Mapping Array Communication onto FIFO Communication - Towards an Implementation

Jeffrey Kang Albert van der Werf Paul Lippens
Philips Research Laboratories
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
Jeffrey.Kang@philips.com Albert.van.der.Werf@philips.com Paul.Lippens@philips.com

Abstract

In high-throughput real-time media processing systems, the communication between processing units is typically specified as multi-dimensional arrays. However, the implementation of such applications is mostly FIFO-based. Mapping array communication onto a FIFO-based implementation requires complex address generators if the arrays have multiple dimensions. In this paper, we present a method for mapping array communication onto an efficient microcomputer architecture implementation based on FIFO communication via shared memory. A good hardware/software partitioning for the address generation is proposed. Furthermore, a complete design flow from specification to implementation is described. We illustrate this method with a design case: the communication of video frames between the frontend and the compressor in an MPEG encoder.

1. Introduction

Contemporary signal processing ICs are becoming more and more complex; the ever decreasing feature size and increasing level of integration allow more functionality to be integrated on one chip. Such a complex system consists of many functional blocks (e.g. encoders and decoders), which operate independently and in parallel and yet synchronize with each other by data communication. In media processing systems, such communication is mostly stream-oriented, where an *array* (for example of video pixels) is filled by a *producer*, which is then emptied by a *consumer* (a stream model can be found in [10]). This can be viewed as writing to and reading from a *shared memory*. For the remainder of this paper we will assume that communication between producer and consumer goes via shared memory.

An example of array-based communication can be found in the PHIDEO technology [5] [8]. PHIDEO is a high-level hardware synthesis design methodology, and supports the complete design flow starting from a high-level specification all the way down to layout[1]. It is mainly targeted towards high-throughput applications such as video processing. PHIDEO's model of computation is based on exchanging *multi-dimensional arrays* between processing units; the implementation is based on a single controller which synchronizes all communication. The motivation for introducing multi-dimensional arrays is that in most Digital Signal Processing (DSP) applications, (nested) loops play an important role [5]. They originate from the repetitive nature of DSP applications, and the way they are specified. For example, in a given video application a certain operation may be repeated for all pixels in a macroblock, and then for all macroblocks in a frame. In this case, there is a three-dimensional loop which iterates over frames, within frames over macroblocks, and within macroblocks over pixels. Especially for high-throughput applications, the loop hierarchy is important since parallel execution of loops is required for optimal implementations.

In media processing architectures, *FIFO-based* communication is preferred as implementation model. These FIFOs buffer data from the producers while the consumers are emptying them. In this way the processing of the producer and consumer can be detached and they can operate in parallel. The communication between them is thus asynchronous. Furthermore, in order to explicitly expose parallelism, such applications are often modeled as a network of parallel processes [7], which communicate via FIFO channels.

Mapping array-based communication onto FIFO-based communication requires complex address generators on both the producer and consumer sides, especially when the array is multi-dimensional, and hence many iterators are involved in the calculation of the physical memory addresses of the array elements. The problem addressed in this paper is therefore: how to come to an efficient hardware/software

[1] To be exact, PHIDEO supports the design flow down to RTL level, but its output is such that other tools for generating the netlist and layout can easily take over from there

implementation of array-based communication via shared memory using FIFOs.

This paper is structured as follows: in Section 2, we discuss the difference between synchronous array-based communication (as in PHIDEO) and asynchronous FIFO-based communication. Section 3 describes our method to come from a specification of array-based communication to an implementation of FIFO-based communication. Section 4 illustrates this method with a case study. In Section 5, the design flow from a high-level specification to a hardware architecture implementation is described. Section 6 shows some experimental results for different FIFO implementations. Section 7 presents the conclusions.

2. Synchronous versus asynchronous communication

The communication controller model of PHIDEO is shown in Figure 1. The producer and consumer communicate via a shared memory. Each has its own Address Generator (AG) to generate memory addresses. A controller synchronizes the communication between both processes. We can see that the communication is highly synchronous: the timing of the producer and consumer is tightly bound by the controller. This has the disadvantage that they cannot really operate independently. Furthermore, the exact schedule has to be determined at compile-time, which increases the complexity of the compiler. Consequently, the PHIDEO model can best be applied for high-throughput processing *within* a task. To achieve true task-level parallelism, a different kind of model is needed.

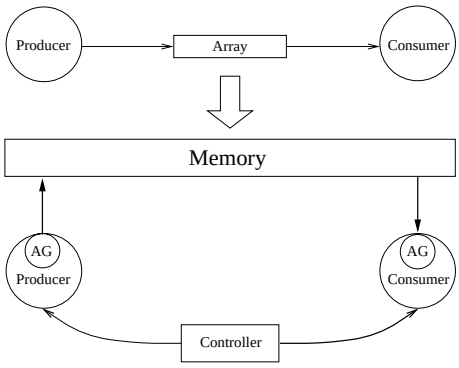

Figure 1. PHIDEO controller model for array-based communication.

In a media processing system, the following aspects of real-time video processing applications are important:

- Video applications require fast computation on large amounts of data. Thus a high level of parallelism in processing and communication is essential (task-level parallelism).
- Compared to the data processing rate, the control decisions are taken with a much lower frequency.
- Each processing unit in the processing chain may have its own latency or even its own clock domain. Therefore, the execution schedule cannot be determined statically, because the timing relations between these processing units are not known in advance.

Considering these aspects, a data-driven model is most suitable. The communication is asynchronous and the schedule is dynamically determined based on data/space availability. The Kahn process network model [7] is such a model for asynchronous communication. In this model, the applications are represented as directed graphs, in which the nodes are the processes and the edges represent the communication channels between them. Kahn process networks assume channels of unlimited capacity. This means that write operations to a channel are never blocked due to channel overflow. However, read operations can be blocked because of an empty input channel.

Video applications are often mapped onto Kahn process networks. The data-flow consisting of a number of independent processing tasks is mapped onto the nodes of the directed graph. The communication of data is asynchronous and is based on *packets* of arbitrary size, which are buffered in the edges. These queues maintain the order in which the packets arrive, and can be implemented with FIFO buffers. Of course, in a real system, the channel capacity is limited, thus the Kahn model has to be modified. In order to prevent channel overflow, also the writes have to be blocked if the output channel is full. This modification does not jeopardize the validity of the Kahn model, see [4]. Several tools have been developed to model Kahn process networks, an example can be found in [3]. The strength of FIFOs is that they are a good step towards implementation, however they are too limited for the specification of an application.

3. From specification to implementation

We have observed that many real-time processing applications have specifications given in terms of array communication, while their implementation is based on FIFO communication. This section presents a method for mapping array-based onto FIFO-based communication in an efficient way. We will use the example of MPEG video [6] communication to explain the basic principles.

The communication of a video frame between two processes can be completely specified by means of multi-dimensional arrays. For sake of simplicity, we choose the macroblock (16×16 pixels) to be the smallest unit of data. This leads to the following array expression:

```
mblocks[t][i][f][s][l][m]
```

The iterators and their ranges for a PAL image are shown below:

ITERATORS	RANGE
t time	0..oo (infinity)
i frame (in group)	0..2
f field	0..1
s stripe	0..35
l line	0..7
m macroblock	0..44

The time iterator corresponds with the outer loop which repeats forever. The remaining iterators represent the units of data into which a video stream may be decomposed. A different decomposition may be chosen, resulting in more or fewer iterators, however this is not important for this example. Figure 2 shows an example of indexing a macroblock

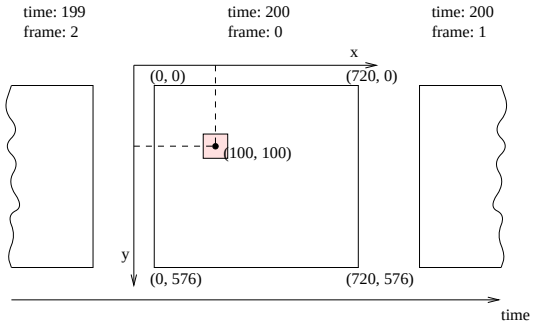

Figure 2. Indexing a particular macroblock.

in a particular frame: a macroblock in the first frame of a group of pictures within time unit 200, within which a pixel with coordinates (100, 100) resides, is expressed as:

```
mblocks[200][0][0][6][2][6]
```

The filling and emptying of the array by the producer and consumer is described as by nested loop as given by the pseudo-code below:

```
Producer:
for t = 0 to oo do
  for i = 0 to 2 do
    for f = 0 to 1 do
      for s = 0 to 35 do
        for l = 0 to 7 do
          for m = 0 to 44 do
            put mblocks[t][i][f][s][l][m];
          end for
        end for
      end for
    end for
  end for
end for
```

```
Consumer:
for t = 0 to oo do
  for i = 0 to 2 do
    for f = 0 to 1 do
      for s = 0 to 35 do
        for m = 0 to 44 do
          for l = 0 to 7 do
            get mblocks[t][i][f][s][l][m];
          end for
        end for
      end for
    end for
  end for
end for
```

Note that the loop nesting structure of the producer may differ from that of the consumer, depending on their specified functionality. For example, the producer may process the video data line- and field-wise, while the consumer does this macroblock- and frame-wise. This is one of the reasons to have multiple dimensions.

Assuming communication via shared memory, we need a scheme to transform the array indices into memory addresses. The address expression for a particular macroblock looks as follows:

```
mb_addr(t,i,f,s,l,m) = (T*t + I*i + F*f +
                        S*s + L*l + M*m) mod K
```

In the above expression, the capital letters represent the offset contribution to the address of each iterator. The factor `K` indicates the size of the allocated memory. This expression specifies the *memory map* of the video frame.

To ensure that data is actually produced prior to being consumed, some kind of *synchronization* functionality between the producer and consumer is needed. We prefer not to use the PHIDEO controller model shown in Figure 1, because as mentioned before, no true task-level parallelism can be achieved in that way. An alternative solution is to use *tokens* for synchronization. They are passed between the producer and consumer via token FIFOs (not to be confused with data FIFOs as referred to in high-level models). This is illustrated in Figure 3. Note that the controller is *distributed* over the tasks instead of centralized. In this way the producer and consumer are not strictly bound and can operatate independently. The status of the FIFO (full/empty) plays an important role for the synchronization.

In the communication model shown in Figure 1, the address generators in the producer and consumer have to compute an address for each macroblock using the above expression. This may make them very complex, especially if the number of array dimensions grows. This makes the design and especially the verification difficult, because due to the large state space of the address generation hardware, long simulations have to be run before all the states have been traversed (this can be up to several frame periods) and

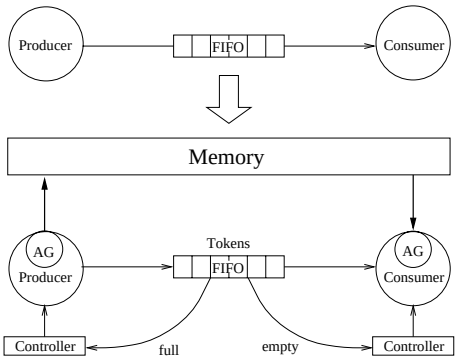

Figure 3. FIFO communication model with token FIFO and distributed controller for synchronization.

verified. In order to reduce the complexity of the address generators, we apply a method called *index splitting*. The addressing of the array is split into a higher level address generation (with higher level iterators) and a lower level address generation. In general, a split can be done at arbitrary level. This choice depends on the grain size at which the synchronization is done.

For instance, if we apply index splitting at stripe level (higher iterators are `f` and `s`, lower iterators are `l` and `m`), then we get the following address expression:

```
mb_addr(t,i,f,s,l,m) = base_addr +
                       L*l + M*m
```

Where:

```
base_addr = (T*t + I*i + F*f + S*s) mod K
```

Looking at the above expression, we can distinguish two different parts:

1. A *base address*, which was obtained after index splitting at stripe level.
2. An *offset* (given by `L*l + M*m`), which calculates addresses using the iterators `l` and `m` to determine the exact locations of the individual macroblocks in memory.

Note that with the above definition of the base address, `K` has to be chosen as a multiple of the stripe size, otherwise the offset may cause the *modulo* operation to wrap around to the beginning of the allocated memory block.

We now make a hardware/software partitioning of the address generation. The base addresses can be calculated *once* in software at startup. These addresses can be attached to the synchronization tokens and passed between the producer and consumer. The offset part of the address expression has to be computed by the address generators within the producer and consumer. It can be seen that the number of iterators used has decreased, hereby reducing the complexity of the address generators.

Figure 4 shows the complete FIFO-based communication model. The address generators in the producer and consumer are smaller than before due to the mentioned optimization. The synchronization of the data communication is done via a token FIFO. The base addresses are passed between the processes along with the tokens, and the offsets are calculated by the address generators to obtain the actual memory locations of the macroblocks. After being consumed, these base addresses can be returned to the producer via a *return FIFO*, so that the memory blocks they refer to can be reused. The interaction between the FIFOs and the controllers are not depicted here.

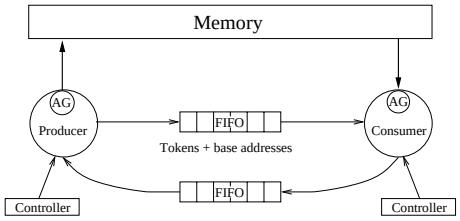

Figure 4. FIFO communication model with recycling of memory blocks.

The approach as has been described in this section has several advantages. They are listed below:

- Figure 4 shows that we have separate data *communication* and *synchronization* channels. Assuming communication via shared memory and busses, the data memory and the token memory can be separated over different busses and memories. This reduces the load on these (often scarce) resources. And because the token memory is small (each token is in our case a 32-bit address word), it can be implemented on-chip for fast access.
- Some processes do not actually perform any processing on the data, but rather just consume them and output them in a different order, e.g. reshuffling of video frames. In a straightforward implementation, this process would copy the input data and buffer them for reordering before sending the data to its consumer. This is extremely inefficient for the bandwidth considering that no processing is done on the data itself. In our solution, only the base addresses are copied and reordered, and their bandwidth demand is relatively low.
- In some applications a process may have multiple consumers (multi-cast). Rather than maintaining multiple copies for all consumers, only the base addresses are copied, again reducing the amount of copying. Since

there are also multiple return FIFOs for each base address, a mechanism is needed to check whether a particular base address has already been consumed, before it may be reused. A solution for this problem is to add a *reference counter* to each token, which is set to the number of consumers. We will not go into further detail here.

- During hardware/software partitioning of the address generation, the base address can be calculated by software and put into the return channel at boot-time. Note that the software has a lot of freedom in this, since this is a memory allocation problem. The address offsets are calculated at run-time by hardware within DMA (Direct Memory Access) blocks. This address generation hardware is less complex because a part has been taken over by software. This makes it easier to design and verify.

In the following sections, we will illustrate the principles explained here with a small case study. We will start from a high-level specification with array communication and derive an implementation with FIFO communication. Furthermore, we will perform hardware/software partitioning and present an architecture onto which this application can be mapped.

4. Case study: MPEG encoder's frontend-compressor communication

We have applied the method described in the previous section to a case taken from an MPEG-2 video encoder [1] [11]. The signal path of MPEG encoding can be roughly partitioned into a frontend, a compressor and a backend as shown in Figure 5. Each of these parts is a process which communicates with other processes via buffers. The input and output buffers handle the communication between the frontend, compressor and backend, and an additional buffer is used to store reference frames which are needed for compression.

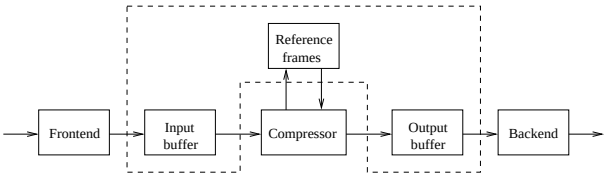

Figure 5. The signal path for MPEG video encoding partitioned into processes communicating via buffers.

In our case study, we concentrated on the communication between the frontend and the compressor. The input buffer also performs some *reshuffling* of the incoming frames (this is part of the MPEG standard for IPB GOPs [6]); in this sense it is not strictly a FIFO.

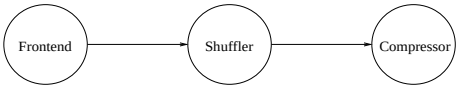

Figure 6. Frontend-compressor communication modeled as a Kahn process network.

When modeling this communication as a Kahn process network, we can distinguish three different processes: frontend, shuffler, and compressor, and they communicate via FIFO channels (see Figure 6). The shuffling of the frames by the shuffler can be specified as:

```
for t = 1 to oo do
  get mblocks[t][0];
  get mblocks[t][1]; put mblocks[t][0];
  get mblocks[t][2]; put mblocks[t-1][1];
                     put mblocks[t-1][2];
end for
```

The lower iterators have been omitted here to keep the code clear. It can be seen that the shuffling is specified by simply manipulating the iterator `t`. This is one of the advantages of multi-dimensional arrays.

We have decided to perform the index splitting at stripe level. It has been chosen because at lower levels, the synchronization overhead becomes high because synchronization tokens have to be sent more often. For larger synchronization grains, the required input buffer size becomes too large as shown in Table 1.

Grain size	Compression	Input buffer size (no. of frames)
Frame	Frame based	4
	Field based	4
Field	Frame based	4
	Field based	3
Stripe	Frame based	2.5 + 3 stripes
	Field based	2 + 3 stripes

Table 1. Required input buffer size for various synchronization grains.

The resulting FIFO-based communication model is shown in Figure 7 (address generators and controllers have been omitted). Figure 7(a) shows the result of the straightforward application of the techniques explained in the previous section. This implementation involves a lot of needless copying, because the shuffler does nothing with the data.

Figure 7(b) shows the optimized solution. The shuffler shuffles the addresses instead of the actual data. This optimization is only possible because we have split synchronization from communication. Initially, a number of base addresses are calculated and stored in the return FIFO. Note that this number must be at least equal to the one indicated in Table 1, otherwise deadlock will occur. The frontend writes the stripes into the memory and sends the base addresses of those stripes to the shuffler. The shuffler reorders the addresses and sends them to the compressor. It only needs the higher level iterators. The compressor reads the stripes starting from the received base addresses and returns these to the frontend for memory recycling.

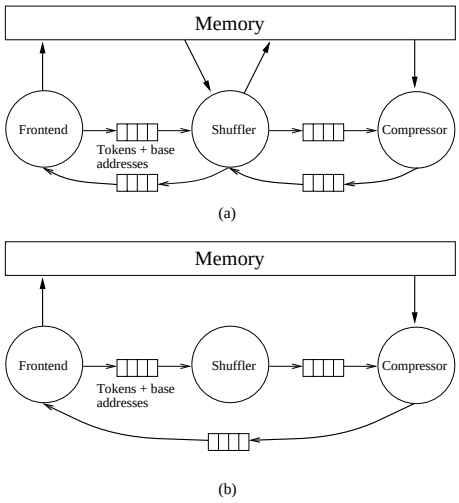

Figure 7. Frontend-compressor communication by passing base addresses: (a) straightforward, (b) optimized.

5. Design flow

This section describes the design flow to map the frontend-compressor communication onto a microcomputer architecture. The used architecture template, as well as a more detailed example of the design flow, can be found in [2].

At the starting point of the design flow, the application was modeled with a C-language description. An API (Application Programmer's Interface) has been provided, which handles the processes' read and write operations on the channels. At this level, the simulation was purely functional, without any link to the architectural implementation.

Next, the processes were mapped onto abstract processors (with this we mean that only the functionality is contained in these processors; no choice has yet been made about mapping onto hardware or software) and the communication was verified. Here we used a cycle-based architecture simulation tool to capture and simulate the hardware architecture. The architecture models used in this simulators are written in ANSI-C. The target hardware architecture is a *heterogeneous multi-processor architecture* consisting of both dedicated hardware components and general purpose processors, which communicate via shared memory and busses. In this step, the intended processing devices are still abstract processors, in the sense that their functionality is still implemented using the C-code of the previous step. The communication infrastructure (e.g. busses and memories) is modeled in a cycle-accurate way. An interface block is used to connect the functional code of the abstract processors to the communication infrastructure at architecture simulator level. It is possible to annotate the functional code with delay statements in order to model the latency of the abstract processors. This level of simulation is used for verifying functional correctness, and to get a first estimate of the real-time feasibility of the particular partitioning into processes.

Further down the design flow, hardware/software partitioning was done. For each task, it was decided either to map it onto dedicated hardware, or to implement it in software running on a general purpose processor (CPU). The challenge here was to find a good balance between hardware and software, in order to get to an optimal architecture in terms of performance, power dissipation, and flexibility. To this end, an instruction-set simulator for a CPU was incorporated in the simulation, on which the software tasks were run. The functional code that should run on the CPU was compiled using the compiler available for that processor. The hardware tasks were kept as abstract processors and all was co-simulated. In our case, the frontend and compressor were mapped onto hardware for performance and power reasons, and the shuffler was mapped onto the CPU to keep flexibility in the application (different shuffling modes can be supported).

The resulting hardware architecture is depicted in Figure 8. The communication of tokens and video data are split over two busses and memories. The CPU used is a MIPS processor, on which the root task (for initialization) and shuffler task are running. The frontend and compressor are connected to both busses. The arbitration on the busses is done by Bus Control Units (BCUs).

When the required performance is met, the hardware models can be designed (in VHDL). During design, the latency figures become more clear, and can be fed back to the annotated delay statements in the abstract processors for a more accurate simulation. The hardware blocks can be designed and replace the abstract processors one by one, and after each step co-simulated with the rest using the C-VHDL interface provided by the simulator.

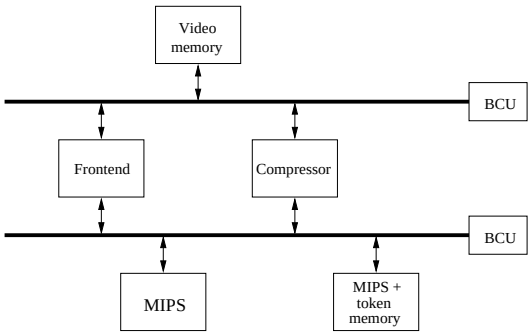

Figure 8. The resulting hardware architecture.

6. Experimental results

We performed several simulations to evaluate the different FIFO implementations described in this paper. The target architecture was the two-bus architecture shown in Figure 8, with a small difference that all blocks were kept as abstract processors. The reason is that we were only interested in comparing the communication overhead of the different options and did not want to take into account the processing delays. Therefore all tasks were assumed to have zero delay. The communication, however, was simulated cycle-accurately. Three FIFO implementations were examined: 1) passing synchronization tokens (Figure 3), 2) passing base address tokens (Figure 7(a)), and 3) optimized passing of address tokens (Figure 7(b)). The simulations were run for a period of four frames ($1\times$ shuffling). The results are shown in Table 2.

	Synchro. tokens	Address tokens	Addr. tokens (optimized)
Total cycles	12,782,942	13,326,405	6,090,746
Token bus cycles	104,591	214,456	176,139
Data bus cycles	12,708,941	13,167,092	4,963,209

Table 2. Simulation results for different FIFO implementations.

7. Conclusion

A method for efficiently mapping multi-dimensional array communication onto an implementation using FIFO communication via shared memory has been presented. Multi-dimensional arrays make it easy to specify high-throughput DSP applications. By manipulating the iterators, we can define an optimal memory map for the application and make use of index splitting in a very flexible way. Index splitting can be used to separate the address generation into a base part that can be determined during startup by software, and an offset part that is calculated by address generation hardware at run-time. In this way, the address generators can be kept simple to design and verify.

A FIFO communication model has been derived in which data communication is separated from synchronization with tokens. By attaching base addresses (obtained after index splitting) to these tokens, we can avoid wasting bandwidth due to unnecessary copy operations by processes that do not perform any data processing. Furthermore, multi-cast can be implemented by copying the tokens, not the data itself.

We have illustrated the proposed method with a case study: the communication between the frontend and the compressor in an MPEG encoder. Except for the physical hardware implementation, the complete design flow has been described from top-level specification and simulation down to a hardware architecture implementation.

References

[1] A. van der Werf et al. I.McIC: A single-chip MPEG2 video encoder for storage. *IEEE Journal of Solid-State Circuits*, 32(11):1817–1823, November 1997.

[2] A.K. Nieuwland and P.E.R. Lippens. A heterogeneous HW-SW architecture for hand-held multi-media terminals. *1998 IEEE Workshop on Signal Processing Systems*, pages 113–122, SiPS '98.

[3] E.A. de Kock et al. YAPI: Application modeling for signal processing systems. *Proceedings of the 37th Design Automation Conference*, pages 402–405, June 2000.

[4] J.A.J. Leijten et al. Stream communication between real-time tasks in a high-performance multi-processor. *Proceedings of DATE '98*, pages 125–131, February 1998.

[5] J.L. van Meerbergen et al. PHIDEO: high-level synthesis for high-throughput applications. *Journal of VLSI Signal Processing*, 9:89–104, 1995.

[6] Joan L. Mitchell, Didier Le Gall and Chad Fogg. *MPEG Video Compression Standard*. Chapman & Hall, 1996.

[7] G. Kahn. The semantics of a simple language for parallel programming. *Proceedings of the IFIP Congress 74*, 1974.

[8] P.E.R. Lippens et al. Allocation of multiport memories for hierarchical data streams. *Proceedings of the International Conference on Computer-Aided Design*, pages 728–735, 1993.

[9] W.F.J. Verhaegh et al. Modelling periodicity by PHIDEO streams. *Proceedings of the Sixth International Workshop on High Level Synthesis*, pages 256–266, November 1992.

[10] W.F.J. Verhaegh et al. Multidimensional periodic scheduling: model and complexity. *Proceedings of the Euro-Par*, II:226–235, 1996.

[11] W.H.A. Brüls et al. A low-cost audio/video single-chip MPEG2 encoder for consumer video storage applications. *Proceedings of the International Conference on Consumer Electronics (ICCE)*, pages 314–315, June 2000.