An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures

Bart Kienhuis[1,2], Ed Deprettere[1], Kees Vissers[2], Pieter van der Wolf[2]
[1] Delft University of Technology, Mekelweg 4, 2628 CD, Delft, The Netherlands
[2] Philips Research Laboratories, Prof. Holstlaan 4, 5656 AA, Eindhoven, The Netherlands

Abstract

In this paper we present an approach for quantitative analysis of application-specific dataflow architectures. The approach allows the designer to rate design alternatives in a quantitative way and therefore supports him in the design process to find better performing architectures. The context of our work is Video Signal Processing algorithms which are mapped onto weakly-programmable, coarse-grain dataflow architectures. The algorithms are represented as Kahn graphs with the functionality of the nodes being coarse-grain functions. We have implemented an architecture simulation environment that permits the definition of dataflow architectures as a composition of architecture elements, such as functional units, buffer elements and communication structures. The abstract, clock-cycle accurate simulator has been built using a multi-threading package and employs object oriented principles. This results in a configurable and efficient simulator. Algorithms can subsequently be executed on the architecture model producing quantitative information for selected performance metrics. Results are presented for the simulation of a realistic application on several dataflow architecture alternatives, showing that many different architectures can be simulated in modest time on a modern workstation.

1: Introduction

In the application domain of real-time video, the required processing power is in the order of hundreds of Risc-like operations per pixel, while the data rate of pixel streams is in the range of 10 to 100 Msamples per second. Consequently architectures are needed that perform billions of operations per second and have an internal communication bandwidth of Gbytes per second.

In the application domain of real-time video we focus on dedicated architectures that support the concept of streams [17] and achieve the required performance by exploiting the inherent parallelism of the applications on domain-specific, coarse-grain processors, with limited internal flexibility (i.e. weakly programmable). An example of such a domain-specific architecture is given in figure 1. The architecture consists of different dedicated application-specific coarse-grain processors that operate independently of each other on data-streams. These streams are exchanged between the coarse-grain processors via a communication network and is controlled by some global controller. These kinds of architectures are typically embedded in a larger system that also contains memory and a general purpose processor, e.g. a Risc processor.

In the design of these architectures, many choices have to be made. In this paper we present a simulation environment that aids the designer in making these choices based on quantitative information. In section 2 we present our problem statement. A solution approach is given in section 3. In section 4 we review related work of quantitative evaluation of design alternatives. The solution approach is further detailed for application-specific dataflow architectures in the following sections. In

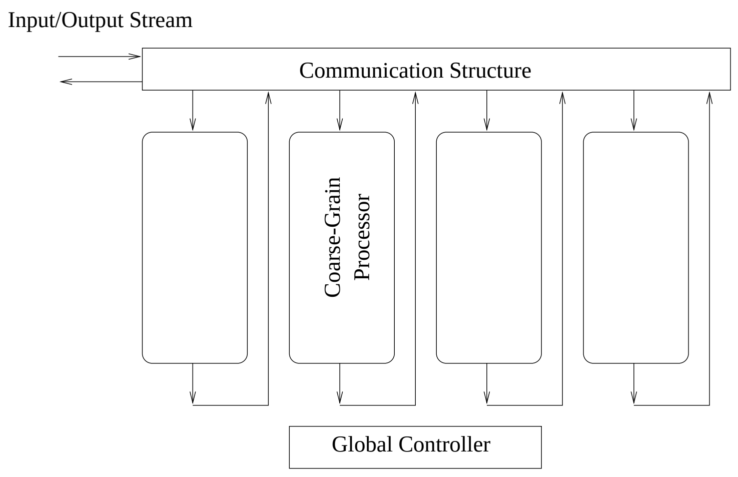

Figure 1. Example of an Application-Specific Dataflow Architecture

section 5 and 6 we detail the modeling of signal processing algorithms and the modeling of coarse-grain dataflow architectures. We discuss the construction of the configurable architecture simulator in section 7. An experiment using the environment and the simulator is described in section 8. We conclude this paper in section 9.

2: Problem Statement

When designing an architecture for a set of algorithms, the designer has first to decide which kind of architecture model is best suited for the implementation. Algorithms are characterized by their model of computation (e.g. stream-based processing, data-dependent execution, asynchronous execution). A candidate architecture should in some way support these characteristics.

The designer starts by drawing a rough model of the architecture on paper, as is done in figure 1, consisting of different elements (e.g. functional units, memory, and controllers) and their interconnection structure. Each element can be one of many different types (e.g. first in first out (fifo) buffers or random access buffers) with different parameter values (e.g. memory size and read and write times).

In the next step, the designer refines the architecture by selecting for each element, a type, and values for the parameters. In this selection process many choices between alternatives have to be considered, a process made even more difficult if a set of algorithms must be supported.

A structured approach that helps the designer in the process of refining the architecture and evaluating the alternatives is not available for the type of applications and architectures that we target. Current practice is to construct a detailed executable model (e.g. VHDL or C-code). Although a more abstract model can be constructed, a lot of detail is often incorporated. Designers may become preoccupied by details of the design, not thoroughly evaluating decisions at a higher level. The higher level decision to use a particular kind of buffer of a certain size has a greater impact on the overall performance of the architecture than the details of how a buffer communicates with a coarse-grain processor. Moreover, the more detailed the description is the harder it is to change the architecture and the lower the simulation speed will be. As a result, only a few alternatives can be evaluated, not taking full advantage of opportunities for improving the performance of the architecture.

3: Solution Approach

The general scheme we propose for performing quantitative analysis of architectural designs is shown in figure 2. This scheme provides an outline for an environment in which architectural design can be exercised. In this environment different alternatives are evaluated in a *quantitative way*. These design alternatives [1] are *rated* in some way, enabling the designer to determine that one alternative is better than another on the basis of *well defined criteria.* The analysis of the performance of an architecture under design, on which benchmark algorithms are mapped and executed, delivers the quantitative information.

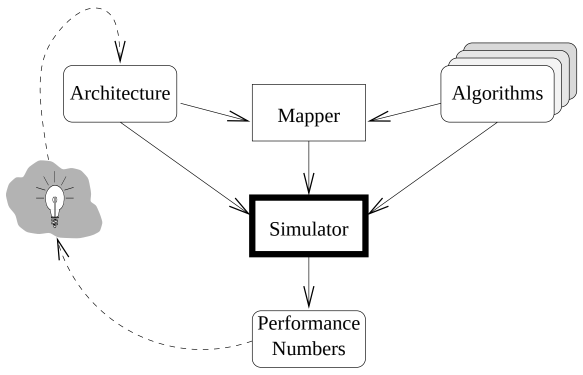

Figure 2. A General Scheme for Performing Quantitative Analysis of Architectural Designs

A key tool is the *simulator,* which can be configured for a particular architecture. The simulator yields performance numbers e.g. contention on communication structures, utilization of functional units, the filling of buffers etc. The designer analyzes these performance numbers to propose improvements of the architecture (indicated in figure 2 by the "lightbulb"). The designer iterates in the environment on the architecture until a satisfactory design is found. The faster such iterations can be done the more architectures can be evaluated. Hence, efficiency is a key requirement for the simulator. Note that the designer can use the same environment to evaluate the algorithms.

The mapping (often called compilation) is an essential element in the design of programmable architectures, and is performed in the environment of figure 2 by the mapper tool. Embodied in this tool is a mapping strategy that is developed in parallel with the definition of the architecture. The mapper tool will not be further detailed in this paper.

We intend to use the quantitative approach as given in figure 2 in the design of application-specific *dataflow architectures*. Since dynamic effects are involved in both the dataflow architectures as well as the algorithms, we obtain performance numbers by simulation. We model and simulate the architectures at a *high level of abstraction.* It permits designers to modify their proposed architectures with little effort, in order to evaluate design alternatives. Further, the abstract level will help to make simulations fast. However the architecture models must be at such level that simulation will yield *accurate* performance numbers.

4: Related Work

The quantitative approach is a well known technique in the design of general purpose computer architectures. See for example the excellent book of Hennessy and Patterson on the design of Risc

[1] Design alternatives include both the selection of a type of an element and a parameter setting for an element.

based architectures [4], using a quantitative approach. Camposano and Wilberg used the approach for designing application-specific VLIW architectures for low-speed video algorithms [2]. Rathnam and Slavenbrug used the approach for the design of the TriMedia programmable multi-media processor TM1 [14]. For DSP processors, the approach has been used by Živojnović et al. [16]. In all cases, refinements were made to known architectures for which good detailed models and compilers existed.

In the area of high performance digital signal processing (e.g. video), new types of dataflow architectures are emerging [6, 1, 7]. These types of architectures support stream-oriented, coarse-grain, data-dependent DSP algorithms, exploiting parallelism at the task level. Current practice in the design of these architectures is to construct a single detailed model in some executable language e.g. VHDL or C-code.

5: Algorithms

Digital signal processing algorithms can be represented in a natural way by dataflow models of computation like SDF or DDF [10]. These models support the concept of *streams* efficiently and retain the level of parallelism available within the algorithm.

We use the Kahn Process Network model of computation [11] to specify the set of video algorithms shown in the upper right part in figure 2. This Kahn model of computation describes dynamic algorithms, but in contrast to the DDF-model of computation, it yields a deterministic execution trace [9]. In Kahn graphs, nodes represent processes that execute functions and edges represent unbounded fifo buffers. If a process reads from a buffer, the execution of the process is blocked if no data is available. The execution of a process which writes to a buffer can always proceed, since the buffers are unbounded.

We assume that the functions in the Kahn graph are coarse-grain and operate on streams of samples. Typical coarse-grain functions are for example a "sample rate converter" or a "filter". These functions are free of any side-effects, may contain state, and the output is completely determined by the sequence of the input samples.

An example of a Kahn graph is shown in Figure 3. This is the Kahn graph of a picture in picture (PiP) algorithm, an application used in modern TVs. This algorithm reduces a picture to half its size in both horizontal and vertical direction and places the reduced picture onto a full screen picture showing two images on a tv screen.

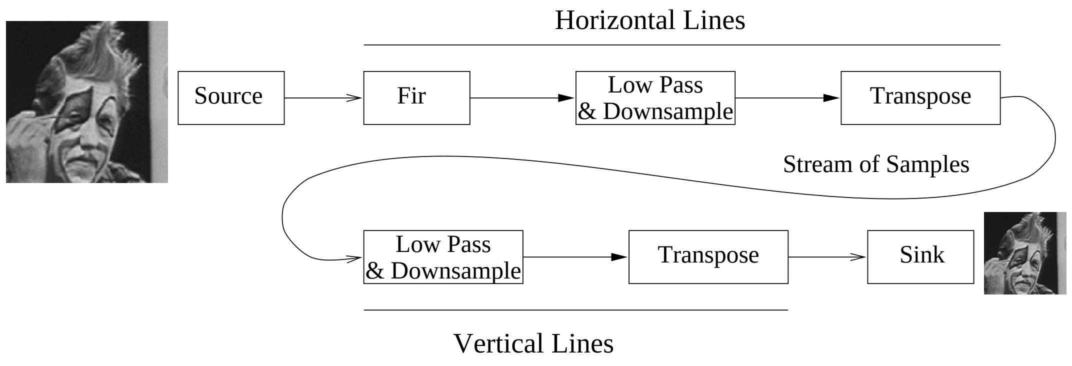

Figure 3. The Kahn Graph for Part of the Picture in Picture Algorithm

In the example, a stream of video pixels is filtered by a 18-taps FIR-Filter and passed through a 2-taps Low Pass filter, which performs a down sampling of a factor two. Next the image is transposed: re-ordering samples in such a way that two consecutive samples belong to two different video lines. The stream then passes a Low Pass filter again, this time to perform a vertical down sampling of a factor two. Then the second transpose function is performed on the stream resulting in consecutive samples now belonging to one and the same video line. Finally the samples are sent to a sink. A dynamic algorithm is obtained if the sizing of the reduced picture changes during the execution.

6: Stream-Based, Coarse-Grain Dataflow Architectures

Dataflow machines, and especially fine-grain dataflow machines, are no longer considered to be a viable option for general purpose computations. For DSP algorithms however, the dataflow model of architecture is a natural fit. Especially the stream-based, coarse-grain dataflow machines overcome many of the problems encountered in fine-grain dataflow architectures. The coarse-grain dataflow architectures enable efficient implementations for high performance digital signal processing [6, 1, 7]. For video algorithms described in a static dataflow model of computation, a programmable architecture [8] and dedicated architectures [13] already exist.

In this section we present a model of architecture for dynamic dataflow (e.g. the Kahn model). This model can be represented as an architecture template. This template represents a class of architectures from which individual architectures can be instantiated. These instantiated architectures serve as input to the simulator as shown in the upper left part in figure 2.

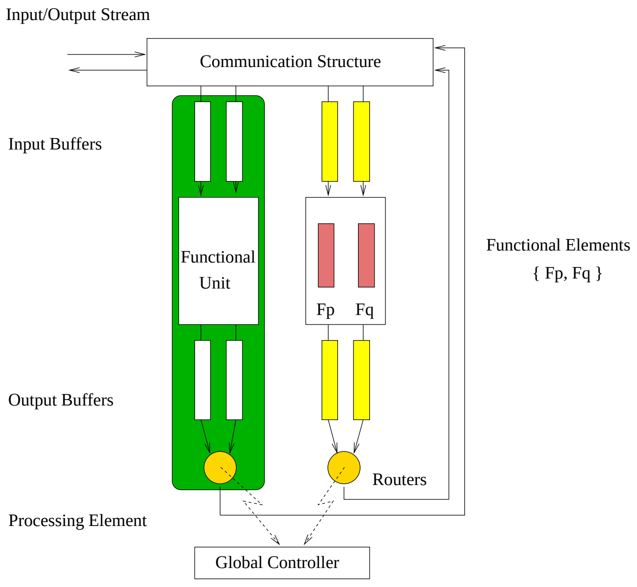

Figure 4. The Dataflow Architecture Template

6.1: The Architecture Template

The dataflow architecture template is based on a model proposed in [6] and is described in figure 4. The architecture template is a more elaborated description of the architecture given in figure 1. According to the architecture template, an architecture consists of several elements; a *global con-*

troller, a *communication structure*, and a number of *processing elements* (PEs), connected to the communication structure.

A PE consists of one *functional unit* (FU), a number of input buffers, and a number of output buffers. A FU consists of one or more *functional elements* (FE) of a certain type. This collection of FEs is called the *function repertoire* of a FU and is represented as $\{F_p, F_q\}$ in figure 4. Only one FE of the FU can be active at the same time and the FE executes only one function of the repertoire (e.g. F_p). The *grain size*, counted in equivalent Risc instructions like add or compare, indicates the complexity of a FE. In our architecture the FEs implement coarse-grain functions with a grain size of 10 .. 100, at least ten times as many as fine grain functions. A FE can be pipelined.

Data streams are divided into a number of *packets* with a certain packet length. The *header* of each of these packets contains the routing of the packet, the function to be called from the function repertoire, and the length of the data stream. The body of the packet contains the actual data, e.g. video samples. Packets of different lengths can exist simultaneously within the architecture.

A *Router* is located after each output buffer. Via such a buffer, packets can flow from one PE to the next PE, determined by short interactions between the Routers and the Global Controller. Hence, the length and the routing of the packets can change *dynamically* during the flow through the dataflow machine.

6.2: Architecture description

In our environment, the designer describes an instance of the architecture template of figure 4 in a textual format. The architecture description serve as input to the simulator. In figure 5, an example of an architecture description is given. The architecture is called "dataflow" and consists of a switch matrix, a controller, and one processing element with one input buffer and one output buffer, which are both connected to the FU "Filter". The FU "Filter" contains two FEs that execute the functions "LowPass" and "HighPass" respectively. A binding specifies the connection between the ports of the FEs and the buffers of the FU.

A certain *type* is chosen for each of the elements in the description. If the "Fcfs" type is specified the controller implements a first-come-first-served protocol. The elements are also *parameterized.* In the example of figure 5, a controller is specified that handles one request at the time, taking five clock-cycles to serve each request.

Instead of a "Fcfs" protocol, other protocol types like "Round Robin" or "Random" could have been selected. In table 1, the available types are given for various elements. We can specify different architecture instances, by selecting another type for an element or using different parameters.

7: The Simulator

The simulator plays a central role in the environment as given in the bold marked box in figure 2. The simulator constructs a fast executable model from an architecture description as given in figure 5, resulting in performance numbers which are clock-cycle accurate. To get the correct dynamic execution trace the video algorithms have to be executed in a clock-cycle accurate way.

7.1: Performance Modeling

The performance of architectures as given in figure 4, can be described in a clock-cycle accurate manner at a high level of abstraction by using *performance models.* The implementation of *data synchronization* between parallel processing elements as well as *mutual exclusivity* of shared resources,

```
Architecture Dataflow {
   GlobalControl { PacketLength = 20; }
   Communication { Type: SwitchMatrix; }
   Controller { Type: Fcfs( 1, 5 ); }
   ProcessingElement Filter(1,1) {
      InputBuffer { Type: BoundedFifo( 100 ); }
      OutputBuffer { Type: BoundedFifo( 100 ); }
      Router { Type: Single; }
      FunctionalUnit {
         Type: Packet;
         FunctionalElement LowPass(1,1) {
            Function { Type: LowPass(throughput=1,latency=18); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
         }
         FunctionalElement HighPass(1,1) {
            Function { Type: HighPass(throughput=1,latency=10); }
            Binding {
               Input ( 0->0 );
               Output ( 0->0 );
            }
         }
      }
   }
}
```

Figure 5. An Example of an Architecture Description

determines the performance of parallel architectures. Describing architectures at this high level requires less detail to be specified and simulated, and therefore facilitates changes to the architecture and enables faster simulations.

To capture both data synchronization and mutual exclusivity, the *PAMELA* method [15] has been developed at the Delft University of Technology. PAMELA defines a language that describes parallel systems using only a few constructs; parallel processes (*pam_fork*), semaphores (*pam_P* and *pam_V*), and time delays (*pam_delay*). Semaphores are used to synchronize the exchange of data between processes or to give exclusive access to shared resources. Processes advance in time explicitly through the use of the *pam_delay* statement or implicitly by waiting for a *pam_P* operation. Within the PAMELA method, time is measured in units and we model one unit to be equivalent to one clock-cycle in the architecture.

7.2: The Software Implementation

A configurable simulator has been constructed in C++ code. The use of this object-oriented programming language turned out to be a key aspect in the construction of a fast and still flexible simulator.

The PAMELA constructs have been implemented in a run-time library as C-code functions, on top of a multi-threading package. We use the run-time library as the parallel simulation engine. On top of this engine we have built a model of the architecture, using C++.

Using the technique of inheritance, we have implemented the different types for the various elements given in table 1 as C++ classes. The simulator reads an architecture description file as given in figure 5. The different elements of the architecture are instantiated as 'building blocks' in memory, while reading the file. We resolve and fix the interconnections between the elements at instantiation

Table 1. Available Types for the Different Elements

Element	Type	Element	Type
Buffers	Fifo Buffer Priority Buffer Unbounded Fifo Matching Unit	Functional Units	Sample-Switching Packet-Switching Pipelined Non-Pipelined
Comm. Structure	Switch Matrix Omega Network Shared Bus	Controller	Fcfs Round Robin Random
Routers	Single Shared	Memory	Main Memory Transpose Memory

time, avoiding the need to evaluate connections at run-time, which would introduce a lot of overhead.

The read and write functions of a fifo buffer are shown in figure 6, illustrating the use of the PAMELA constructs `pam_P`, `pam_V`, and `pam_delay`. The PAMELA function calls are printed in italics. In this fifo buffer, samples are stored in a buffer `buffer` of a size `capacity`.

When a process (e.g. a FE) tries to read a sample from a fifo buffer, it first executes `pam_P(data)` decrementing the semaphore `data`. If no sample is available, the semaphore `data` equals zero and the reading process will block on this statement (implementing a blocking read). Otherwise the process continues by reading the sample. A blocked process has to wait until semaphore `data` becomes larger than zero. By writing a sample to the buffer, the function `pam_V(data)` is executed, incrementing semaphore `data` to a non-zero value. The blocked process can now proceed and read the available sample. The semaphore `room` is initialized to the capacity of the buffer and is used to implement a blocking write when the buffer is full.

```
void Fifo::write(Sample* a)
{
  pam_P(room);                       // Is there Room on the Fifo?
  buffer[writefifo] = a;            // Write in buffer
  writefifo = (++writefifo)%capacity;
  pam_delay(1);                     // It takes 1 clock-cycle to write
  pam_V(data);                      // Tell there is data available
}

Sample* Fifo::read(void)
{
  pam_P(data);                      // Is there data available?
  Sample* tmp = buffer[readfifo];   // Read from buffer
  readfifo = (++readfifo)%capacity;
  pam_delay(1);                     // It takes 1 clock-cycle to read
  pam_V(room);                      // Tell there is room again

  return tmp;
}
```

Figure 6. An Implementation of the Read and Write Functions of a Fifo Buffer

We use PAMELA processes to model Functional Units, Functional Elements and Routers of the architectures given in figure 4. These processes exchange data (e.g. the samples of packets) with other processes (other FUs, FEs or Routers) via the communication structure or via buffers, which are both modeled using semaphores. The controller is modeled as a process and is accessed by different Routers. Using semaphores we can grant exclusive right to one of the Routers, while the other Routers have to wait. The PAMELA run-time library orders (i.e. schedules) the processes dynamically in time based on the availability of data, implementing the architecture's data-driven execution model.

7.3: Metric Collectors

Metric Collectors gather performance numbers, at run-time, for all kinds of elements. For example, the collectors observe how long a semaphore was blocking a process. The "response time of the controller" is obtained this way by observing how long a Router waits for the controller to become available. Other performance metrics are for example: utilization, distribution of the buffer filling, number of operations executed by a FE. Some metrics for different elements are given in table 2. A special metric collector gathers information of the complete architecture, like the number of executed operations and the total execution time in clock-cycles. These two numbers are used to derive the performance metric "parallelism".

Table 2. Implemented Metrics for the Different Elements

Element	Metric
Comm. Structure	Utilization
Controller	Utilization
Buffer	Filling distribution
Routers	Response Time Controller
Functional Unit	Utilization, Number of Context Switches
Functional Element	Utilization, Pipeline Stalls Throughput, Number of Operations
Architecture	Number of Operations, Total execution time

7.4: Functional Elements

The function performed on a Functional Element can be described as a C-code function that accepts and produces samples. Via the technique of function overloading, a C-code function is instantiated from a Library onto a FE. This Library contains a set of C-code functions. In case of figure 5, the functions "LowPass" and "HighPass" are instantiated on the architecture. The instantiation of C-functions on the FEs is necessary to obtain the correct data-dependent execution of algorithms on the architecture.

The throughput and latency are determined for each function of the FE. For example, a throughput of 1 and a latency of 18 for a "LowPass" function indicates that an 18-stage pipelined version is being instantiated.

We re-used the PAMELA run-time library to build a *Kahn-graph simulator.* We can use this simulator to simulate the algorithm given in figure 3. The blocking reads of the Kahn model can easily be modeled with semaphores [12]. Since the data-driven execution of the Kahn-graph nodes mirror the execution of FEs, and the Functions are functional, we can use the same C-code functions in both the architecture simulator and the Kahn-graph simulator. The Kahn-graph simulator is not indicated in figure 2.

7.5: Mapping

In our environment the architecture description file allocates a number of FEs of a certain type. The mapping of a Kahn graph on the architecture involves the assignment of Kahn-graph nodes of a certain type onto FEs with the same type. Because the architecture is data-driven, the scheduling of the FEs is done during the simulation. Therefore we only have to specify the static FE assignment as a mapping in a separate file. The FE assignment is a combinatorial problem. Currently the designer has to specify the assignment by hand in a file, but the automation of the mapping process is subject of research that is performed concurrently. Later on the mapping results will be integrated in our simulation environment.

8: Results

In the architecture given in figure 4, PEs operate in parallel and a single, centralized controller is used. This controller can easily become a bottleneck in the machine. Using the environment given in figure 2, we investigate the relationship between the *packet length*, the *service time* of the controller, and the achieved level of *parallelism* in the machine for part of the picture in picture (PiP) algorithm described in section 5.

For the experiment, we defined an architecture with seven FUs, as explained in section 6. Each FU captures one of the functions from figure 3 including the sink and the source functions. An one-to-one assignment of the functions of the algorithm onto the FEs of the architecture, is performed. In general many-to-one mappings are also supported, but not used in this experiment. In the experiment we select 5 values for the parameter *packet length* in the range of $\{5..200\}$ samples and 5 values for the parameter *service time* of the controller in the range of $\{1..20\}$ clock-cycles, resulting in 25 different architectures. For each architecture we measured the achieved parallelism and utilization of the controller while processing two small video frames with 14,400 video samples per frame.

The results are presented in figure 7 and figure 8. It took the simulator 16 minutes to find the performance numbers for the 25 architectures. A piece-wise linear interpolation model is used to find the parallelism and utilization for the whole range of parameters based on only 25 points. The presented figures show the relationship between two parameters for a fixed value for the other parameters. The presentation of multi-dimensional performance numbers is a problem on its own and we refer the reader to [3] for more information.

From the simulations, we can conclude that the controller service time has a significant influence on the obtained level of parallelism in the investigated architecture. At points were the parallelism is low, we notice that the controller is utilized for almost 100%, indicating that the controller is indeed the bottleneck in achieving more parallelism. We show again the utilization of the controller in figure 9, for different packet lengths for various service times, for clearness sake. By increasing the packet length, the load on the controller is lowered indicating a trade-off between service time and packet length. Initial investigations [6] indicate that a controller service time of 4 clock-cycles is possible. In that case, the controller is no longer a bottleneck when a packet length greater than 60 is selected. The controller utilization is then 25% or less.

The simulation speed is essential to perform a study of several architectures. The faster we can iterate in the environment of figure 2, the more architectures, and thus alternatives, can be evaluated. The current version of the simulator can simulate 10.000 C-function calls per second instantiated on arbitrary FEs with all metric collectors active. This means that our simulator needs 9 minutes to process a full video picture of 720×576 pixels through 10 FUs with each implementing one FE including execution of the functions instantiated on the FEs.

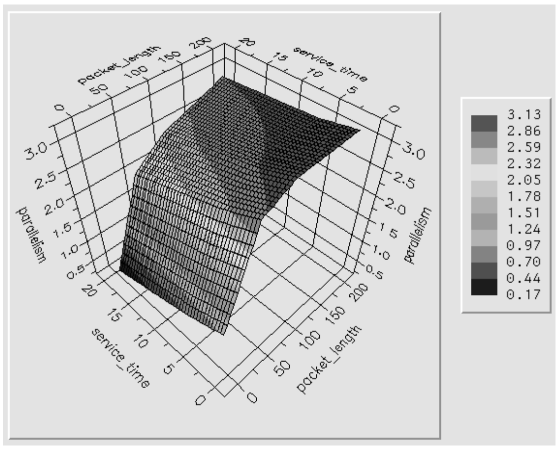

Figure 7. Achieved Parallelism in Operations per Clock-Cycle for Packet Length versus Service Time

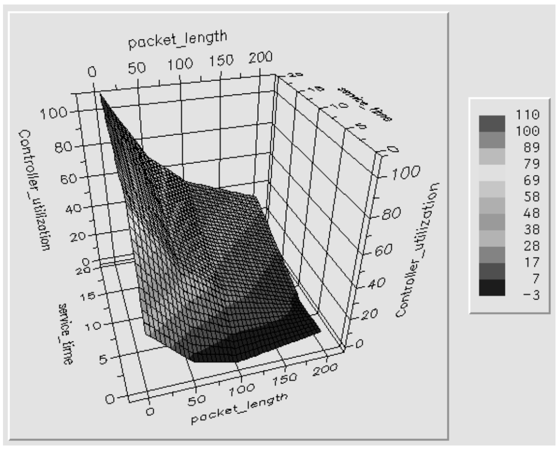

Figure 8. Utilization of the Controller in Percentage for Packet Length versus Service Time

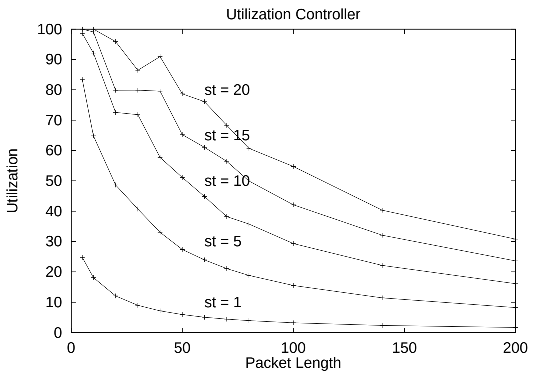

Figure 9. Utilization of the Controller in Percentage versus Packet Length for various Service Times

The dataflow architecture has also been implemented in VHDL in more detail at the RTL-level [6]. Our simulator is about 5 to 10 times faster than this VHDL model, since we simulate at a higher level of abstraction. The results shown in figure 7 and figure 8, produced by the abstract model, could also be obtained via the same VHDL model but in much more simulation time. The architecture simulator is not only faster, it is also configurable, and can implement any functionality on the FEs.

We currently use the presented environment in a study of an IC that will be applied in video applications for the consumer market. The environment gives us the opportunity to compare many different architecture alternatives. We do not have the intention to synthesize architectures with this system. Subject for further research is how to explore the design space of these architectures in a systematic and automatic way. Interesting work in this direction is done by Teich *et al* [5].

9: Conclusion

We have presented an approach for the quantitative analysis of application-specific dataflow architectures. We described how we specify video algorithms as Kahn graphs and how we can instantiate, in a textual format, different architectures from an architecture template. A configurable simulator in C++ has been constructed for the architecture template, using multi-threading and object oriented programming techniques. This simulator can efficiently execute different types of dataflow architectures at a level that is clock-cycle accurate. Results have been presented using this simulation environment for a realistic example. The abstract architecture simulator could simulate 25 different architectures executing the Picture in Picture benchmark algorithm in only 16 minutes of execution time on a modern workstation. This is a useful evaluation environment for a designer to evaluate design alternatives in application-specific dataflow architectures.

10: Acknowledgments

We thank the colleagues of Philips Research who work on the Prophid architecture for sharing their results on the architecture and VHDL model.

References

[1] Arthur Abnous and Jan Rabaey. Ultra-low-power domain-specific multimedia processors. In *VLSI Signal Processing, IX*, pages 461–470, 1996.

[2] R. Camposano and J. Wilberg. Embedded system design. *Design Automation for Embedded Systems*, 1(1):5 – 50, 1996.

[3] Margaret L. Simmons Cherri M. Pancake and Jerry C. Yan, editors. *Performance Evaluation Tools for Parallel and Distributed Systems*, number 11 in IEEE Computer, November 1995. Theme Feature.

[4] John L. Hennessy and David A. Patterson. *Computer Architectures: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., second edition, 1996.

[5] J. Teich, T.Blickle and L. Thiele. An evolutionary approach to system-level synthesis. In *Proceedings of the Fifth Int. Workshop on Hardware/Software Codesign*, pages 167 – 171, March 24-26 1997.

[6] Jeroen A.J. Leijten, Jef L. van Meerbergen, Adwin H. Timmer and Jochen A.G. Jess. Prohid, A Data-Driven Multi-Processor Architecture for High-Performance DSP. In *Proc. ED&TC*, March 17-20 1997.

[7] V. Michael Bove Jr. and John A Watlington. Cheops: A reconfigurable data-flow system for video processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(2), April 1995.

[8] K.A. Vissers, G. Essink, P.H.J. van Gerwen, P.J.M. Janssen, O. Popp, E. Riddersma, .J.M. Veendrick. *Algorithms and Parallel VLSI Architectures III*, chapter Architecture and programming of two generations video signal processors, pages 167 – 178. Elsevier, 1995.

[9] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[10] Edward A. Lee. Design Methodology for DSP. Technical Report 92-084, University of California at Berkeley, 1992-1993.

[11] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.

[12] Tom Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, 1995.

[13] P.E.R. Lippens, J.L. van Meerbergen, A. van der Werf, W.F.J. Verhaegh, B.T. McSweeney, J.O. Huisken and O.P.McArdle. PHIDEO: A silicon compiler for high speed algorithms. In *Proc. EDAC*, pages 436 – 441, 1991.

[14] S. Rathnam and G. Slavenburg. An architectural overview of the programmable multimedia processor tm1. In *Proc. Compcon*. IEEE CS press, 1996.

[15] Arjan J.C. van Gemund. Performance Prediction of Parallel Processing Systems: The PAMELA Methodology. In *Proc.* 7^{th} *ACM Int. Conference on Supercomputing*, pages 318–327, July 19-23 1993.

[16] Vojin Živojnović, Stefan Pees, Christian Schläger, Markus Willems, Rainer Schoenen, and Heinrich Meyr. DSP Processor/Compiler Co-Design: A Quantitative Approach. In *Proc. ISSS*, 1996.

[17] John A. Watlington and V. Michael Bove Jr. Stream-based computing and future television. In *Proc. of the* 137^{th} *SMPTE Technical Conferenc*, pages 69–79, September 1995.