Communicating Process Architectures – 2001
Alan Chalmers, Majid Mirmehdi and Henk Muller (Eds.)
IOS Press, 2001

Efficient Execution of Process Networks

Twan Basten[1] and Jan Hoogerbrugge[2]

[1] *Eindhoven University of Technology, Dept. of Elec. Eng., PO Box 513, NL-5600 MB, Netherlands and Philips Research Laboratories, Prof. Holstlaan 4, NL-5656 AA, Netherlands*
a.a.basten@tue.nl
[2] *Philips Research Laboratories, Prof. Holstlaan 4, NL-5656 AA, Netherlands*
jan.hoogerbrugge@philips.com

Abstract. Kahn process networks (KPNs) [1] are a popular modeling technique for media- and signal-processing applications. A KPN makes parallelism and communication in an application explicit; thus, KPNs are a modeling paradigm that is very suitable for multi-processor architectures. We present techniques for the efficient execution of KPNs, taking into account both execution time and memory usage.

Keywords: Kahn process networks, dynamic scheduling, deadlock resolution, multi-processor architectures, media processing, signal processing

1 Introduction

Many media- and signal-processing applications in consumer electronics and telecommunications are naturally represented as a collection of cooperating sequential processes that communicate via streams of data. Typical examples of such applications are MPEG2 encoding and decoding. In his seminal paper [1], Gilles Kahn investigates the semantics of collections of cooperating processes. The model of computation described in [1] is nowadays known as the Kahn-process-network model. In [2], Kahn and MacQueen describe a language for specifying process networks and an implementation of that language. To date, (variants of) Kahn process networks (KPNs) are commonly used for developing media- and signal-processing applications. A strong aspect of KPNs is that they make (task-level) parallelism and communication in an application explicit, which means that they are very suitable for execution on multi-processor architectures. Furthermore, KPNs have a straightforward, unambiguous semantics and they are fully compositional. Thus, KPNs facilitate the reuse of application models, which is becoming increasingly important for reducing the design time of new products and services.

The main focus of this paper is on multi-processor architectures (although the presented techniques also work for single-processor architectures). We consider both (heterogeneous) multi-processor architectures with shared memory and (homogeneous) architectures without shared memory such as the one described in [3]. An important question is how to execute KPNs efficiently on some given multi-processor system, where efficiency is measured in terms of both memory usage and execution time. The two main problems that need to be solved are how to find an efficient schedule, i.e., an execution order for the processes and a mapping to processors over time, and how to efficiently allocate memory to store the data streams that the processes use to communicate. The two problems cannot be solved in isolation. The amount of memory reserved for communication influences the possible execution orders of the processes in a KPN. It is also important to realize that the problem is often not to execute a KPN as fast as possible using as little memory as possible; usually, the problem is to execute the KPN within the timing restrictions imposed by the application and within the

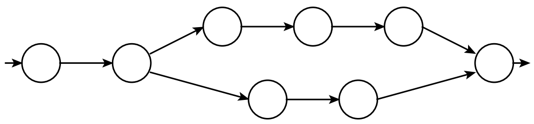

Figure 1: An example of a KPN

memory that is available in the system. It is often the case that the more memory is allocated, the better is the execution time. Ideally, a system designer must be able to make a design-time trade-off between execution time and memory usage, preferably using no more memory than required for obtaining a certain execution time. This paper presents techniques for the efficient execution of KPNs on multi-processor architectures that allow such a trade-off.

In Section 2, we introduce KPNs and their implementation. Section 3 discusses the problem of scheduling KPNs and it gives an overview of existing scheduling approaches. It turns out that one of these approaches is the most promising given our requirements. In Section 4, we discuss this scheduling approach in more detail. It has two drawbacks: It may introduce artificial deadlocks in the execution of a KPN and it may use more memory than necessary. Artificial deadlocks occur when not enough memory is allocated to store data streams between processes; if too much memory is allocated to store these data streams, the application uses more memory than strictly necessary. Thus, these two problems need to be solved before the scheduling approach can be used as the basis for the efficient execution of KPNs. Existing solutions do not meet our efficiency criteria. Section 5 presents our solution to cope with the problem of artificial deadlocks. Section 6 presents a solution to allocate memory to store data streams. In both solutions, memory is, as much as possible, allocated only if necessary. Note that our solutions are not optimal in the sense that they guarantee that the minimal amount of memory required is allocated; it is known that this problem is in general undecidable. However, the experiments in Section 7 show that our solutions are better than existing techniques. They also show that the techniques presented in this paper can be used for a design-time trade-off between execution time and memory usage, as explained above. In Section 8, we briefly explain how our techniques scale to massively parallel architectures such as the homogeneous multi-processor architecture described in [3]. Finally, in Section 9, we make some concluding remarks and discuss future work.

2 KPNs and their implementation

A *process* is essentially a deterministic mapping from some input data streams to some output data streams. For simplicity, we consider only processes that have at least one input and at least one output stream. In practical implementations, processes are often implemented in a high-level programming language such as C++ ([4]) or Java ([5]). A typical process consists of a, possibly non-terminating, iteration; in each cycle of the iteration, it consumes some data from its input streams, it manipulates the data received in this way, and it produces some data on its output streams.

Processes are connected into a network via *theoretically unbounded queues*, called *fifos* because data is consumed from these queues in a first-in first-out order. These fifos provide storage for buffering the data streams between processes. Processes have no other means to communicate with each other than these fifos. Inputs and outputs of processes in a KPN that are not connected to a fifo form the inputs of the KPN from the environment and its outputs to the environment. Again, we consider only KPNs that have at least one input and one output; a KPN can only communicate with its environment via its inputs and outputs.

Figure 1 depicts a typical KPN. Processes are denoted by circles, fifos by arcs between processes, and inputs from and outputs to the environment by unconnected arcs.

An execution of a KPN consists of the execution of the processes it is built from, assuming that a process blocks if it tries to read data from an empty fifo or an empty input from the environment; a blocked process may resume execution as soon as new data becomes available. In [1], Kahn shows that the order of execution of the processes in a network is irrelevant in the sense that, given some input data streams, any order yields the same output streams. As a consequence, a KPN is not essentially different from a process and can thus be used as a process in a larger KPN. In other words, KPNs are compositional, which is a very desirable property although it does not play a role in the remainder of this paper.

Following [6], we distinguish *strictly bounded*, *bounded*, and *unbounded* KPNs. A KPN is strictly bounded if and only if *any* execution of the network requires bounded memory; it is bounded if and only if there is *some* execution that requires bounded memory; it is unbounded if and only if any execution requires unbounded memory. Unfortunately, the question whether a KPN is (strictly) bounded is undecidable [7].

A particularly interesting class of KPNs are the bounded KPNs whose execution continues indefinitely when they are provided with sufficient input. Many practical media- and signal-processing applications fall into this class. The challenge is to execute such a KPN using only bounded memory, preferably not more than necessary to meet the given performance requirements.

3 Scheduling KPNs

Assume that some application designed as a KPN must be implemented on some given multiprocessor system. The problem of scheduling a KPN on such a system is to determine which process executes at which processor at each point in time. A *schedule* is a mapping from processes to processors over time. Two different approaches to scheduling can be distinguished, namely *static* and *dynamic* scheduling. Static scheduling means that a schedule is determined at compile-time, whereas dynamic scheduling means that a schedule is determined at run-time. It is known that it is impossible to determine efficient static schedules for KPNs in general [7]. One could decide to solve this problem by restricting the class of KPNs to a class that allows static scheduling such as synchronous data flow networks [8]. However, we do not want to limit the modeling freedom of application designers. Thus, we do not restrict the class of KPNs and, consequently, the emphasis in this paper is on dynamic scheduling.

The basis of a dynamic scheduling algorithm is a set of *ready* processes. At any point in time, the scheduler chooses as many of the ready processes for execution as there are free processors available. Two different general approaches can be distinguished for determining the set of ready processes at any given point in time. In *demand-driven* scheduling, the driving force behind the scheduling algorithm is the demand for data. The key idea is that only data that is really needed is produced. Demand originates at the outputs of the KPN. That is, initially, the ready processes are those processes responsible for producing output. If a running process requires data from an empty input fifo, it blocks. Thus, new need for data arises and demand propagates through the KPN. The producer of the desired data becomes a ready process and the processor that was executing the blocked process makes a context switch to start executing another ready process. If a process produces requested data, it stops being executed and it is no longer a ready process. In *data-driven* scheduling, the driving force behind the scheduling algorithm is the availability of data. Initially, data is (or should be) available at the inputs of the KPN. Thus, the processes consuming these data are the ready processes. Running processes may produce data, meaning that other processes may become ready. A process stops being ready when it has no longer input data available.

Both dynamic-scheduling approaches have important disadvantages. Data-driven scheduling easily leads to unbounded execution without ever producing output. Input processes start processing input data and continue doing so as long as input data is available; since fifos are theoretically unbounded, storage for produced data is never a problem. Despite the fact that other processes may become ready because of the produced data, a naive execution will make no progress towards output and eventually run out of memory. Demand-driven scheduling often leads to many expensive context switches from one process to another. Since initially all fifos in a KPN are empty, demand needs to propagate from the outputs to the inputs of a network, resulting in a chain reaction of context switches. If input has been read and processed, the produced data propagates from the inputs to the outputs, again resulting in a chain of context switches. This process repeats over and over again, yielding very poor performance. In addition, because of the need to propagate demand, this approach is not very suitable for massively parallel architectures such as the one described in [3]. Demand propagation causes a lot of additional inter-processor communication which is usually relatively expensive.

The answer lies in hybrid approaches. In [2], Kahn and MacQueen propose an extension of demand-driven scheduling with anticipation coefficients. Each fifo f is assigned an anticipation coefficient $ac(f)$. A producer of data requested through f may continue producing an additional $ac(f)$ amount of data in f after it has fulfilled the demand. A disadvantage of this approach is that it is still essentially demand-driven, which means that it is still not optimal for exploiting massive parallelism.

In [9], Jagannathan and Ashcroft propose the Eazyflow model of computation in the context of the functional data-flow language Lucid. In Eazyflow, parts of a Lucid program are classified as data-driven and parts are classified as demand-driven. The disadvantage of this approach is that the classification of program parts and the resulting execution engine are nontrivial.

A final approach, is to bound the size of fifos combined with an essentially data-driven execution, as is done in [4, 6]. The bounds on fifos guarantee that only a restricted amount of data is produced for which there is not yet a clear demand (which means that the approach is also demand-driven in some sense). A running process that attempts to produce in a full fifo blocks and is removed from the set of ready processes so that the processor is freed for other processes. Advantages of this approach are that it is efficient and very easily implementable, that it is well-suited for exploiting parallelism, and that it fits well with the typical media- and signal-processing applications. An important disadvantage is that it is difficult to determine the appropriate fifo sizes. If bounds are chosen too large, the application uses more memory than necessary. If bounds are chosen too small, the performance might drop because of the many context switches. In addition, one might introduce artificial deadlocks in an application. The latter occurs when all processes in a process network block on full or empty fifos and at least one process blocks on a full fifo. The deadlock is artificial because such a deadlock cannot occur in a KPN with unbounded fifos. Nevertheless, because of its advantages, we decided to further investigate the approach with bounded fifos and to try to overcome its disadvantages. The next section goes into more detail.

4 Process networks with bounded fifos

In the remainder, a *process network* (PN) refers to a collection of processes connected into a network via *bounded* fifos. The term *Kahn* process network (KPN) is reserved exclusively for a network of processes connected via unbounded fifos. The execution of a PN is said to *deadlock* if all processes block on full or empty fifos. A deadlock is *artificial* if at least one process blocks on a full fifo.

Figure 2 shows a PN in a deadlock. Attached to each fifo is its size. All processes are blocked either in a write action on a full fifo, denoted by a “w”, or in a read action on an empty fifo, denoted by an “r”. The deadlock is a typical example of an artificial deadlock. It

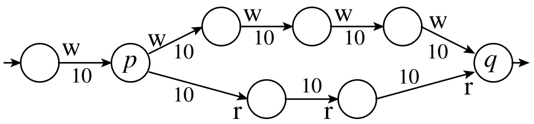

Figure 2: A PN in an artificial deadlock

is caused by the fact that the total amount of storage in the upper branch between processes p and q is insufficient. It can be resolved by increasing the total size of the four fifos in that branch by some appropriate amount.

The idea behind PNs is to use them as an approximation of KPNs. An application designed as a KPN is implemented by choosing an appropriate PN, executing this PN in a purely data-driven way, and resolving artificial deadlocks at run-time, if necessary. Two criteria apply: First, the execution of the PN should be *output complete* in the sense that output that would have been produced by the KPN must also be produced by the PN. Second, if possible, the execution must be *bounded*. In practice, these two criteria cannot always be fully met because KPNs may inherently be unbounded and thus may need unbounded memory in order to produce all possible output. In case of such an unbounded KPN, it will be executed until it runs out of memory, producing as much output as possible. Two problems need to be solved. First, how to choose appropriate bounds on the fifos in a KPN? That is, how to transform the KPN into an appropriate PN? Second, how to resolve artificial deadlocks, if necessary? Note that it is not necessary to resolve an artificial deadlock if the execution of the PN is already output complete.

In [6], Parks also discusses the execution of process networks with bounded fifos. He describes a straightforward strategy for resolving artificial deadlocks. Assume that the size of a fifo is defined in terms of some predefined unit, possibly related to the size of the type (number of bytes) of the data stored in the fifo. Initially, the size of all fifos is set to some predefined or user-defined value. If an artificial deadlock occurs, the *smallest* full fifo is increased by one unit; in this way, it is guaranteed that all bottlenecks in a PN are eventually removed. The question is whether this strategy for determining and executing PNs solves the two problems mentioned above in a satisfactory way. Unfortunately, it does not. It has three major drawbacks that are best illustrated via the examples of Figure 3, which shows some variants of the PN of Figure 2.

Following the above mentioned strategy for resolving artificial deadlocks, in all three examples of Figure 3, the fifo between i and p is enlarged first. However, in none of the cases,

PN$_1$

PN$_2$

PN$_3$

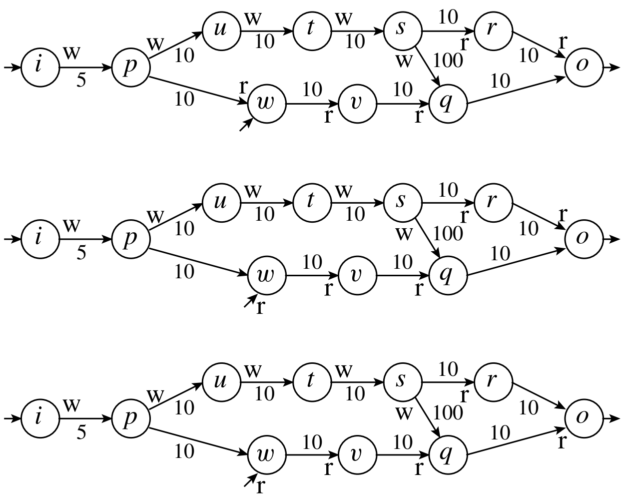

Figure 3: Examples of artificial deadlocks

this is appropriate. In PN_1, the artificial deadlock is, as in Figure 2, caused by insufficient space in the fifos in the upper branch between p and q. The solution is to enlarge any of these four fifos. In PN_2, the cause of the artificial deadlock is different. Process w is blocked on an empty input from the environment. In this case, the bottleneck is the fifo between s and q, despite the fact that it is much larger than any of the other fifos. In PN_3, there is no problem at all. Note that it is identical to PN_2 except that output process o is blocked on an empty fifo between q and o. Provided that the input of w stays empty, the PN is guaranteed to be output complete. It does not make sense to increase the size of any of the full fifos.

The disadvantages of the above mentioned straightforward strategy are now obvious. First, as the first two examples of Figure 3 show, the smallest full fifo in a PN is not necessarily the bottleneck causing an artificial deadlock. Thus, choosing to increase the smallest full fifo might lead to unnecessarily large fifos. In PN_1, the overhead is still relatively small. However, in PN_2, the four fifos connecting i to s are all enlarged to a size of 100 or 101 before the fifo between s and q is enlarged (assuming that sufficient input is available at process i and that the input at w stays empty). Second, the strategy does not take into account whether or not a PN is output complete, which means that in some cases fifos continue to grow even if the PN is already output complete; PN_3 of Figure 3 illustrates this point. Finally, for a user it is often difficult to define meaningful fifo sizes, especially for large applications consisting of hundreds of processes and fifos or if optimal fifo sizes vary as in the examples of Figure 3. The alternative of using a predefined default size has the important drawback that it leads to unnecessarily large fifos if optimal fifo sizes vary.

The next two sections introduce an approach that does solve the two problems of choosing appropriate fifo sizes and resolving artificial deadlocks satisfactorily. We first focus on resolving artificial deadlocks.

5 An efficient scheduling algorithm

The idea behind our scheduling algorithm is to schedule PNs in a purely data-driven way and to resolve artificial deadlocks in such a way that, as much as possible, fifo sizes are increased only if necessary. The examples of Figure 3 serve as our main motivation.

The crucial problem is to find a way to identify the bottlenecks in a PN in case of an artificial deadlock. An important observation is that the cause of a read or write block in a PN is always a unique other blocked process. The reason is that processes always block on precisely one fifo and that fifos are directed point-to-point connections from one process to another one. Consider, for example, PN_1 of Figure 3. Process i is blocked on a full fifo to p. Thus, the cause of i's block is the fact that p is blocked; similarly, p's block is caused by u's block, u's block by t's block, t's block by s's block, and s's block by q's block. Particularly the latter combination is interesting because s's block is a write block whereas q's block is a read block. To resolve the artificial deadlock of PN_1, we have to increase the size of one of the full fifos in the PN; the full fifos are exactly those in the chain from i to q via s. Given the above observation on the causes of the blocks, there is no reason to choose a fifo other than the fifo from s to q. In fact, we may conclude that the bottleneck causing an artificial deadlock is always the first fifo in chain of full fifos, provided of course that there is a first one. In case of a *cycle* of full fifos, there is no clear first full fifo. In such a case, there simply is insufficient space in the cycle and thus the combined fifos in the cycle form the bottleneck. Summarizing, bottlenecks causing artificial deadlocks are always of one of the two forms depicted in Figure 4.

Of course, it may happen that a PN in an artificial deadlock has more than one bottleneck. In addition, the question arises whether situations as depicted in Figure 4 are always bottlenecks that need to be resolved. In fact, this second question has already been answered

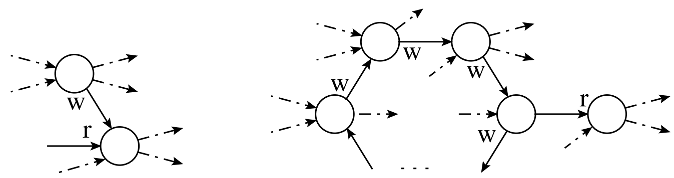

Figure 4: The structure of bottlenecks causing artificial deadlocks

negatively. In PN_3 of Figure 3, the fifo between processes s and q appears to be a bottleneck. However, in the previous section, we have already seen that PN_3 is output complete and, therefore, that fifo s-q is not a bottleneck. It is straightforward to construct a similar example with a cycle of full buffers. Thus, two questions need to be answered: (i) when are situations as depicted in Figure 4 bottlenecks, and (ii) how to choose which bottleneck(s) to solve in case of multiple bottlenecks?

The answer to these questions lies in the criterion that the execution of a PN should be output complete. The starting point for the analysis of an artificial deadlock are therefore the processes producing output for the environment. Since each read or write block has a unique cause, it is possible to track the entire chain of causes leading to the block of an output process. Consider again the three PNs of Figure 3. Figure 5 depicts for each PN the chain of causes leading to the block of output process o.

The answer to the first question posed above is that patterns as in Figure 4 are only bottlenecks if they occur in the chain of causes leading to a blocked output process. Resolving these bottlenecks in the examples of Figures 3 and 5 leads to the correct solutions. In both PN_1 and PN_2, the fifo s-q is the only bottleneck in the chain of causes blocking o, whereas in PN_3 there is no bottleneck leading to the block of o. Thus, in the first two cases, fifo s-q should be enlarged, whereas in the third case no action should be taken because PN_3 is output complete.

The question remains what to do if there are multiple bottlenecks in a causal chain blocking an output. Figure 6 shows an example of such a situation. Both fifos p-q (of size 10) and r-o (of size 5) are bottlenecks. If the PN can be executed in bounded memory, it is guaranteed that the artificial deadlock can be resolved by enlarging one of these two fifos. The choice might not matter. However, maybe it does. Without additional information about

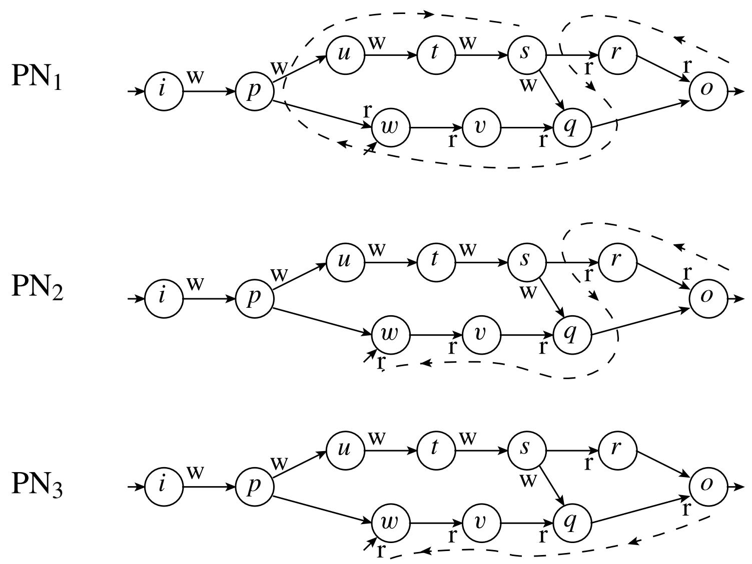

Figure 5: Chains of causes leading to blocks of output processes

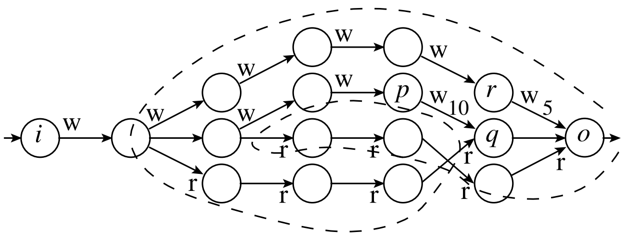

Figure 6: Two bottlenecks in a causal chain leading to a blocked output

the implementation of the processes, it cannot be guaranteed that the correct choice is made. Since we want to keep the deadlock-resolution algorithm as simple as possible and, in particular, independent of the implementation of processes, we decided to choose a strategy that is sometimes suboptimal (with respect to the obtained fifo sizes). In case of multiple bottlenecks in a single chain of causes blocking an output, we choose to enlarge the *smallest* fifo in any of the bottlenecks. In the example of Figure 6, this means that fifo r-o is enlarged. In the worst case, if r-o is not the real bottleneck, this strategy means that the size of r-o becomes 10 or 11 before p-q is enlarged. Thus, as in the approach of [6], fifo sizes might become unnecessarily large. However, our strategy minimizes the chance that this problem occurs.

As a final point, observe that a PN might have multiple output processes. In case of an artificial deadlock in such a PN, it is interesting to analyze for each output process the chain of causes leading to its block. If these chains are disjoint, the bottlenecks in these chains are independent. To ensure maximal progress of the PN, it is a good strategy to enlarge in each of the chains the smallest bottleneck fifo in that chain. Note that several chains might share bottlenecks; in such a case, it of course suffices to enlarge a bottleneck fifo only once.

Summarizing the above observations leads to the following scheduling algorithm for PNs:

1. execute the PN in a data-driven way until a deadlock occurs;
2. if the deadlock is not artificial, then terminate the execution;
3. if the deadlock is artificial, determine all causal chains leading to blocked output processes;
4. determine for each causal chain the bottleneck fifos in that chain;
5. if there are no bottleneck fifos, the execution is output complete and it can be terminated;
6. enlarge the smallest bottleneck fifo in each causal chain blocking an output, making sure that no fifo is enlarged more than once;
7. return to step 1.

From the above discussions, it is clear that artificial deadlocks are resolved if and only if the execution of a PN is not yet output complete. Furthermore, as long as the chains of causes leading to blocked output processes do not have multiple bottlenecks, fifo sizes are only increased if really necessary. In case that a chain has multiple bottlenecks, fifos might grow without necessity but this growth is guaranteed to be bounded if a bounded execution of the PN exists. As a consequence, the two criteria for executing PNs introduced in the previous section, namely that an execution must be both output complete and bounded if possible, are satisfied. As a final remark, note that deadlock resolution can be implemented in the scheduling algorithm in a straightforward way with only little overhead.

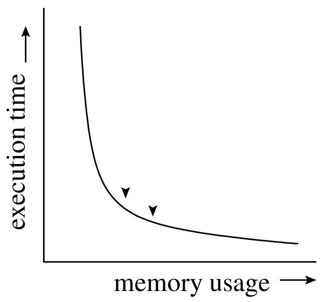

Figure 7: Tuning the sizes of fifos in a PN

6 Choosing appropriate fifo sizes

It remains to solve the problem of obtaining appropriate bounds for the fifos in a KPN, in order to turn it into a PN. Our solution is to tune the size of fifos in a PN at design time via simulation runs. Initially, the size of fifos is set to some predefined small value. In a number of simulation runs, possibly with varying inputs, bottlenecks in the PN are resolved fully automatically and the total amount of data streaming through each fifo is measured. The idea is to set fifo sizes proportional to the amount of data streaming through them and sufficiently large to avoid artificial deadlocks. An interesting observation is that the fifo sizes determined in this way are in some sense *relative* sizes. The experiments might, for example, show that one fifo transports five times as much data as another fifo and that no artificial deadlocks occur if the two fifos sizes are set to 5 and 1, respectively. However, such small sizes may still result in many context switches. Thus, if execution time is an issue and memory is available, then one could choose to set the sizes of the two fifos to 50 and 10, respectively.

In general, one could do a number of experiments to generate a graph as depicted in Figure 7. With very small fifo sizes, the overhead of context switches results in very poor execution time. If sizes are increased – while maintaining relative sizes –, the execution time improves. Usually, the improvement becomes smaller if fifo sizes become larger and, at some point, a further increase in fifo sizes will have no further effect on the execution time. Depending on the execution-time requirements and the available memory, one will probably choose fifo sizes corresponding to points close to the marked points on the graph in Figure 7. Thus, the proposed technique for determining fifo sizes allows the application designer to make a trade-off between execution time and memory usage, as explained in the introduction.

Following the strategy sketched above, 'optimal' fifo sizes are obtained. Of course, the sizes are not truly optimal: First, they are only based on a number of simulation runs; second, it is known that, since deciding boundedness of KPNs is undecidable, the problem of determining optimal fifo sizes for PNs is also undecidable. It can also not be guaranteed that artificial deadlocks can always be avoided. Other inputs than those used in the simulations might lead to new artificial deadlocks. As a consequence, in the final product, artificial deadlocks must still be resolved run-time.

Of course, it is also possible to make a trade-off between execution time and memory usage via a graph as the one depicted in Figure 7 using the straightforward strategy of determining fifo sizes and resolving artificial deadlocks of [6]. However, we believe that our approach leads to a better execution time when a fixed amount of memory is used, because memory is allocated to fifos in a more intelligent way. The effect is illustrated in Figure 8, where the upper graph corresponds to the performance of the straightforward techniques of [6] and the lower graph to the performance of the techniques proposed in this paper. The experiments of Section 7 confirm our belief.

An interesting issue is that the concept of profiling the communication between processes

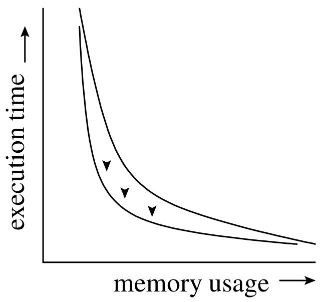

Figure 8: Improving the performance of PN executions

for determining initial fifo sizes complements the new deadlock-resolution strategy. Both techniques allocate memory to data streams only if necessary. The reason that they complement each other very well is that the strategy of resolving deadlocks is independent of fifo sizes. The standard deadlock-resolution strategy of [6] does depend on fifo sizes. The risk that standard deadlock resolution destroys the effect of profiling is high because it always enlarges the smallest full fifo, independent of whether or not this fifo is a bottleneck fifo.

7 Experiments

In this section, we discuss a number of experiments to compare the proposed techniques for the efficient execution of PNs to the techniques proposed in [6]. The experiments were performed using YAPI [4], which is a library of C++ classes implementing PNs. Although PNs have been developed as a programming paradigm for multi-processor architectures, the experiments were performed on a single-processor HP system. The best way to quantify the effects of the execution strategy on the execution time of a PN is to measure the number of context switches that occur during the execution. The reason is that the execution time of a PN is mostly determined by the time needed for data processing and communication, and the time needed for context switches. The former is to a large extent independent of the execution strategy, whereas the latter is in principle overhead that can be reduced by choosing a good execution strategy. The number of context switches that are needed in the execution of a PN is most accurately measured on a single-processor machine, because on such a system the execution strategy is the main factor influencing the number of context switches. In a multi-processor system, the number of context switches in the execution of a PN is influenced not only by the execution strategy but also by other factors such as, for example, bus loads or (relative) processor speeds.

Two applications are used in the experiments, namely MPEG2 decoding and MPEG2 encoding. The experiments quantify the effect of the proposed techniques on the total performance in terms of execution time and memory usage. The aim of the experiments is to draw graphs similar to the graph in Figure 7. As explained above, the number of context switches is used as a measure for the execution time. The total size of all fifos at the end of the execution is used as a measure for the memory usage. For each of the two applications, we compared three execution strategies:

1. Set the initial fifo sizes to some default value and use the straightforward deadlock-resolution algorithm of [6].
2. Set the initial fifo sizes to some default value and use the new deadlock-resolution algorithm proposed in this paper.
3. Determine initial fifo sizes via profiling as proposed in the previous section and use the new deadlock-resolution algorithm.

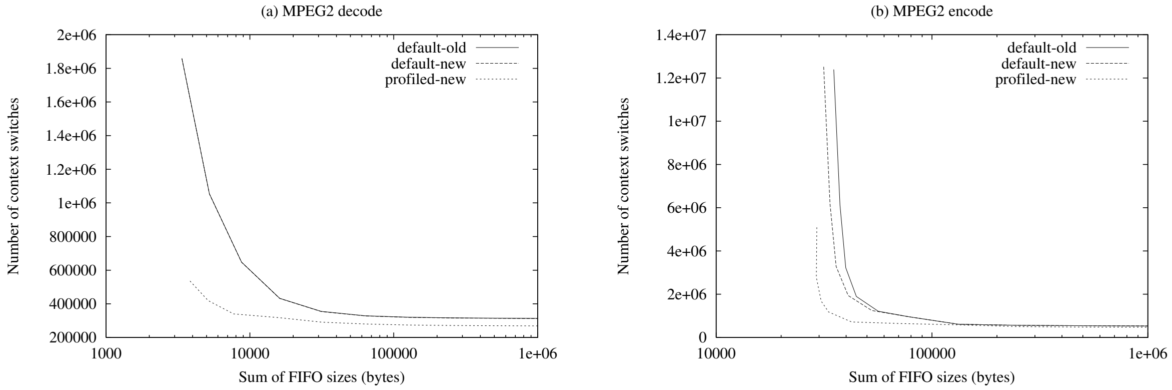

Figure 9: Experimental results: total performance

The first and third strategy correspond to the execution strategies proposed in [6] and in this paper, respectively. The second strategy combines default initial fifo sizes with our deadlock-resolution strategy in an attempt to quantify the effect of intelligent deadlock resolution in isolation.

Figure 9 shows the results of the experiments. The graphs corresponding to the three strategies mentioned above are labeled 'default-old', 'default-new', and 'profiled-new', respectively. For both applications, the strategy proposed in this paper (profiled-new) gives much better results than the strategy proposed in [6] (default-old).

For the MPEG2 decoder, the graphs of the first two strategies (almost) coincide. A closer analysis of the results reveals that in the execution of the MPEG2 decoder with sufficiently large initial fifo sizes no artificial deadlocks occur, whereas only a single artificial deadlock occurs if the default fifo sizes are small. In the latter case, the deadlock-resolution strategy proposed in this paper clearly identifies the bottleneck causing the artificial deadlock and increases the size of only a single fifo, whereas the strategy of [6] increases a few fifos unnecessarily. However, the effects of the different strategies on the total performance are too small to be visible in Figure 9. The observed improvement in performance when going from the first to the third strategy mentioned above is caused completely by the approach to determine initial fifo sizes via profiling. Note that the input data used for profiling the PN model of the MPEG2 decoder and the input data used in the measurements for the 'profiled-new' graph in Figure 9(a) are different.

The results for the MPEG2 encoder show a large improvement when comparing the strategy proposed in this paper with the strategy of [6]. Interestingly, also the strategy combining default initial fifo sizes with intelligent deadlock resolution shows better results when compared to the strategy of [6]. Both the intelligent deadlock-resolution strategy and the approach to determine initial fifo sizes via profiling contribute significantly to the overall performance improvement. The reason is that both techniques allocate memory to fifos in a more intelligent way than the straightforward techniques of [6]. Since deadlock-resolution has a large influence on memory usage, it is interesting to compare the two deadlock-resolution strategies with respect to the amount of extra memory that is allocated during execution to resolve artificial deadlocks, starting with fifos of a default size. Figure 10 shows that the intelligent deadlock-resolution strategy needs less memory than the straightforward strategy. When initial fifo sizes are small, and thus the most artificial deadlocks occur, the amount of memory saved is over 10%.

A final remark is in order. The YAPI library that has been used for the experiments implements an extension to PNs consisting of a construct that allows designers to specify non-deterministic behavior. The MPEG2-encoder model contains one such a construct. The

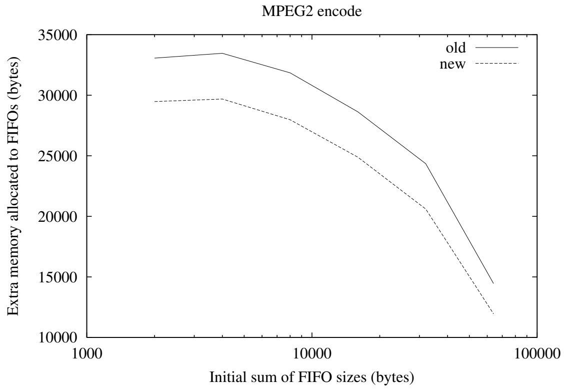

Figure 10: Experimental results: memory allocation

current implementation of our deadlock-resolution algorithm cannot yet efficiently cope with this construct. Our current solution is to switch to the straightforward deadlock-resolution approach of [6] if the construct occurs in the chain of causes leading to a blocked output in an artificial-deadlock situation. In the experiments with the MPEG2-encoder such a switch occurs a number of times. As already mentioned in Section 6, the straightforward deadlock-resolution approach does not work well together with the profiling technique. For similar reasons as those mentioned in Section 6, it does also not work well together with the new deadlock-resolution technique. The fact that the experiments give good results despite these observations strengthens our belief that the proposed techniques are promising. Nevertheless, we consider it an important topic for future work to incorporate non-determinism in PNs (on a conceptual level, in YAPI, and in the execution strategy).

8 Distribution

The number of transistors that can be put on a single chip is increasing rapidly meaning that massively parallel, single-chip multi-processor architectures, such as the one described in [3], are becoming reality. Kahn process networks are an excellent programming paradigm for such architectures because they make parallelism and communication explicit. Given these observations, we want our execution strategy for PNs to be scalable to such massively parallel architectures. The architecture described in [3] has a homogeneous top-level morphology, consisting of identical interconnected *tiles*; each tile is a heterogeneous multi-processor system consisting of a number of general-purposes CPUs, dedicated coprocessors, (on-chip) memory, and interconnect. The low-level heterogeneous specialization guarantees performance, whereas the high-level homogeneous structure of tiles allows for easy scalability. An important aspect of the architecture is that it does not have processing units or shared memory on the level of tiles. If PNs are used as a programming paradigm for such an architecture, the execution of PNs must be distributed over the tiles. The techniques that we propose are very suitable for that purpose. First, the data-driven execution distributes trivially and it allows for the easy exploitation of any parallelism that is allowed by the data. Second, the profiling technique is independent of the underlying architecture. Third, deadlock detection can be implemented efficiently using, for example, the algorithm of [10]. This algorithm is based on the exchange of a number of tokens between the components participating in a distributed computation, i.e., the tiles. The information needed to decide whether or not a deadlock in the execution of a PN is artificial can easily be added to these tokens. If a deadlock turns out to be artificial, the deadlock-resolution algorithm proposed in this paper can be initiated on the tile(s) that is (are) hosting the output process(es) of the PN.

9 Concluding remarks

In this paper, we have described techniques for the efficient execution of Kahn Process Networks. The basis consists of a data-driven execution of the process network while setting bounds on the fifos used for storing data streams between processes. We propose to determine initial fifo sizes by profiling the communication behavior of an application. Since bounding the fifo sizes in a process network might result in artificial deadlocks if initial fifo sizes are chosen too small, we also propose a technique for identifying and allocating extra memory to the bottleneck fifos causing such artificial deadlocks. The profiling- and deadlock-resolution techniques complement each other well because they allocate memory to fifos only if necessary (as much as possible). Our experiments show that the proposed execution strategy has a better performance (execution-time/memory-usage ratio) than the existing solution implemented in for example [4] and [5]. Furthermore, it allows for a design-time trade-off between execution time and memory usage. A final strong aspect is that it scales in a straightforward way to massively parallel architectures.

As future work, we would like to do more experiments to study the execution behavior of process networks. The current execution strategy only increases fifo sizes. However, in certain circumstances, an intelligent strategy for reducing fifo sizes could lead to further performance improvements. It might also be possible to improve the profiling technique by taking into account not only the amount of data streaming through a fifo but also the distribution of data over time. If data streaming through a fifo is distributed evenly over time, it might be possible to reduce the memory allocated to the fifo.

Further improvements in the execution of process networks might be possible by improving the deadlock detection and resolution strategy. A process network often consists of a number of relatively independent parts. Currently, a deadlock in one such a part can only be resolved after the whole network has stopped executing. It would be an improvement if bottlenecks causing artificial deadlocks in a part of the network that is relatively independent of other parts could be identified and removed without having to wait until the entire execution has deadlocked.

Another aspect that deserves further study is the integration of the possibility to specify non-deterministic behavior in process networks. This would make the paradigm more suitable for developing applications combining streaming behavior with reactive behavior. A good starting point can be the work done in the Ptolemy project [5] that focuses on a coupling of a large number of paradigms including process networks and state machines.

Finally, we would like to further investigate the execution of process networks on the homogeneous multi-processor architecture described in [3]. We believe that the combination of process networks and the homogeneous architecture is a very promising one that allows for the fast development of new applications with high-performance requirements.

Acknowledgement We want to thank Paul Stravers and the anonymous referees for their useful suggestions.

References

[1] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J.L. Rosenfeld, editor, *Information Processing 74, Proceedings*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland, Amsterdam, The Netherlands, 1974.

[2] G. Kahn and D.B. MacQueen. Coroutines and Networks of Parallel Processes. In B. Gilchrist, editor, *Information Processing 77, Proceedings*, pages 993–998, Toronto, Canada, August 1977. North-Holland, Amsterdam, The Netherlands, 1977.

[3] P. Stravers and J. Hoogerbrugge. Homogeneous Multiprocessing and the Future of Silicon Design Paradigms. In *International Symposium on VLSI Technology, Systems, and Applications (VLSI-TSA), Proceedings*, Hsinchu, Taiwan, April 2001.

[4] E.A. de Kock et al. YAPI: Application Modeling for Signal Processing Systems. In *37th. Design Automation Conference, Proceedings*, pages 402–405, Los Angeles, CA, June 2000. IEEE, 2000.

[5] E.A. Lee. Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M01/11, University of California, EECS Dept., Berkeley, CA, March 2001.

[6] T.M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, EECS Dept., Berkeley, CA, December 1995. Technical Memorandum UCB/ERL M95/105.

[7] J.T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*. PhD thesis, University of California, EECS Dept., Berkeley, CA, 1993. Technical Memorandum UCB/ERL M93/69.

[8] E.A. Lee and D.G. Messerschmitt. Synchronous Data Flow. *IEEE Proceedings*, 75(9):1235–1245, September 1987.

[9] R. Jagannathan and E.A. Ashcroft. Eazyflow: A Hybrid Model for Parallel Processing. In R.M. Keller, editor, *Parallel Processing, 1984 International Conference, Proceedings*, pages 471–475, Livermore, CA, August 1984. IEEE Computer Society Press, Silver Spring, MD, 1984.

[10] E.W. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren. Derivation of a Termination Detection Algorithm for Distributed Computations. *Information Processing Letters*, 16:217–219, 1983.