for ownership at runtime before each write operation. This, however, would have to be done carefully in order not to allow global communication through this checking mechanism. In this work, the approach is to formally ensure at design time that write accesses are only performed when they are allowed.

For defining split and merge operations later, we also need the concept of a *sub-block*. A sub-block is a part of a memory block, which can be created by splitting a memory block. A sub-block behaves like a normal block, in particular it can have access tokens linked to it.

With these definitions, we can now describe the set of DMS operations that can be executed by the processes.

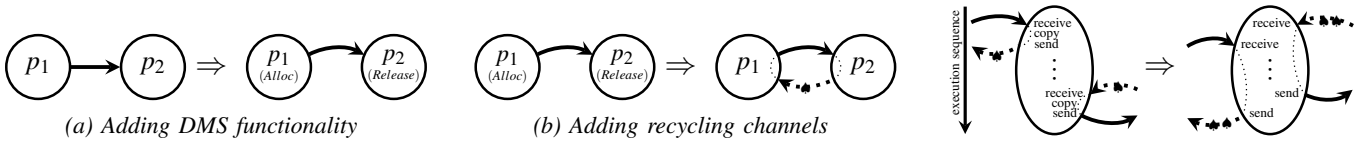
Allocation: Creates a new memory block of a given size and returns an access token to it.

Duplication: A copy of a given access token is created, thereby ending a possible ownership of (and thus inhibiting further write accesses to) the memory block linked to the token.

Splitting: The memory block (or sub-block) a given access token is linked to is split into two or more sub-blocks. The access token which was provided is destroyed. Instead, access tokens to the sub-blocks are returned. For simplicity reasons, we demand that the calling process must own the memory block for splitting. All sub-blocks created are then owned by the calling process, but can later also be owned each by different processes.

Merging: Two or more sub-blocks of the same memory block that are adjacent in memory are merged together to a bigger sub-block or back to the entire memory block. The calling process must own all the sub-blocks. The access tokens which were provided are destroyed. Instead, an access token to the merged (sub-)block is returned.

Release: Destroys an access token. If the calling process owns the memory block linked to the access token, this block is destroyed as well. In the case of a sub-block, destruction only



(a) Adding DMS functionality *(b) Adding recycling channels* *(c) Reordering channel accesses*

Figure 3: Illustrations of the basic transformations introducing DMS to a KPN. Recycling channels are shown as dotted lines.

happens to the entire memory block once the last access token linked to it or one of its sub-blocks is released.

These operations can now be used for implementing a DMS-enabled KPN.

VI. Translation to DMS

In the last sections, the idea of DMS was explained and formally described. However, it is not clear yet how it can be applied and in particular how an existing KPN can be transformed to use DMS. Note that the transformation process must be organised such that the generated code follows DMS rules and that the semantics of the original KPN are preserved (no deadlocks are introduced etc.).

It is important to note that the translation is not an all-or-nothing operation; in fact, it may sometimes be advantageous to convert a KPN only partially. Note that due to the high expressiveness and the intricate process interactions of KPNs, it is not possible in general to always implement the optimisation ideas shown in Section IV.

This section will introduce a set of transformations that can be carried out in process networks and we will try to establish an intuitive understanding for them. We will show in the next section that they all are correct and do not change the semantics of the process network.

We have conceived the translation such that it works step by step, each time applying one transformation. The translation is performed in three stages:

1) Basic transformations: All channels are transformed to DMS for which this is desired. Recycling channels can be added.
2) Optimisation: The performance of the application is increased and its memory footprint is decreased by taking advantage of techniques like splitting and merging or token duplication.
3) Final clean-up: The attained process network is simplified.

In the following, the transformations in each stage shall be discussed. Afterwards, it is shown how the optimisation transformations can be coordinated.

A. Basic transformations

The basic transformations are described below and illustrated in Figs. 3a to 3c. We assume each channel to have been assigned a maximum capacity. (This is necessary for any implementation. Dynamically increasing channel capacities later as in [4], [5] is also possible with our approach.)

Converting classic channels to DMS channels: For every channel in the network that is intended to use DMS, its sending process and its receiving process are altered such that they send and receive access tokens instead of traditional "data" tokens. The access tokens are obtained by allocating memory blocks in the sending process and directly released after reading in the receiving process.

Adding recycling channels: For each channel converted to DMS as shown above (referred to as *data channel*), a recycling channel can be added. The recycling channel goes into the opposite direction of the data channel and has the same capacity. Initial access tokens linked to separate memory blocks are added to the recycling channel such that the number of initial tokens on both channels is equal to the capacity of the data channel. Releasing tokens and memory block allocation are replaced with sending and receiving tokens from the recycling channel, respectively.

Reordering channel accesses: After applying the transformations above, accesses to a data channel and its corresponding recycling channel happen in pairs, i.e. one channel is accessed directly after the other, with no other channel access in between. This means that only one access token is available to a process at a time. Simultaneous access to two or more memory blocks can be achieved by moving reads from or writes to recycling channels such that they happen earlier or later in the execution path, respectively. However, an additional initial access token may have to be added to the recycling channel in order to prevent a change in the semantics of the process network.

B. Optimisation transformations

For the optimisation transformations, we will give a non-exhaustive list of the most common optimisations that can be applied to a DMS-enabled KPN. We will assume recycling channels have always been added to the channels involved (the other case can be easily derived). All these transformations can be applied by looking at one process and a *subset* of its data channels. We look at processes which access all the channels in this subset only in the form of *elementary transactions*, which consist of reading/writing exactly one token from/to each channel. Note that this only restricts the access pattern of a process concerning the *subset of channels considered*. Its other behaviour – in particular, its accesses to other channels – does not play a role.

We look at three groups of transformations here, which are illustrated in Figs. 4a to 4d on the following page.

In-place editing: If a process always reads from one channel c_i and writes to another channel c_o, and if the operations to the two memory blocks concerned are such that they could happen *in-place* in only one memory block, then the process can be altered such that it performs the in-place operation on the memory block received from c_i and then sends the access token to c_o. The recycling channels corresponding to c_i and c_o are joined as shown in Fig. 4a, with their capacities and number of initial tokens adding up for the joint recycling channel.

Splitting and merging: These transformations work in a similar way as in-place editing. The difference is that in the case of split, the input memory block is split into smaller sub-blocks, which are then distributed to multiple output channels. In the case of merge, multiple input-sub-blocks are merged to give the output block. As sub-blocks can only be merged when they belong to a common memory block, a dummy split process has to be generated as a part of the recycling infrastructure when applying a merge transformation (cf. Fig. 4c). In the case of a split transformation, a dummy merge process is generated to ensure correct memory block recycling (cf. Fig. 4b).

Duplicating access tokens: If a process always sends the same data to multiple channels, it can be transformed such that it only allocates one memory block which is filled with this data. The access token to this block is then duplicated and sent to each of the channels. As with split, an additional dummy process is generated in order to collect all these duplicate tokens again, such that the memory block can be safely recycled (cf. Fig. 4d).

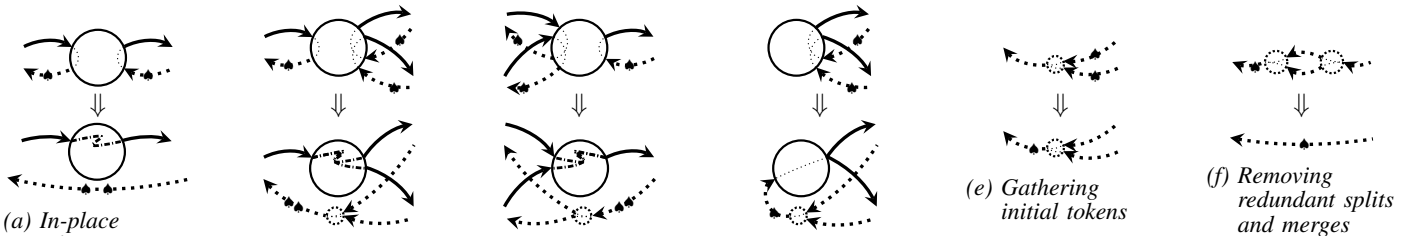


Figure 4: Illustrations of different DMS transformations. Dummy processes generated during transformation are shown as dotted circles.

C. Simplifications

Simplification transformations become necessary due to the overhead caused by the previous optimisations. While this overhead is necessary to ensure correctness of the optimisations, it can be safely eliminated after they are finished. We limit ourselves to two important simplifications, which are illustrated in Figs. 4e and 4f.

Gathering initial tokens: Initial access tokens should always be linked to entire memory blocks, not to sub-blocks. Therefore, initial access tokens linked to sub-blocks are moved behind a merge or in front of a split. This implies a merge operation on the initial access tokens, as illustrated in Fig. 4e. Should the number of initial tokens on different branches differ, this situation can be resolved by adding additional initial access tokens to certain branches.

Removing split and merge processes: In certain situations, a dummy split and a dummy merge process just neutralise each other after a sequence of transformations. In that case, both can be removed and the channels connected to them are joined, as shown in Fig. 4f.

D. Optimisation coordination

All of the optimisation transformations shown above can be applied if their requirements are met; however, not all of them can be applied together. For instance, an in-place edit transformation is not allowed after an access token has been duplicated. As transformations are applied individually to the processes, one after another, a mechanism is required which keeps track of the transformations applied and reveals which transformations are still allowed. In particular, one must be able to prove at design time whether or not a process, after a given sequence of operations, owns the memory blocks arriving from a certain channel. For this purpose, we use an *ownership annotation* of the channels: $\text{own}(c) \in (0, 1] \cup \{*\}$ for every channel c using DMS.

Directly after applying the basic transformations to a channel, its target process always owns the memory blocks sent over it. On the other hand, it only reads from the blocks and then recycles them, i.e. it does not need to own them. Therefore, the ownership annotation is $*$ directly after the basic transformation to indicate that optimisations changing the ownership are still possible.

When a process needs ownership of the tokens coming through a channel (i.e. for in-place edit, merge and split transformations), the annotation of the channel is changed to 1 during the transformation. When a token is duplicated, the channels which the duplicates go to are marked with an ownership < 1. Once a channel has been annotated with an ownership other than $*$, this annotation must not be changed any more. Any transformation may only be applied if the corresponding annotations are still possible or if the channels already have the annotation the transformation would entail.

In case of token duplications or collecting duplicates, the exact value of the ownership annotation is determined such that the sum of the annotations of the outgoing channels is always equal to that of the incoming channels carrying access tokens linked to the same memory blocks. In case of token duplication, this means that the annotation value of the incoming channel (or recycling channel, i.e. 1) is divided by the number of duplicates. In case of collecting duplicates, the annotation values of the incoming channels are added up to obtain the value for the outgoing channel. This is illustrated in Fig. 5. As the ownership value is always one before the first duplication, the sum of the ownership values after any combination of duplications and collected duplicates is always equal to one. If all the channels transporting a duplicate of an access token are collected again, the resulting ownership value must be one. Conversely, if there is one duplicate channel which has not yet been collected, this ownership value cannot be one because the ownership value of the uncollected channel is greater than zero by definition. Remember that an ownership annotation of one for a channel means that its target process owns the memory blocks linked to the tokens it receives from that channel.

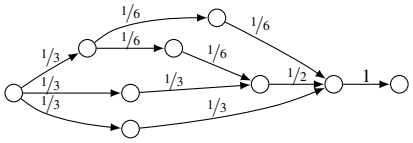


Figure 5: Example for the ownership annotation of channels. The graph shown only contains channels transporting duplicate access tokens linked to the same memory blocks. Note that the rightmost process owns the memory block.

VII. Correctness of the Translation

In this section, we show that the translation described in the previous section is correct. This comprises three points:

1) Memory integrity: We show that all memory blocks are correctly deallocated.
2) Determinacy: We show that the modified KPN is still determinate.
3) Semantics: We show that the semantics of the original KPN are preserved.

The first two points will be shown in general, whereas the third point will be shown to hold for each transformation individually.

A. Memory integrity

If an access token is sent to a channel – including recycling channels –, the memory block linked to it stays in use and must not be deallocated. If an access token is released, the memory block linked to it will be deallocated if it is no longer in use, as described in Section V.

With the transformations shown in this work, one of both is always the case in each process for each access token. Therefore, no memory leaks can occur among the memory blocks allocated using the DMS mechanisms.

B. Determinacy

To show the determinacy of the modified KPNs, we have to show that they meet two properties:

1) Communication only happens through channels.
2) Read accesses on the channels are blocking and destructive.

Then, determinacy follows from [1].

The second property is inherited from the underlying KPN channels, which still transport the access tokens.

The first property is met because a process can only write to a memory block if it owns it, i.e. if no other process can access the block. The only possibility for the process of making the data available to other processes is to send the access token over a channel. At this point, it loses the access to the memory block and thus cannot use it for any further communication. Therefore, any data transfer needs a sending operation over a KPN channel.

C. Semantics

In the following, we show that each of the transformations showed in Section VI preserves the semantics of the process network, i.e. that the same data is sent over the channels and that no deadlocks are introduced by the transformations[1]. This makes it necessary to also formalise the prerequisites that must be met for the transformation to be allowed as well as possible annotations made during a transformation.

First, we have to specify more exactly the patterns of the processes which we are looking for during the optimisation phase. One pattern which is required by all the optimisation transformations is what we call *regular behaviour*.

Definition 1 (Regular behaviour)**.** A process p performs an *elementary transaction* on a subset C_i of its input channels and a subset C_o of its output channels iff

- it reads exactly one token from each channel $c \in C_i$ and releases/recycles it after usage and afterwards
- it allocates/obtains through recycling exactly one memory block for each $c \in C_o$ and sends it over that channel.

p behaves regularly on C_i and C_o iff all accesses to any of the channels in C_i and C_o are part of an elementary transaction.

With the following definitions, we generally describe another major pattern which is part of many rules below. We are looking for operations that can be performed in-place. The criterion for this is as follows:

Definition 2 (Inplaceable behaviour)**.** The behaviour of a process p on an input channel c_i and an output channel c_o is *inplaceable* iff

- p behaves regularly on $\{c_i\}$ and $\{c_o\}$ and
- p never writes to memory blocks received from c_i and
- in every elementary transaction, a being the memory block received from c_i and b being the block sent to c_o, for every $k \in \mathbb{N}$, p never reads $a[k]$ after writing $b[k]$.

If this is the case, one can set $a = b$ because (i) before any write operation to a location $b[k]$, $b[k]$ is still undefined and $a[k]$ contains the expected value and (ii) after a write operation to a location $b[k]$, $b[k]$ contains the expected value and $a[k]$ is not accessed any more.

To extend this definition for split and join operations, we gather multiple channels to one virtual channel, which we call *compound channel*. A *compound channel* over a tuple of channels is interpreted such that it reads/writes one token each from/to each channel of the tuple, virtually linking them to a *compound block*. For a tuple of n different memory (sub-)blocks $b_1 ... b_n$, the compound block k over these blocks then has an access operator defined as:

$$k[s(j)+l] = b_j[l] \;\; \forall l \in \{0..\text{size}(b_j)-1\} \;\; \forall j \in \{1..n\}$$

with $s(j) = \sum_{i=1}^{j} \text{size}(b_i)$.

With this notion, the definition of inplaceable behaviour naturally extends to sets of input and output channels:

Definition 3 (Inplaceable behaviour on channel sets)**.** The behaviour of a process p on a subset C_i of its input channels and a subset C_o of its output channels is inplaceable iff there exists a permutation x_i of C_i and x_o of C_o such that the behaviour of p is inplaceable on the compound channel over x_i and that over x_o.

With these definitions, we can give exact specifications of the transformations already shown in Section VI and draw conclusions about their correctness:

Transformation 1 (Converting a KPN channel to a DMS-enabled channel)**.**
Prerequisites: Channel c not using DMS
Annotations: $\text{own}(c) := *$
This transformation is always valid, since any data that can be sent using normal KPN channels can also be sent using DMS.

Transformation 2 (Adding recycling channels)**.**
Prerequisites: DMS-enabled channel c with fixed capacity
This transformation's influence on the process network semantics is identical to that of introducing feed-back channels as described in [4]. There, feed-back channels are used to model the fixed capacities of KPN channels. As we assume a fixed capacity for c, the transformation is neutral to the semantics of the process network.

Transformation 3 (Simultaneous access to memory blocks)**.**
Prerequisites: A process p accessing multiple channels, at least one of which uses DMS
As mentioned in Transformation 2, a recycling channel models the fixed capacity of its corresponding data channel. Postponing a send operation to a recycling channel thus delays the provision of channel capacity after a (destructive) read. Similarly, preponing a receive operation from a recycling channel brings forward the beginning of a write operation in the sense that channel capacity is claimed. A possibly blocking operation between a pair of data and recycling channel accesses may therefore, in connection with other processes, lead to deadlocks. As these deadlocks, however, are only related to a limitation of virtual channel sizes, they can be prevented by increasing these virtual channel sizes by adding initial access tokens to recycling channels.

Transformation 4 (In-place editing)**.**
Prerequisites: Process p with inplaceable behaviour on input $\{c_i\}$ and output $\{c_o\}$
Annotations: $\text{own}(c_i) := 1$
The transformation of the data channels does not change the semantics of the process network as shown above. The same holds for the recycling channels, because (i) the overall number of initial memory blocks does not change and therefore all the data channels affected still can be filled completely and (ii) the elimination of a receive and a send operation in p can only decrease, not increase, the possibilities for an (unwanted) blocking.

Transformation 5 (Splitting)**.**
Prerequisites: Process p with inplaceable behaviour on input $\{c_i\}$ and output set C_o
Annotation: $\text{own}(c_i) := 1$
In general, we do not assume to have separate split processes, but rather consider the split as an operation happening inside a process of any kind. Therefore, we require inplaceable behaviour, allowing the process to write to the memory block

[1] We do not consider limits of channel capacities as a part of the semantics, since KPN channels are theoretically unbounded. If one needs this feature, one can always add feed-back channels.

before splitting it.

The transformation of the data channels does not change the semantics of the process network as shown above. The same holds for the recycling channels, because (i) the overall number of initial memory blocks for each cycle containing a split branch does not change and therefore all the data channels affected still can be filled completely and (ii) the out-sourcing of multiple receive and a send operation from p to a dedicated process can only decrease, not increase, the possibilities for an (unwanted) blocking.

Transformation 6 (Merging)**.**
Prerequisites: Process p with inplaceable behaviour on input set C_i and output $\{c_o\}$
Annotation: $\forall c_i \in C_i, \text{own}(c_i) := 1$

Transformation 7 (Duplication)**.** Prerequisites:

- Process p with regular behaviour on input set $\{\}$ and output set $C \cup \{c^*\}$
- The channels in C just receive copies of the data going to c^*

Annotation: $\forall c \in C \cup \{c^*\}$, $\text{own}(c) := \omega$, where $\omega = \frac{1}{|C|+1}$ or, if the access tokens sent to c^* already come from an input channel c_{orig}, $\omega := \frac{\text{own}(c_{orig})}{|C|+1}$.

VIII. Applying DMS to the Ultrasound Algorithm

Having theoretically discussed the transformation of a classic KPN to a KPN using DMS in the previous section, we will now show how these transformations are applied to a given KPN. We choose to use the ultrasound image reconstruction algorithm explained in Section III as an example for a multimedia application which has a high demand of computation power and handles large amounts of data.

Starting point is a slight variation from the KPN shown in Fig. 1. To reduce the large number of processes in the application, the index extraction processes and the apodisation multiplication processes have already been merged together to beamforming processes. Due to the moderately complex structure of the network, all channels can be limited to a capacity of one token, where a token usually is a vector or a matrix. The different transformation steps which are performed are going to be explained below.

In a first step, all channels are transformed to DMS channels according to Transformation 1. To all channels drawn in horizontal direction in Fig. 1, Transformation 2 is applied, i.e. a recycling channel is added. The other channels just transport the precalculated working data to the processes that use it. These processes will however keep the data forever, thus rendering recycling channels pointless. Wherever necessary or advantageous, the processes are transformed such that they permit simultaneous access to certain memory blocks (Transformation 3). Again, due to the moderate complexity of the network structure, the additional initial tokens suggested in this rule are not necessary here. Fig. 6 shows the process network after these transformation steps.

In the following, it is described how the optimisation transformations can be applied. This is essentially done traversing the process network from left to right, although other transformation sequences are also possible.

For the `split` process, both splitting and token duplication are an option. We thus postpone the decision here. The element-wise multiplication processes next to it (attenuation compensation) lend themselves to applying an in-place editing transformation (Transformation 4). During this transformation, the channels coming from `split` are annotated with an ownership of one. With this annotation, Transformation 7 (token duplication) can no longer be applied to the `split` process, so this process is transformed according to Transformation 5 (splitting). This transformation requires a new merge process to be put in place which takes all the recycling channels coming from the high-pass filter processes (the first convolution processes from the left), merges them back together and then sends a token linked to the full memory block back to the transducer input process for later use.

The high-pass filter processes have a hybrid functionality: Each one convolves the incoming vector with a high-pass kernel. The convolution is done such that the resulting vector has the same size as the incoming one. Its implementation allows the convolution to be carried out in-place. On the other hand, one copy of the resulting vector is sent to a beamforming process in each beamforming block. We therefore apply Transformation 4 (in-place editing) first for one of the output channels. Then, we apply Transformation 7, access token duplication, for this output channel together with the other output channels. The procedure will be such that the high-pass process obtains an access token from the attenuation compensation process, convolves the data in-place and then duplicates the token, sending one duplicate to each out-going channel. This also necessitates a new process which collects all the duplicate tokens coming back through recycling channels from the different beamforming processes. It will then release these sub-block access tokens except for one, which is sent further to the previously generated merging process.

For the beamforming processes, in-place editing is not an option, since the access tokens they receive are duplicates. This is reflected by the fact that the prerequisites of Transformation 4 are not met, the input channels being annotated with an ownership of a fraction of one.

The following summation processes also fulfil the requirements for in-place editing: From the vectors obtained through the input channels, it is possible to take one out and then add all the others to it. Thus, the operation between the input channel that provides this vector and the output channel is inplaceable, which makes it possible to apply the in-place editing transformation to the summation processes considering one of the input channels and the output channel. The other output channels remain untouched.

For the following processes (envelope detection, low-pass filter and logarithm), in-place editing can again be applied. The `merge` process can be optimised using Transformation 6 (merging). For this, a new splitting process is created, which takes the full blocks recycled from the `display` process and splits them again for reuse at one beamforming process in each beamforming block.

Finally, the processes generating the initial working data are transformed according to Transformation 7 (duplication). All the data generated by them is thus no longer copied but instead only the access tokens are duplicated.

The resulting KPN after applying the clean-up transformations is shown (without the initialisation processes) in Fig. 7. Note that the number of initial access tokens has decreased; this, however, is just due to the merging of smaller vector to bigger matrix tokens. The amount of memory linked to these initial tokens is still the same.

IX. Implementation in DAL

In the previous sections, DMS has been theoretically specified and it was shown abstractly how a given KPN can be transformed to a KPN using DMS. It has, however, not been explained yet how DMS can be implemented on a target architecture. In particular, the notion of an access

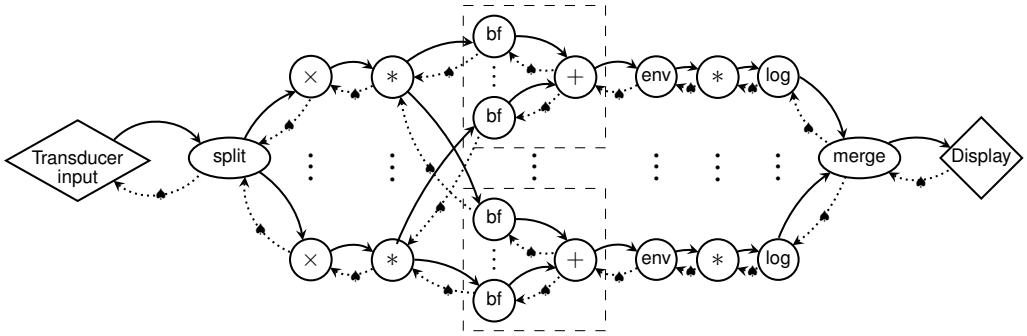


Figure 6: Implementation of the ultrasound image reconstruction algorithm after basic DMS transformations. The data precalculation processes have been left out for reasons of clarity.

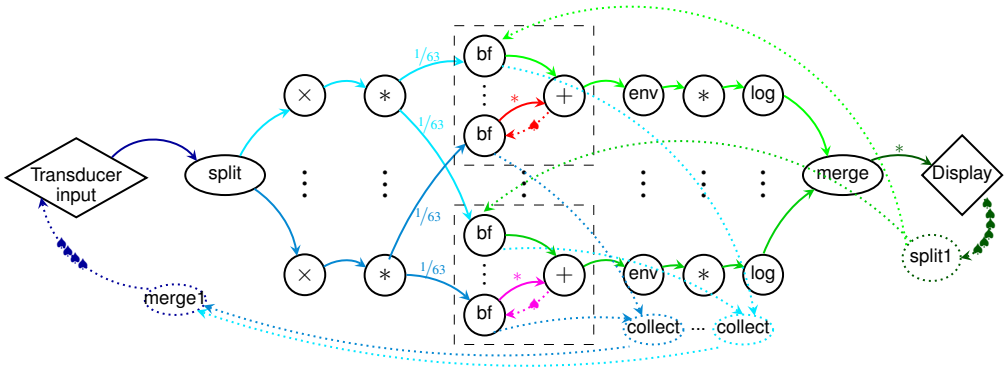


Figure 7: Implementation of the ultrasound image reconstruction algorithm after DMS transformations and optimisations. Colours have been used to mark the individual token cycles. Splits are illustrated by using lighter colours of the same hue. Next to each data channel, its ownership annotation is shown if it is not equal to one.

token was only introduced as an abstract concept. This section will show how the ultrasound image reconstruction algorithm was implemented as a C-based program using the Distributed Application Layer (DAL) framework [3].

DAL is a programming framework which allows the user to specify a KPN and then translates this definition to parallel C code. The specification of a KPN application in DAL consists of two parts. In the first part, one specifies as C code the behaviour of a set of processes with input and output ports. Sending and receiving data works through these ports, using special `read` and `write` functions. The second part is an XML specification of how many copies of these processes exist and how they are connected through channels. There are different back-ends producing native C code for different target platform types; in our case we use a back-end creating POSIX threads since it provides a shared memory model.

The implementation of the DMS mechanisms can contain arbitrarily many safety checks. For instance, one could add a thread-safe reference counting mechanism to each memory block for keeping track of its usage and deallocating it when necessary. One could also store privilege information in the access token to inform a process whether it is allowed to write to the memory block linked to it. Another possibility would be to use the reference counter to dynamically check if a write access is allowed and, if not, block until this is the case.

Our approach concentrates rather on performance than on run-time assertions. As it has been shown in the previous sections, one can formally ensure that the code one creates is correct by construction. If the programmer strictly follows the rules and mechanisms of DMS, no global run-time checks are necessary.

We therefore implement access tokens as simple pointers. We use `malloc` and `free` for allocating and deallocating memory blocks and pointer arithmetic for splitting and merging. The sending and receiving of tokens is done by using the DAL `read` and `write` functions on the pointer itself, sending it over the channel as one would send a normal integer.

DAL allows to specify an initialisation and a clean-up function for each process. We use the former for creating the initial tokens on the channels and the latter for deallocating the memory. As the clean-up function is only called once the whole process network has stopped executing, every process can just store a reference to the memory blocks it allocated during initialisation an then deallocate those during clean-up.

X. Experimental results

The previous sections have shown many optimisation possibilities that are provided by DMS. Now we examine if these theoretical advantages translate to actual performance improvements with the ultrasound image reconstruction algorithm.

To this end, we execute the algorithm on an Intel Xeon Phi 5110P accelerator running a Linux kernel (version 2.6.38.8). This accelerator has 60 processor cores, each running at a clock frequency of 1053 MHz. Each core has four instruction decoding pipelines, which allows for a better utilisation of the ALU and for an overhead-free context switch between four threads per core. The cores are linked by a token ring communication infrastructure to each other and to the memory, which has a total capacity of 8 GB. They can not directly communicate; data exchange is done exclusively through memory. However, each processor has a 32 KB L1 data cache and a 512 KB L2 cache. A complex hardware-implemented cache synchronisation mechanism allows data exchange directly through the caches without accessing the memory. The code is compiled using the Intel compiler ICC, version 14.0.1 with optimisation level 2.

Two implementations of the ultrasound algorithm are tested. One is the configuration discussed earlier in this paper. The second implementation is obtained by aggressively merging processes in the original KPN before translating it to DMS. In particular, all the beamforming and apodisation processes for one output image column (all the processes in a dashed rectangle in Fig. 1) are merged to one single process. The transducer samples are obtained from pre-recorded data loaded into memory during program initialisation. The configuration is for 63 transducers and 2048 12 bit samples (stored as 16 bit integers) per transducer. This gives a process count of roughly 4000 for the first implementation and 200 for the second implementation.

Threads	Mapping	Method	Init	Cap	Mem (MB)	Rate (s^{-1})
4000	dynamic	classic		12	397	65.3
		classic		5	212	61.1
		DMS	4	1	47	121.5
200	dynamic	classic		12	254	147.6
		windowed		2	109	157.7
		DMS	30	7	32	187.3
		DMS	3	2	5	180.0
200	static	classic		50	805	154.3
		classic		3	124	151.0
		windowed		2	109	161.7
		DMS	6	14	8	192.1
		DMS	4	2	6	191.8

Table I: Experimental results for the ultrasound image reconstruction algorithm. Init *denotes the number of initial tokens (i.e. memory blocks) on recycling channels.* Cap *denotes the capacity (in tokens) of the channels (except for the initialisation channels, which only hold one token).* Mem *denotes the total amount of memory used for all channels and the initial tokens.* Rate *denotes the amortised average reconstruction framerate achieved.*

The 4000 thread implementation is tested using dynamic mapping (operating system decides on binding and scheduling of the processes at runtime) with two configurations, namely classic KPN channels and DMS channels.

The 200 thread implementation is also tested using static mapping (binding of processes is decided at design-time according to load-balancing considerations, scheduling is done by hardware, since there are more instruction pipelines than threads). Furthermore, windowed FIFO channels are tested as a third option for channel implementations.

The performance of the different configurations is measured in the form of the amortised average image reconstruction framerate. To this end, the execution time of the program is measured for 50 frames and for 250 frames, 30 times each. The values obtained are then fitted using the least squares method on a linear model (time vs. number of frames).

In general, the channel capacities have a considerable influence on the performance of an application. While too low capacities restrict the parallelism and the scheduling options for a KPN, too high capacities will result in higher memory footprints and thus a worse performance of the caches. We have therefore tested different values for the capacity of all the channels in the range of 1 to 250 tokens per channel. For the DMS implementation, we have also varied number of initial access tokens in all the token cycles in the range of 1 to 250. In Table I, we give two configurations for each implementation: The configuration achieving the best framerate and, if it exists, the configuration coming next to this framerate with a smaller memory footprint. The memory footprint is also given for these configurations. Note that in the case of classic or windowed FIFOs, it depends mainly on the channel capacities whereas for DMS implementations, it is the number of initial tokens that counts.

The numbers show that: (i) the framerate with DMS is significantly higher than with windowed FIFOs and classic channels, (ii) the memory footprint with DMS is drastically lower than with windowed FIFOs and classic channels and (iii) using DMS, it is possible to achieve good performance already with small amounts of memory.

Especially from the fact that no special optimisations for the target platform were applied, it can be concluded that the effort of transforming a KPN to use DMS pays off in terms of performance and memory footprint.

XI. Conclusion

In this paper, we have presented deterministic memory sharing, a concept for sharing memory blocks between different processes in a KPN. We have shown how the concept of access tokens ensures that the determinacy of KPNs still persists even when multiple processes access the same memory regions. Rules have been set up which allow to transform a traditional KPN application such as to make use of DMS. A first set of rules transforms the channels into DMS channels. A second set of rules optimise individual processes. They can be applied locally, i.e. to one process and the channels connected to it without having to consider the rest of the process network.

An ultrasound image reconstruction algorithm was explained and a classic KPN implementation of it was presented. This KPN was then transformed according to the rules mentioned above. Experiments on the Intel Xeon Phi accelerator show that even without any special adaptations, a significant speed-up can be achieved while immensely reducing the memory footprint of the application.

It is important to note that the Intel Xeon Phi is already optimised to ensure good performance also with suboptimal memory configurations. At the same time, it is has a rather complex hardware architecture, which makes it difficult to optimise programs on it for optimal memory usage. We believe that on less sophisticated, simpler and more transparent platforms even higher performance gains can be achieved. The topic how a certain hardware configuration should influence the KPN transformation and optimisation steps described in this paper will be part of our future research as well as the question how these optimisations can be automated.

Acknowledgement

This work was supported in part by the UltrasoundToGo RTD project (no. 20NA21 145911), evaluated by the Swiss NSF and funded by Nano-Tera.ch with Swiss Confederation financing.

References

[1] G. Kahn, "The semantics of a simple language for parallel programming," in *IFIP 74*. North Holland, 1974.

[2] K. Huang *et al.*, "Windowed FIFOs for FPGA-based multiprocessor systems," in *Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*. IEEE, 2007, pp. 36–41.

[3] L. Schor *et al.*, "Scenario-Based Design Flow for Mapping Streaming Applications onto On-Chip Many-Core Systems," in *Proc. CASES*, 2012, pp. 71–80.

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

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

[6] A. Stulova *et al.*, "Throughput driven transformations of Synchronous Data Flows for mapping to heterogeneous MPSoCs," in *Embedded Computer Systems (SAMOS)*, July 2012, pp. 144–151.

[7] E. Lee and D. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[8] G. Bilsen *et al.*, "Cyclo-static dataflow," *Signal Processing, IEEE Transactions on*, vol. 44, no. 2, pp. 397 –408, feb 1996.

[9] H. Oh and S. Ha, "Fractional rate dataflow model for efficient code synthesis," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 37, no. 1, pp. 41–51, 2004.

[10] S. Verdoolaege *et al.*, "Pn: A Tool for Improved Derivation of Process Networks," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, Jan. 2007.

[11] Y. Cho *et al.*, "Buffer size reduction through control-flow decomposition," in *Embedded and Real-Time Computing Systems and Applications, 2007.* IEEE, 2007, pp. 183–190.

[12] E. Cheung *et al.*, "Automatic buffer sizing for rate-constrained KPN applications on Multiprocessor System-on-Chip," in *Proceedings of the 2007 IEEE International High Level Design Validation and Test Workshop*, 2007, pp. 37–44.

[13] H. Oh and S. Ha, "Data memory minimization by sharing large size buffers," in *Proceedings of the 2000 Asia and South Pacific Design Automation Conference*, 2000, pp. 491–496.

[14] K. G. W. Goossens, "A protocol and memory manager for on-chip communication," in *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, vol. 2, May 2001, pp. 225–228.

[15] C. Park *et al.*, "Extended synchronous dataflow for efficient DSP system prototyping," *Design Automation for Embedded Systems*, vol. 6, no. 3, pp. 295–322, 2002.

[16] S. Kiran *et al.*, "A complexity effective communication model for behavioral modeling of signal processing applications," in *Proceedings of the 40th annual Design Automation Conference*, 2003, pp. 412–415.