The Construction of a Retargetable Simulator for an Architecture Template

Bart Kienhuis[1,2], Ed Deprettere[1],Kees Vissers[2], Pieter van der Wolf[2]
[1] Delft University of Technology
[2]Philips Research Laboratories Eindhoven
E-mail: kienhuis@cas.et.tudelft.nl

Abstract

Systems in the domain of high-performance video signal processing are becoming more and more programmable. We suggest an approach to design such systems that involves measuring, via simulation, the performance of various architectures on which a set of applications are mapped. This approach requires a retargetable simulator for an architecture template. We describe the retargetable simulator that we constructed for a stream-oriented application-specific dataflow architecture. For each architecture instance of the architecture template, a specific simulator is derived in three steps: the architecture instance is constructed, an execution model is added, and the executable architecture is instrumented to obtain performance numbers. We used object oriented principles together with a high-level simulation mechanism to ensure retargetability and an efficient simulation speed. Finally, we explain how a retargetable simulator can be encapsulated within an environment for automated design space exploration.

1. Introduction

In the past, systems in the domain of high-performance video signal processing were often designed for only a single application. Nowadays, however, these systems have become more *programmable* and they must be able to support a *set* of applications instead of a single application. Programmability is important for two reasons: First, it permits adjustments which, for example, arise from evolving standards, and secondly, it offers the possibility of function sharing, which is cost effective in multi-functional silicon products.

Architectures for a single application are derived in various steps of refinement until a final architecture is found that can be synthesized to silicon. In the case of a set of applications, this refinement method would result in one architecture for each application of the set. A different approach is to start from an *architecture template* from which individual architectures can be instantiated. The problem then is to assess the quality and validity of such an architecture instance for the targeted set of applications. We want to use an approach in which many alternative architectures can be evaluated *quantitatively*. This provides an objective basis on which eventually one architecture can be selected that performs best for a set of applications.

We measure the *performance* of an architecture instance on which a set of applications is mapped and executed as suggested in [1]. This approach, henceforth called as the Y-chart (not to be confused with Gajski and Kuhn's Y-chart [2]), is shown in Figure 1. We need to be able to describe various architectures and sets of applications and to derive mappings for applications onto architectures. We also need to make and analyze a model of an architecture such that we can obtain its performance numbers. System with better performance can be achieved by making adjustments in the architecture as indicated by the "lightbulb" in Figure 1. A better performance can also be achieved by changing the way the algorithms are described or choosing a different mapping. We focus here solely on the architecture.

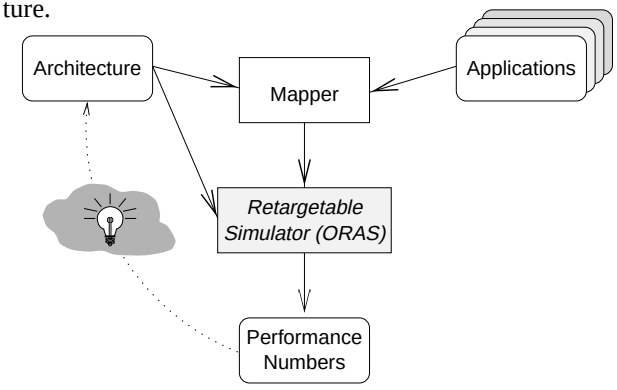

Figure 1. The Y-chart approach

An *architecture template* describes in a parameterized form a class of architectures by giving the available element types and the composition rules for these elements types. The architecture template parameters need to be assigned values to obtain an instance. All these parameters together span a large *design space*. We can *explore* this design space by changing some of the parameter values in a structured way and evaluating the performance of an architecture instance.

At Philips Research a particular class of stream-oriented application-specific dataflow architectures has been investigated [4]. The dataflow architecture is to be used in video

applications for the consumer market. An example of such a dataflow architecture is given in Figure 2. It consists of different application-specific processors operating independently of each other on data streams. The processors can perform *coarse-grain* functions like "filtering" or "sample rate conversion". The streams are exchanged between the coarse-grain processors via a communication network which is controlled by some global controller.

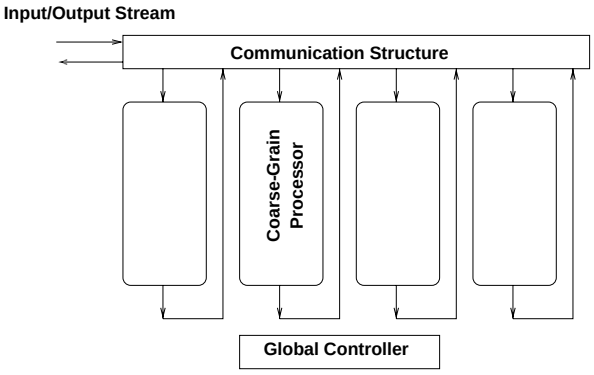

Figure 2. Stream Based Dataflow Architecture

Because the video applications of interest are data-dependent, we use simulation to get clock-cycle accurate performance numbers. As a consequence, a *retargetable simulator* [6] is needed for this class of dataflow architectures. We have constructed a retargetable simulator for the dataflow architecture. In this paper we explain how we have constructed our Object Oriented Retargetable Architecture Simulator (ORAS) and how ORAS derives a specific simulator for one particular architecture instance.

The organization of this paper is as follows. We present the problem statement and requirements in Section 2. Related work is discussed in Section 3, followed by our solution approach in Section 4. In Sections 5, 6, and 7 we elaborate on the three steps suggested in the solution approach. In Section 8 we show how a retargetable simulator can be used in a design space exploration. We present conclusions in Section 9.

2. Problem Statement & Requirements

The problem we address in this paper is how to build a retargetable architecture simulator for a class of dataflow architectures and how specific simulators can be derived for members of this class. The retargetable simulator is used to explore the design space of this class of architectures. This exploration is performed by specifying many different architecture instances. We would like to automate this process by having an *exploration environment* where new architecture instances are created and evaluated in a structured way. The architecture instances that are specified need to be *valid* members of the class of dataflow architectures we look at. Therefore we need to verify each architecture instance with respect to the architecture template. A specific simulator is derived for each architecture instance. The *speed* of this simulator determines how many instances can be exercised within a certain amount of time in the design space. This speed requirement is especially important in the video domain. A certain amount of overhead will be incurred to support *retargetability* in the retargetable simulator. This overhead reduces the simulation speed and should be kept as small as possible. We know the class of dataflow architecture and thus the kind of retargetability needed. We should exploit this fact to get efficient retargetable simulators.

A specific simulation can be done on different levels of detail with different *accuracy*. Although in the design of an architecture it is often enough to compare different architectures on the basis of *relative* performance numbers, in later design stages comparisons must be based on detailed performance numbers. However, there is a trade-off between simulation accuracy and hence simulator complexity, and simulation speed [3]. A *clock-cycle accurate* simulation takes longer than an *instruction accurate* simulation because more detail needs to be simulated. We looked at different general purpose simulators and found the following number of instructions per second: 200,000 for an *instruction set* simulator, 40,000 for a *clock-cycle accurate* simulator and 500 for an *RTL accurate* simulator in VHDL. Simulating, for example, one video frame of 720x576 pixels by a simple video algorithm of 300 RISC-like instructions per pixel on each of these simulators, requires respectively 10 minutes, 54 minutes and more than a whole day. Hence the selected accuracy of a specific simulator which is to be used in the Y-chart depends very much on the level at which design decisions must be made.

3. Related Work

The Y-chart approach defines the need for retargetable architecture simulators. The approach is common practice in the design of instruction set processors, where dedicated simulators are built in the C programming language mainly for performance reasons [3]. The C language, however, does not support parallelism. In the RASSP project, VHDL is used to perform dedicated high-level architecture simulations of high performance signal processing systems [9]. VHDL offers a parallel execution model and allows architecture models to be instrumented to get performance numbers, but it cannot be used to derive different architecture instances. Only when very high-level architecture models are used, is the execution speed of VHDL acceptable. The Ptolemy [5] environment is very useful for the specification of algorithms, but is less suitable for architecture explorations. The Atrade tool [8] is reported to implement a kind of Y-chart using the Ptolemy environment. It is only applicable for static applications defined in the SDF-domain.

4. Solution Approach

A specific simulator is derived for an architecture instance in three steps:

1. An architecture instance is constructed from a textual architecture description.
2. An execution model is added for the simulation of the architecture instance.
3. The architecture instance is instrumented with *Metric Collectors* to extract performance numbers for selected performance metrics.

The first step concerns the structure of architecture instances. We used *object oriented principles* extensively and generated a *parser* to read instances of the architecture template of the dataflow architecture in a *building block* fashion. In the second step, an executable instance is obtained by adding an execution model to the architecture structure, based on a multi-threading package. Finally, performance numbers are extracted during a simulation by instrumentation of the architecture instance with probes or Metric Collectors. These three steps, which are implemented in ORAS, are shown in Figure 3. Next we explain in more detail the three steps.

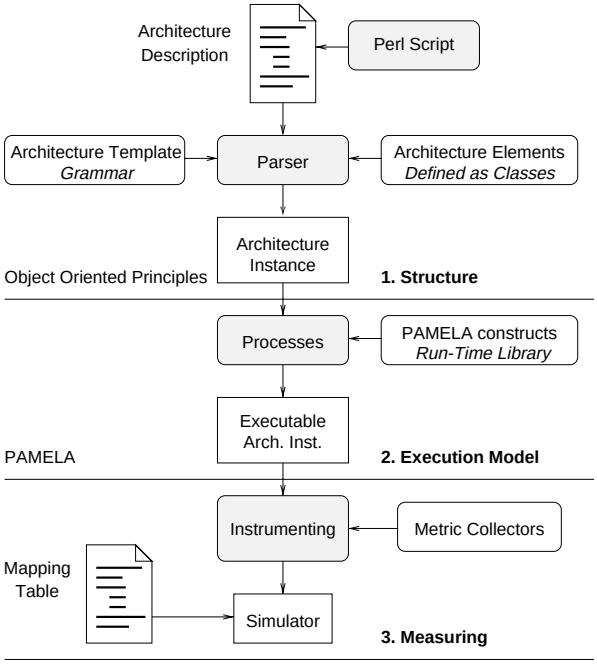

Figure 3. The 3 steps used within the Object oriented Retargetable Architecture Simulator (ORAS)

5. Construction of Architecture Instances

The first step taken in the retargetable simulator is to construct an architecture instance. This instance has to be a feasible member of the class of dataflow architectures. What are the requirements for the programming language that is used to specify this structure?

Composition: Support the idea of element types, which are the main constituents, as building blocks.

Verification: Allow the verification of compositions of element types.

Implementation: Allow element types to have different implementations.

High Level Constructs: Provide high-level programming constructions for effective and efficient compositions of element types.

The *object oriented* language C++ is well suited to model the structure of architectures [10]. A *parser* is used to check whether the architecture instances that were modeled are feasible.

5.1. Object Oriented Principles

Objects are instantiated from class descriptions. In the definition of a class, a strong separation is made between the *interface* of an object method and the *implementations* of this method. This allows us to specify the structural relationship between objects and the behavior of the objects independently. We use this separation idea extensively to implement alternative implementations (e.g. handshake, bounded, or unbounded Fifo) for the same architecture element type (e.g. a Buffer).

For each element type in the architecture there is an *abstract class* description. This description defines a *general* interface to all other objects in the architecture. For the element type Buffer, for example, we define the methods *read* and *write*. Other element types (e.g. a router or communication structure) can use the methods read and write to interact with the element type Buffer, without knowing the particular implementations of these two methods. New classes are derived from the abstract class, using *inheritance*. The new classes implement the methods of the abstract class using *late binding* or *polymorphism*.

To illustrate the separation idea, we have defined an architecture element type *Buffer* as an abstract class. Data can be stored (write) in a buffer or data can be retrieved (read) from it, as shown in Figure 4. The read and write methods must be implemented differently for a handshake buffer, a bounded Fifo buffer, or an unbounded Fifo buffer. Consequently we implemented the methods `read` and `write` differently for the various buffers. We implemented a single buffer position, a circular buffer, and a linked list for the handshake buffer, the bounded Fifo, and the unbounded Fifo respectively.

We have already defined many different element types (like *Buffer*, *Controller*, *Router*, *Functional Element*, and *Functional Unit*) as well as various implementations for our dataflow template. They are stored in the library of Architecture Elements as shown in Figure 3.

Recently the standard C++ programming language has become supported by a very powerful library called *Standard Template Library* (STL) [7]. This library implements

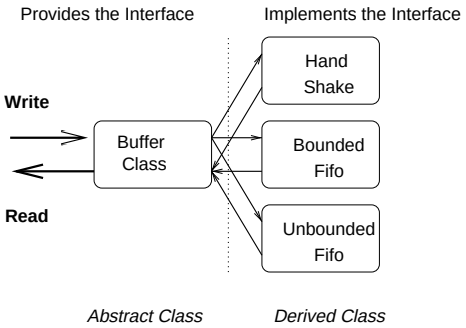

Figure 4. The separation between interface and interface implementation

very efficiently high-level programming constructs like vectors, lists, and maps, to name a few. This has proved to be very useful in the design of our retargetable simulator.

5.2. Architecture Description

We use a *parser* to break up a textual *Architecture Description* into architecture elements and to check if the description is a feasible instance of the architecture template. An architecture template is described in terms of a *grammar*. The grammar is used by the parser to matches compositions of architecture elements as shown in Figure 3. If a feasible composition is found, the elements are instantiated from the library of architecture elements. This process is repeated until a complete architecture instance is constructed. Because we specify architectures at the level of architecture elements we achieve concise architecture descriptions. It takes, for example, about 150 lines of code to describe realistic video processing architecture instances.

6. Execution Model

We simulate dataflow architectures to get performance numbers. As explained before, the simulation speed, and thus level of modeling, is an important feature of a retargetable simulator. The PAMELA work of Van Gemund [12] shows that it suffices to model two constraints in parallel architectures, namely *condition synchronization* of data and *mutual exclusivity* of resources, to get clock-cycle accurate performance numbers. We can model these constraints using the concepts of *processes*, *semaphores* and *delays*.

These concepts are implemented as C functions within the PAMELA run-time library (RTL). This is a *multi-threading* package that performs the necessary process scheduling and has, unlike many other such packages, the notion of *virtual* time.

To simulate our architecture instance, a parallel *execution model* like the RTL is needed that governs the order in which the various architecture elements can make progress. The architecture elements built in C++ and the RTL execution mechanism are coordinated via the PAMELA constructs. Figure 5 shows how the write method of a bounded Fifo buffer is combined with PAMELA constructs. The Fifo buffer is a particular implementation of the architecture element type *Buffer*. The semaphores used in the write method, are `room` and `data` and the delay is modeled via `pam_delay`. We use the delay statement to model that a write to a Fifo buffer will take one time unit, which is equivalent to one clock cycle in the architecture.

Notice that within the figure we actually store samples within an array `buffer` of size `capacity` in a specific order determined by the `writefifo` variable. We don't need the buffer functionality in order to do performance analysis; we could suffice with using only PAMELA constructs. However, to get the correct *run-time* behavior of the architecture, we also perform these *functionally correct* simulations to get more accurate and realistic performance numbers.

```
void Fifo::write(Sample* a)
{
 pam_P(room);   // Is there Room in the Fifo?
 metricCollector->histogram(token++); //Measurement
 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
}
```

Figure 5. The Write Method of a Bounded Fifo Buffer

The PAMELA processes are instantiated in the second step of ORAS, as shown in Figure 3. Since the architecture structure is built in the first step, the processes do not have to decode the structural aspects of the architecture at run-time. This results in an efficient but still flexible simulator. The ORAS simulator can execute 10,000 coarse-grain functions per second. These functions operate on samples.

7. Metric Collectors

To obtain performance numbers in the ORAS, we instrumented the architecture elements in the third step with Metric Collectors. They collect various high-level performance numbers during the execution of the simulator and present results, possibly in statistical form, at the end of a simulation. In Table 7 some performance metrics are given for the element types of the dataflow architecture. The Metric Collectors gather information about the complete architecture, such as the number of executed operations or the total execution time in clock cycles. These numbers are used, for example, to evaluate the performance metric "parallelism". Other collectors measure how long a semaphore blocked a process. This is a way of measuring the "response time" or "waiting time". In the code shown in Figure 5, the Metric Collector `metricCollector` determines the filling distribution within the Fifo buffer. Each time a token is written into the Fifo buffer, the collector determines how many tokens are present in the buffer at that write time. At the end

Element Type	Performance 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

Table 1. Implemented Performance Metrics for the Different Element Types

of a simulation a histogram of the Fifo buffer filling is made for every Fifo buffer in the architecture.

8. Design Space Exploration

Given the retargetable simulator ORAS, we can perform a design space exploration. We need to select parameter values from a range in a structured way. We create different textual architecture descriptions with these parameters, using the versatile scripting language *Perl* [13], as shown in Figure 3. A parameter value can represent a value for an element parameter (e.g. the size of a Fifo buffer), a structural aspect of the architecture (e.g. the number of Processing Elements used), or an alternative implementation of an element type (e.g. select a bounded Fifo buffer or unbounded Fifo buffer).

A specific simulator is derived in the ORAS from each of the resulting architecture descriptions. To simulate the architecture, we also need mappings as explained in the Y-chart. Therefore we down-load a textual mapping table into the specific simulator as shown in Figure 3. A different table is down-loaded for each application of the targeted set. When all simulation runs are completed, we calculate graphs from the stored performance numbers. An example of a visualization of an exploration is shown in Figure 6 and discussed in [1].

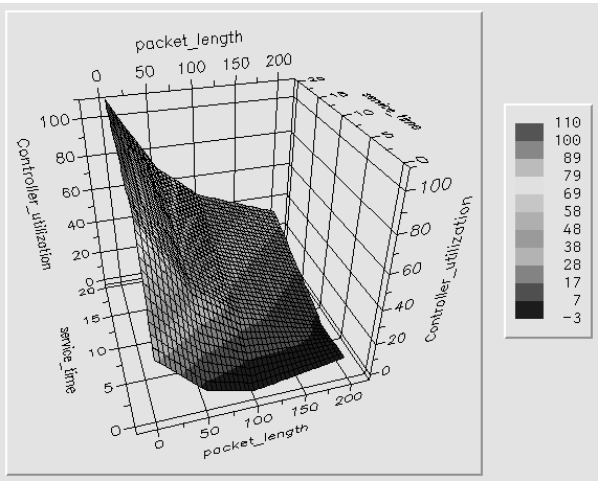

Figure 6. Utilization of a Controller for Packet Length versus Service Time

Enormous amounts of data are generated during design space exploration. For the simple experiment shown in Figure 6 we generated 100 different architecture instances by changing 2 parameters: *packet_length* and *service_time*. Thus 100 different simulation results were produced for each application. The results provide performance numbers like, for example, the *controller_utilization*. To manage these amounts of data and their consistencies, we integrated the Perl script and the retargetable simulator ORAS into the *Nelsis* design data management system [11].

9. Conclusions

We illustrated the relevance of a retargetable simulator in the context of the Y-chart. From the performance of the ORAS at 10,000 coarse-grain instructions per second we may conclude that we have been successful in building an efficient, retargetable simulator that can measure the performance of a class of dataflow architectures in a way that is functionally correct and clock-cycle accurate. We exercised the Y-chart for a kind of dataflow architecture and obtained exploration results. Based on these results, a few interesting architecture instances have been selected to be investigated at more detail.

The current ORAS simulator was constructed for a dataflow architecture template. We are currently investigating whether similar concepts can be used for more heterogeneous architecture templates.

References

[1] Bart Kienhuis, Ed Deprettere, Kees Vissers, Pieter van der Wolf. An approach for quantitative analysis of application-specific dataflow architectures. In *Proceedings of 11th Int. Conference of Applications-specific Systems, Architectures and Processors*, pages 338 – 349, 1997.

[2] D. Gajski. *Silicon Compilers*. Addison-Wesley, 1987.

[3] J. Hennessy and M. Heinrich. Hardware/software codesign of processors: Concepts and examples. In *Hardware/Software Co-Design*, pages 29 – 44. NATO ASI Series, 1996.

[4] 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*, Mar. 17-20, 1997.

[5] Joseph Buck, Soonhoi Ha, Edward A. Lee and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, Aug. 31, 1992.

[6] P. Marwedel and G. Goossens. *Code Generation for Embedded Processors*, chapter 1, pages 13 – 31. Kluwer Academic Publishers, 1995.

[7] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ programming with standard template library*. Addison-Wesley Professional Computing Series, 1996.

[8] E. K. Pauer and J. B. Prime. An architectural trade capability using the Ptolemy kernel. On Ptolemy Site http://ptolemy.eecs.berkeley.edu, Jan. 29, 1997.

[9] F. Rose and J. Shackleton. Performance modeling of system architectures. *VLSI Signal Processing*, 15(1/2):97 – 110, Jan 1997.

[10] B. W. J. Sanjaya Kumar, Jamer H. Aylor and W. A. Wulf. Object-oriented techniques in hardware design. *Computer*, 27(6):64–70, June 1994.

[11] P. van der Wolf et al. Design flow management in the Nelsis cad framework. In *Proceedings IEEE 28th Design Automation Conference*, 1991.

[12] A. J. van Gemund. Performance Prediction of Parallel Processing Systems: The PAMELA Methodology. In *Proc. 7th ACM Int. Conference on Super computing*, pages 318–327, July 19-23, 1993.

[13] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., March 1992.