A Methodology to Design Programmable Embedded Systems

The Y-chart approach

Bart Kienhuis[1], Ed F. Deprettere[1], Pieter van der Wolf[2], Kees Vissers[3,4]

[1] Leiden Institute of Advanced Computer Science, Leiden, The Netherlands
[2] Philips Research, Eindhoven, The Netherlands
[3] TriMedia Technologies, Inc., Milpitas, USA
[4] University of California at Berkeley, USA

Abstract. Embedded systems architectures are increasingly becoming *programmable*, which means that an architecture can execute a set of applications instead of only one. This makes these systems cost-effective, as the same resources can be reused for another application by reprogramming the system. To design these programmable architectures, we present in this article a number of concepts of which one is the Y-chart approach. These concepts allow designers to perform a systematic exploration of the design space of architectures. Since this design space may be huge, it is narrowed down in a number of steps. The concepts presented in this article provide a methodology in which architectures can be obtained that satisfies a set of constraints while establishing enough flexibility to support a given set of applications.

Key words:Y-chart approach, Architecture Template, Stack of Y-charts, Design Space Exploration, Abstraction Pyramid, Embedded Systems

To be published in the LNCS series vol. 2268, pag 18 – 37
by Springer-Verlag (c) 2001.
Booktitle: SAMOS: Systems, Architectures, Modeling, and Simulation
Editors: Ed F. Deprettere, Jürgen Teich, Stamatis Vassiliadis

1 Introduction

The increasing digitalization of information in text, speech, video, audio and graphics has resulted in a whole new variety of digital signal processing (DSP) applications like compression and decompression, encryption, and all kinds of quality improvements. A prerequisite for making these applications available to the consumer market is the complete embedding of the systems onto a single chip that is realized in a cost effective way into silicon. This leads to a demand for embedded systems architectures that are increasingly *programmable*

i.e., architectures that can execute a set of applications instead of only one specific application. By reprogramming the architecture, they can execute other applications with the same resources, which makes these programmable systems cost-effective.

An example of a programmable embedded system is given in Figure 1. It is a high-performance digital signal processing system that should eventually find its way into high-end TV-sets or set-top boxes [1]. The architecture consists of one or more programmable processors (both CPUs and DSPs), some programmable interconnect, a number of dedicated hardware accelerators (also called processing elements) and memory, all on a single chip.

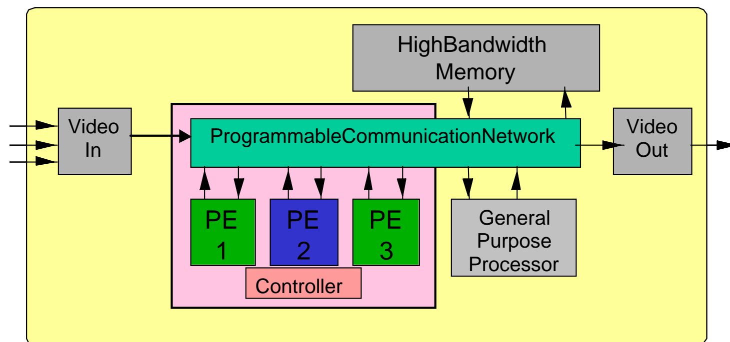

Fig. 1. High-performance digital signal processing system.

The system could be designed by specifying the architecture at a very detailed level using hardware description languages like VHDL, or Verilog, an approach called the *golden point design* [2]. A consequence of this approach is that designers work with very detailed descriptions of architectures. The level of detail involved limits the design space of the architectures that designers can explore, which gives them little freedom to make trade-offs between programmability, utilization of resources, and silicon area. Because designers cannot make these trade-offs, designs end up underutilizing their resources and silicon area and are thus unnecessarily expensive, or worse, they cannot satisfy the imposed design objectives.

Hardware/software codesign [3] is another approach to design the architecture. This design methodology uses a *refinement* approach in which one application description is refined in a number of steps into an architecture. This refinement approach has proven to be very effective for implementing a single algorithm into hardware. The approach is, however, less effective for a set of applications although a first attempt has been addressed in [4]. In general, the refinement approach lacks the ability to deal effectively with making trade-offs in favor of the set of applications.

Yet another design methodology is to assume that the architecture shown in Figure 1, does not represent a single instance, but rather a parameterized description of an architecture; it is an *architecture template* [1]. The architecture template establishes how the various elements should communicate and what the overall structure should look like. The number of processing elements to use, the kind of functionally the processing elements provide etc. is still open. Only by selecting values for all the parameters is a particular architecture instance created. Based on results obtained in general purpose processor design (GPP) [7], in particular RISC based architectures, we believe that using a template architecture and exploring this template on the basis of quantitative data is a good approach to design embedded system architectures that are programmable.

Designing architecture instances from an architecture template imposes new design challenges. Suppose a designer needs to design an architecture for a high-end TV set, given the template shown in Figure 1. Some of the design choices are: what the processing elements (PEs) should look like, what kind of control strategy should be used in the controller of the PEs, and what kind of general purpose processor should be used. Also, these choices need to be made while a number of constraints need to be satisfied, like throughput, silicon cost, silicon efficiency, and power dissipation, all for a set of applications. In this article we will present a design methodology, which is based on the Y-chart approach, which can help designers to explore the design space of the architecture template in a systematic way, to design programmable embedded systems that are programmable and satisfy the design constraints.

The Y-chart approach presented in this article is in itself not new. It has been introduced for the first time in [8]. However, this article is the first time that we present the full methodology for designing programmable embedded systems. In this article, we will present a number of *concepts* that are part of the design methodology. The concepts include, for example, the Y-chart approach, but also design space exploration, stacks of y-charts, mapping, and the abstraction pyramid.

This article is organized as follows. We start by introducing the Y-chart approach in section 2. In Section 3, we explain how to perform design space exploration using the Y-chart. In Section 4, we explain at which level of abstraction a Y-chart needs to be constructed, leading to a particular design methodology in which the design space is stepwise reduced. Mapping applications onto architecture instances is central to the Y-chart but in general very difficult. In Section 5, we propose the basic idea that can help to make the mapping process more manageable at different levels of abstraction. In Section 6, we put the Y-chart approach in context of the design of both general purpose and application-specific processors. In Section 7, we conclude this article.

---

[1] Some speak about a *platform*. We use the term architecture template

2 The Evaluation of Alternative Architectures

The problem designers face when working with an architectural template is the many architectural choices involved. In the context of the architecture template, on what basis should designers decide that one architectural choice is better than another? We somehow have to provide designers with a basis on which they can compare architectural choices in an objective way.

The ranking of architectural alternatives should be based on evaluation of performance models of architecture instances. A performance model expresses how performance metrics like utilization and throughput relate to design parameters of the architecture instance. The evaluation of performance models results in performance numbers that provide designers with *quantitative data*. This data serves as the basis on which a particular architectural choice is preferred above another architectural choice in an objective and fair manner.

We propose a general scheme with which to obtain the quantitative data, as shown in Figure 2. This scheme, which we refer to as the *Y-chart*, provides an outline for an environment in which designers can exercise architectural design and was presented for the first time in [8]. In this environment, the performance of architectures is analyzed for a given set of applications. This performance analysis provides the quantitative data that designers use to make decisions and to motivate particular choices. One should not confuse the Y-chart presented here with Gajski and Kuhn's Y-chart [9], which presents the three views and levels of abstraction in circuit design [2]. We used the term "Y-chart" for the scheme shown in Figure 2 for the first time in [5]. A similar design approach was described independently of this work in [10].

We described the Y-chart approach concept as

Concept 1. *The* Y-chart Approach *is a methodology to provide designers with quantitative data obtained by analyzing the performance of architectures for a given set of applications.*

The Y-chart approach involves the following. Designers describe a particular architecture instance (*Architecture Instance* box) and use performance analysis (*Performance Analysis* box) to construct a performance model of this architecture instance. This performance model is evaluated for the mapped set of applications (*Mapping* box and stack of *Applications* boxes). This yields performance numbers (*Performance Numbers* box) that designers interpret so that they can propose improvements, i.e., other parameter values, resulting in another architecture instance (this interpretation process is indicated in Figure 2 by the lightbulb). This procedure can be repeated in an iterative way until a satisfactory architecture for the complete set of applications is found. The fact that the performance numbers are given not merely for one application, but for

[2] In Gajski and Kuhn's Y-chart, each axis represents a view of a model: *behavioral*, *structural*, or *physical* view. Moving down an axis represents moving down in level of abstraction, from the *architectural* level to the *logical* level to, finally, the *geometrical* level.

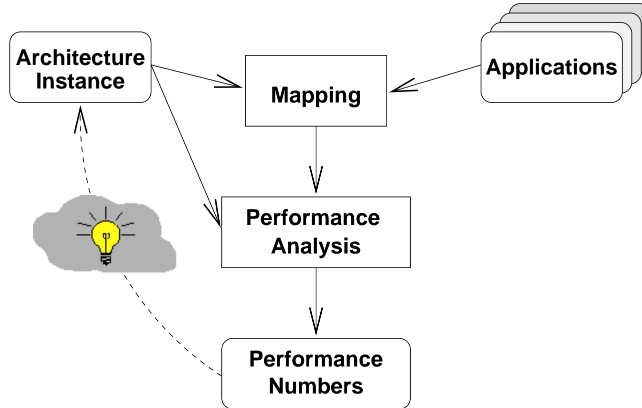

Fig. 2. The Y-chart approach.

the whole set of applications is pivotal to obtaining architecture instances that are able to execute a set of applications and obey set-wide design objectives.

It is important to notice that the Y-chart approach clearly identifies three core issues that play a role in finding feasible programmable architectures, i.e., architecture, mapping, and applications. Be it individually or combined, all three issues have a profound influence on the performance of a design. Besides designing a better architecture, a better performance can also be achieved for a programmable architecture by changing the way the applications are described, or the way a mapping is performed. These processes can also be represented by means of lightbulbs and instead of pointing an arrow with a lightbulb only to the architecture, we also point arrows with lightbulbs back to the applications and the mapping, as shown in Figure 3. Nevertheless, the emphasis is on the process represented by the arrow pointing back to the architecture instance box.

Finally, we remark that the Y-chart approach leads to highly *tuned* architectures. By changing the set of applications, an architecture can be made very general or the opposite, very specific. Hence, it is the set of applications that determines the level of flexibility required by the architecture.

3 Design Space Exploration Using the Y-chart Approach

The Y-chart approach provides a scheme allowing designers to compare architectural instances based on quantitative data. Using an architecture template [5, 6], we can produce a set of architecture instances: we systematically select for all parameters p in the parameter set P of the architecture template AT distinct values within the allowed range of values of each parameter. Consequently, we obtain a (large) finite set $\mathcal{I}$ of points I.

$$\mathcal{I} = \{I_0, I_1, ..., I_n\} \tag{1}$$

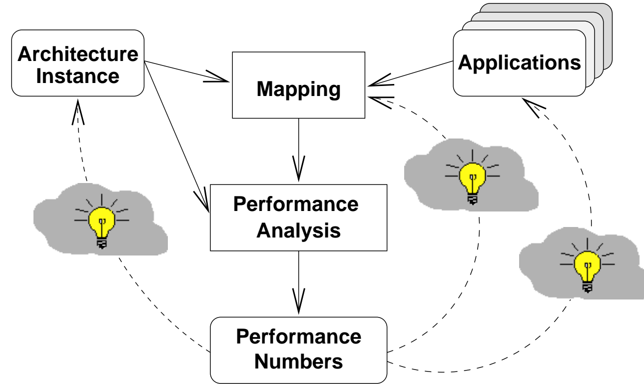

Fig. 3. The Y-chart with lightbulbs indicating the three areas that influence performance of programmable architectures.

Each point I leads to an architecture instance. Using the Y-chart, we map on each and every architecture instance the whole set of applications and measure the performance using particular performance metrics, a process we repeat until we have evaluated all architecture instances resulting from the set $\mathcal{I}$. Because the design space of the architecture template AT is defined by the set of parameters of AT, in the process described above we explore the design space D of AT.

Concept 2. *The* exploration *of the design space D of the architecture template AT is the systematic selection of a value for all parameters $P_j \in D$ such that a finite set of points $\mathcal{I} = \{I_0, I_1, \ldots I_n\}$ is obtained. Each point I leads to an architecture instance for which performance numbers are obtained using the Y-chart approach.*

When we plot the obtained parameter numbers for each architecture instance versus the set of systematically changed parameter values, we obtain graphs such as shown in Figure 4. Designers can use these graphs to balance architectural choices to find a feasible design.

Some remarks are in order in relation to Figure 4. Whereas the figure shows only one parameter, the architecture template contains many parameters. Finding the right trade-off is a multi-dimensional problem. The more parameters involved, the more difficult it will be. Note also that the curve shown in the graph is smooth. In general, designers cannot assume that curves are smooth because the interaction between architecture and applications can be very capricious. Finally, the curve in the figure shows a continuous line, whereas the performance numbers are found only for distinct parameter values. Simple curve fitting might give the wrong impression.

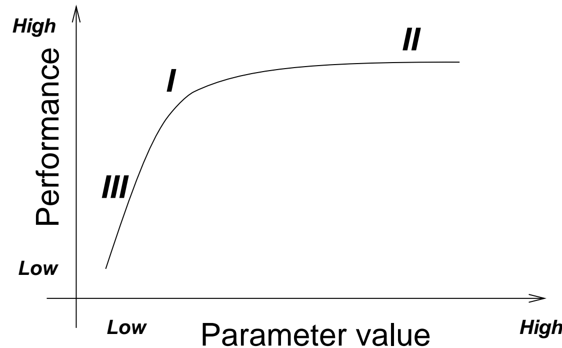

Fig. 4. The relationship between a measured performance in the architecture and a range of parameter values. Point I indicates the best trade-off between a particular performance and parameter values. Point II shows a marginal increase of the performance at a large cost and point III shows a deterioration of the performance.

4 Levels of Abstraction

We have not said anything yet about the abstraction level at which a Y-chart should be constructed. If, however, we observe the Y-chart, it is the performance analysis that determines at what level the Y-chart should be constructed. Within performance analysis, there are interesting trade-offs to be made.

Performance analysis always involves three issues: a *modeling effort*, an *evaluation effort* and the *accuracy* of the obtained results [11, 12]. Performance analysis can take place at different levels of detail, depending on the trade-offs that are made between these three issues. Very accurate performance numbers can be achieved, but at the expense of a lot of detailed modeling and long evaluation times. On the other hand, performance numbers can be achieved in a short time with modest effort for modeling but at the expense of loss of accuracy. We place the important relations between these three issues in perspective in a concept that we call the *Abstraction Pyramid*.

Concept 3. *The* Abstraction Pyramid *puts the trade-off present in performance modeling between modeling effort, evaluation effort, and accuracy in perspective of system level design.*

4.1 The Abstraction Pyramid

The abstraction pyramid (see Figure 5) describes the modeling of architectures at different levels of abstraction in relation to the three issues in performance modeling. At the top of the pyramid is a designer's initial rough idea (shown as a lightbulb) for an architecture in the form of a 'paper architecture'. The designer wants to realize this architecture in silicon. The bottom of the pyramid represents all possible feasible realizations; it thus represents the complete design space of

the designer's paper architecture. A discussion of the three main elements of the abstraction pyramid follows including two additional elements: the opportunity to change models and the different abstraction levels that can be found in system level design.

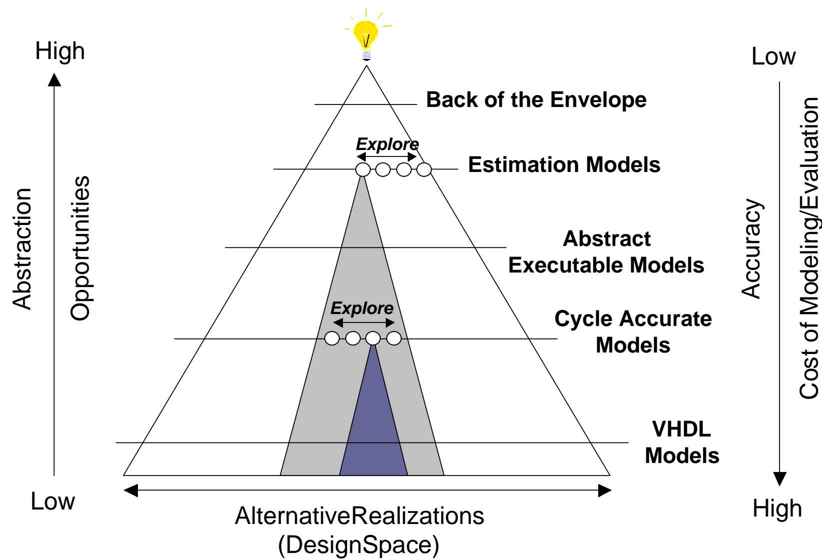

Fig. 5. The abstraction pyramid represents the trade-off between modeling effort, evaluation speed, and accuracy, the three elements involved in a performance analysis.

Cost of Modeling Moving down in the pyramid from top to bottom, a designer defines an increasing expenditure of detail of an architecture using some modeling formalism. This process proceeds at the cost of an increasing amount of effort, as indicated on the *cost of modeling* axis at the right-hand side of the pyramid. As architectures are described in more detail, the number of architectural choices (i.e. the number of parameters in the architecture template) increases, expanding the basis of the pyramid. Each new architectural choice, albeit at a lower level of detail, thus further broadens the design space of the architecture.

Opportunity to Change As the designer moves down and includes more detail using a modeling formalism, the architecture becomes increasingly more specific. Simply because more design choices have been made, it becomes more costly to redo these design choices if another architecture is to be considered. Hence the opportunity to explore other architectures diminishes. This is indicated on the *opportunities* axis at the left-hand side of the pyramid.

Level of Detail Lines intersecting the abstraction pyramid horizontally at different heights represent different abstraction levels found in system level design. The small circles on such line represent architecture instances modeled at that abstraction level. At the highest level of abstraction, architectures are modeled using *back-of-the-envelope models*. Models become more detailed as the abstraction pyramid is descended. The *back-of-the-envelope models* is followed by *estimation models*, *abstract executable models*, *cycle-accurate models*, and, finally, by *synthesizable VHDL models*. This represents the lowest level at which designers can model architectures.

We use the term *back-of-the-envelope model* for simple mathematical relationships describing performance metrics of an architecture instance under simple assumptions related to utilization and data rates (e.g. [7]). Estimation models are more elaborated and sophisticated mathematical relationships to describe performance metrics (e.g. [12]). Neither model describes the correct functional behavior or timing. The term *abstract executable model* describes the correct functional behavior of applications and architectures, without describing the behavior related to time (e.g. [13]). The term *cycle-accurate model* describes the correct functional behavior and timing of an architecture instance in which a cycle is a multiple (including a multiple of one), of a clock cycle (e.g. [14, 15]). Finally, the term *synthesizable VHDL model* describes an architecture instance in such detail, in both behavior and timing, that the model can be realized in silicon.

Accuracy In the abstraction pyramid, accuracy is represented by the gray triangles. Because the accuracy of cycle-accurate models is higher than the accuracy of estimation models, the base of the triangle belonging to the cycle-accurate models is smaller than the base of the triangle belonging to the estimation models. Thus the broader the base, the less specific the statement a designer can make in general about the final realization of an architecture.

Cost of Evaluation Techniques to evaluate architectures to obtain performance numbers range from back-of-the-envelope models where analytical equations are solved symbolically, using, for example, *Mathematica* or *Matlab*, up to the point of simulating the behavior in synthesizable VHDL models accurately with respect to clock cycles. In *simulation*, the processes that would happen inside a real architecture instance are imitated. Solving equations only takes a few seconds, whereas simulating detailed VHDL models may take hours if not days. The axis at the right-hand side represents both cost of modeling and *cost of evaluation*.

4.2 Exploration

The abstraction pyramid shows the trade-offs in performance analysis. When exploring the design space of an architecture template, designers should make different trade-offs at different times. Higher up in the pyramid they can explore

a larger part of the design space in a given time. Although it is less accurate, it helps them to narrow down the design space. Moving down in the pyramid, the design space that they can consider becomes smaller. The designer can explore with increased accuracy only at the expense of taking longer to construct, evaluate, and change models of architecture instances.

The process of exploration and narrowing down on the design space is illustrated in the abstraction pyramid by the circles. These circles are drawn at the level of estimation models and at the level of cycle-accurate models. Each circle represents the evaluation of a particular architecture instance. An exploration at a particular abstraction level is thus represented as a set of circles on a particular horizontal line in the abstraction pyramid.

4.3 Stacks of Y-chart Environments

Due to the level-dependent trade-off between modeling, evaluation, and accuracy, designers should use different models at different levels of abstraction when exploring the design space of architectures. The Y-chart approach used at these different levels is, however, invariant: it still consists of the same elements, as shown in Figure 2. This leads to the following concept of a *Y-chart environment*:

Concept 4. *A* Y-chart Environment *is a realization of the Y-chart approach for a specific design project at a particular level of abstraction.*

The different levels represented in the abstraction pyramid thus indicate that more than one Y-chart environment is needed in a design process for architectures. Therefore, different Y-chart environments are needed at different levels of abstraction, forming a stack as illustrated in Figure 6. This figure shows three possible Y-chart environments: one at the level of *back of the envelope* models, one at the level of *cycle accurate* models, and on at the level of *VHDL* models.

In the abstraction pyramid, more than these three levels of abstraction are given. Nevertheless, we will resort to just three levels from the abstraction pyramid in Figure 5

Back-Of-The-Envelope Early in the design, designers make use of back-of-the-envelope models and estimation models, to model architecture instances. This allows them to construct many architecture instances very quickly. Designers typically employ generic tools like *Matlab* or *Mathematica* to evaluate the performance of these models by solving analytic equations. These tools can compute complex equations (symbolically) within a few seconds. The resulting performance numbers typically represent rough estimates for throughput, latency, and utilization. The tools evaluate the performance metrics either numerically or symbolically.

Cycle-Accurate As the design space for the architecture template narrows, designers use abstract-executable models and cycle-accurate models to describe

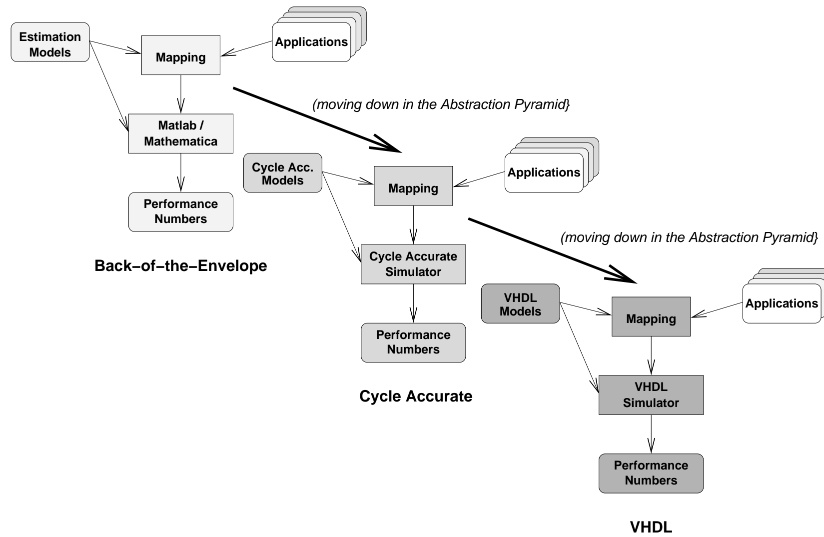

Fig. 6. A stack of Y-chart environments, with a different model at each level of abstraction.

architecture instances. At this level of abstraction, designers can compare the performances of moderately different architectures. Models at this level require architecture simulators that typically run from minutes to hours to carry out a simulation. These simulators most likely employ discrete-event mechanisms. The performance numbers at this level typically represent values for throughput, latency, and utilization rates for individual elements of architecture instances. As the models become more accurate, the accuracy of the performance numbers also becomes higher.

VHDL Finally, as the design space narrows down further, a designer wants to be able to compare the performance of slightly different architecture instances accurately to within a few percent. The designer uses detailed VHDL models to describe architecture instances, taking significant amounts of time and resources. Designers can carry out the simulations using standard VHDL simulators. Simulation time required for these architecture instances can be as much as several days. The obtained performance numbers are accurate enough that a designer can compare differences in the performance of architecture instances to within a few percent.

4.4 Design Trajectory

The abstraction pyramid presents trade-offs between modeling, evaluation, and accuracy that result in a stack of Y-chart environments being used. This stack leads to a *design trajectory* in which designers can model architectures and applications at various levels of detail. The Y-chart approach and the stack of Y-chart

environments thus structure the design process of programmable embedded architectures.

Concept 5. *A* Stack of Y-charts *describes a design trajectory in which different Y-chart environments are realized at different levels of abstraction, each taking a different trade-off position in the Abstraction Pyramid, which leads to a stepwise refinement of the design space of programmable embedded architectures.*

Within this design trajectory, designers perform design space exploration at each level and narrow down the design space containing feasible designs. This approach differs from the golden point design, which is the design methodology currently used in the design of complex programmable architectures. Here a design, the golden point, is modeled directly at low level, i.e., VHDL.

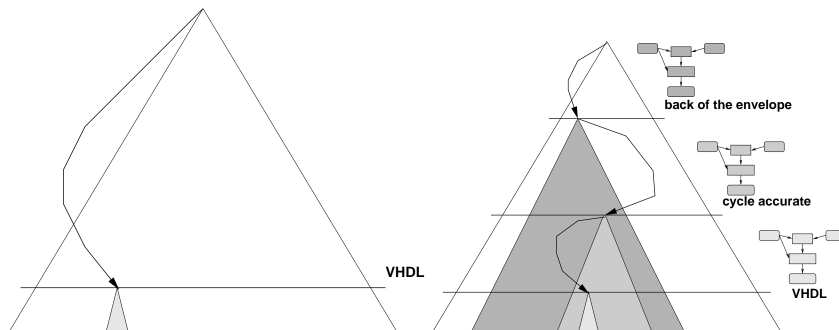

Fig. 7. (a), the golden point design approach. (b), the design approach in which designers use Y-chart environments.

In Figure 7(a), we show the golden point design. Here the golden point is modeled directly at a low level in the pyramid. Because hardly any exploration and validation of ideas took place, except for paper exercises and spreadsheets, it is first of all very much the question whether the selected point results in a feasible design. Secondly, due to the low level of detail already involved, it becomes very difficult to explore other parts of the design space, thus leading to sub optimal design. Thirdly, it is very likely that designers will be confronted with unpleasant surprises at late stages in the design process. This can lead to costly rework and slipping time schedules. In Figure 7(b), the design approach is shown in which designers use Y-charts at different levels of abstraction. This approach, we believe, leads to better-engineered architectures and moreover reduces risk. Each time more resources are committed to the design of an architecture, more knowledge is available to assess if a feasible design can be accomplished within its given set of constraints.

5 Mapping

Mapping pertains to conditioning a programmable architecture instance such that it executes a particular application. It leads to a program that causes the execution of one application on the programmable architecture. Mapping involves, for example, assigning application functions to processing elements that can execute these functions. It also involves mapping the communication that takes place in applications onto communication structures.

Mapping is a difficult problem, but it is essential to the Y-chart approach. To be able to do mapping, at various levels of abstraction, and to develop a mapping strategy concurrently with the development of the architecture, we now discuss the basic concept we use to make mapping as simple as possible. This mapping concept is also depicted in Figure 8.

Concept 6. *In mapping, in context of the Y-chart, we say that a* natural fit *exists if the model of computation used to specify applications matches the model of architecture used to specify architectures and that the data types used in both models are similar.*

To explain the matching of the model of computation with the model of architecture, we first explain what we means by these terms. Then we look at the data types found in both applications and architectures and come back to the notion of the natural fit.

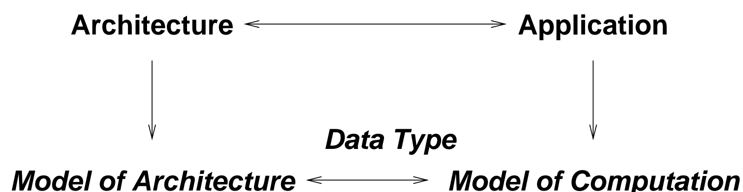

Fig. 8. A smooth mapping from an application to an architecture only takes place if the model of computation matches with the model of architecture and when the same data types are used in both models.

Model of Computation Applications are specified using some kind of formalism that has an underlying model of computation. We define a model of computation, inspired by [16], as a formal representation of the operational semantics of networks of functional blocks describing computations.

So, the operational semantics of a model of computation governs how the functional blocks interact with one another realizing computations. Many different models of computation already exist that have specific properties. Different models have proven to be very effective in describing applications in various application domains [17]. Some examples of well-known models of computation are:

Dataflow Models, *Process Models*, *Finite State Machine Models*, and *Imperative Models.*

Model of Architecture In analogy with a model of computation, we define the concept of model of architecture as a formal representation of the operational semantics of networks of functional blocks describing architectures.

In this case, the functional blocks describe the behavior of architectural resources like CPUs, busses, and memory and the operational semantics of a model of architecture governs how these resources interact with one another. Although models of architecture are less mature then models of computation, one can identify characteristics like whether control is centralized or distributed, or whether there is an emphasis on control flow or data flow.

Data Types In both applications and architectures, data that is exchanged is organized in a particular way and has particular properties. A *data type* describes these properties. Examples of simple data types are integers, floats, or reals. More complex data types are streams of integers or matrices.

To realize a smooth mapping, the types used in the applications should match with the types used in the architecture. If architectures support only streams of integers, the applications should also use only streams of integers. Suppose an application uses only matrices whereas an architecture instance on which we want to map uses only streams of scalars. Because the types do not match, we can already say that we first have to express the matrices in terms of streams of scalars. A stream of scalars, however, has very different properties from a matrix (e.g. a matrix is randomly accessible), having a profound influence on how the application executes on the architecture. Consequently, to obtain a smooth mapping of applications onto architectures, the data types in both the applications and the architectures should be the same or at least the architecture should support a more fine-grained data types then the application data types.

5.1 Natural Fit

Given an application that is described using a model of computation and an architecture instance that is described using a model of architecture, when we say the application fits *naturally* onto the architecture instance, we mean that:

1. The architecture instance provides at least primitives similar to those used in the application. In other words, the grain-size of functions and processing elements should be the same. For example, a FIR filter functions used in the application should also be found for example as a FIR processing element in the architecture instance.
2. The operational semantics of the architecture instance at least matches the operational semantics of the application. For example, if functions in the applications behave in a data-driven manner then the processing elements should also operate in a data-driven manner.

3. The data types used in applications should match the data types available on the architecture instance. For example, when an application uses only streams of samples then the architecture instance should at least provide supports for such streams of samples.

Examples of this 'natural fit' principle can be found in literature. For example, in [18], the CSP model of computation is used for describing applications that are mapped onto asynchronous hardware. In [19], the cyclo-static dataflow (CSDF) is used for describing applications that are mapped on one or more VSP processors, which are real-time programmable video processors. The most well know example of the presented principle is the imperative C-language that is mapped onto a micro-processor [7].

Other examples of the 'natural fit' at higher levels of abstraction, can be found in the ORAS work [5, 6] and SPADE work [13]. In both cases, process networks are mapped onto abstract models of stream-oriented architectures. Another example is the POLIS work [10]. It maps a special kind of Finite State Machines (FSMs) onto abstract models of microprocessor architectures.

6 Processor Design

In the introduction, we described the problem of finding the correct parameters to instantiate an architecture instance that satisfies a number of constraints for a given set of applications. This problem has already been researched for decades in the realm of general-purpose processor (GPP) design. In this domain, complex architectures called *instruction-set processors* or *microprocessors* are designed that execute a word-processor application as easily as a spreadsheet application or even simulate some complex physical phenomenon. Therefore, designers working in this domain know what programmability implies in terms of (complex) trade-offs between hardware, software, and compilers. In this section, we will show that the design of GPPs fits into our Y-chart approach.

6.1 Design of General-Purpose Processors

In the beginning of the 1980s, revolutionary GPPs emerged that were called RISC microprocessors [20]. These processors were developed in a revolutionary way; namely, designers used extensive quantitative analysis of a suite of *benchmarks*, which is a set of applications. As a result, these architectures were smaller, faster, cheaper and easier to program than conventional architectures of that time. With the advent of the RISC microprocessors, the design of GPPs in general began to swing away from focusing purely on hardware design. Designers started to focus more on the quantitative analysis of difficulties encountered in architecture, mapping (or compiling), and the way benchmarks are written. This quantitative approach has become the de-facto development technique for the design of general-purpose processors [21, 7].

As we will show, we can in retrospect cast the design of general-purpose processor architectures in terms of our Y-chart approach as presented in this article.

We do this first by looking at the design of the MIPS R4000 microprocessor [22] as depicted in Figure 9. In this Y-chart, the set of applications is described in the C-language by the *SPECmark* programs. Using a C-compiler, tuned especially to reflect the R4000 architecture, and a special architecture simulator called *Pixie*, Hennessy et al., evaluated the performance of the applications mapped onto an instance of the R4000. The *Pixstat* program interprets the produced performance numbers. In the given Y-chart, the dashed box represents the fact that the architecture is not specified as a separate entity, but that it is hard coded into the GNU GCC compiler and architecture simulator Pixie.

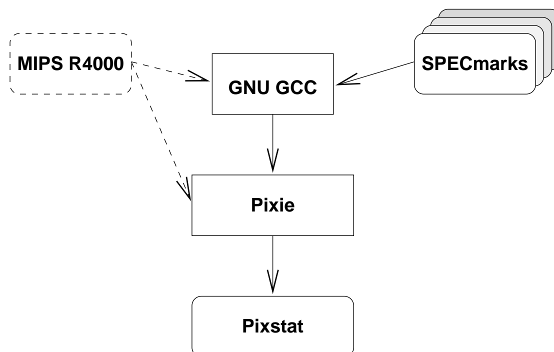

Fig. 9. The Y-chart used in the construction of the MIPS R4000.

According to Hennessy et al., the design space of the MIPS R4000 is extremely large and evaluating alternatives is costly for three reasons: construction of accurate performance models, long simulation runs, and tuning of the compiler to include architectural changes. Precisely these three issues are expressed in the Abstraction Pyramid in figure 5.

To narrow down the design space of the R4000, in a stepwise fashion as shown in Figure 7(b), four different simulators are used at increasing levels of detail, as shown in Table 1. One can clearly see that the simulation speed drops dramatically as more detail is added. Interestingly, Hennessy et al. consider the top two levels of simulation to be the most critical levels in the design of processors. It allowed for the exploration of a large part of the design space of the MIPS R4000, helping the designers to make better trade-offs.

6.2 Design of Application-Specific Processors

Next, we show three processor designs that we have put in context of the Y-chart methodology. In these Y-charts, the selection of benchmarks results in more application-specific processor architectures.

Wilberg et al. [23] used a Y-chart, as shown in Figure 10(a) for designing application-specific VLIW (Very Long Instruction Word) architectures for low-speed video algorithms like JPEG, H.262 and MPEG1. The video applications

Simulator	Level of Accuracy	Sim. Speed
Pixie	Instruction Set	$> 10^6$ cycles/sec
Sable	System Level	$> 10^3$ cycles/sec
RTL (C-code)	Synchronous Register Transfer	> 10 cycles/sec
Gate	Gate/Switch	< 1 cycles/sec

Table 1. Different Levels of Simulation used in the MIPS R4000 design [22].

written in C are compiled into generic RISC-instructions using the GCC/MOVE compiler developed by Corporaal et al. [24].

Sijstermans et al. [25] used a Y-chart, as shown in Figure 10(b) to design the *TriMedia* programmable multi-media processor TM1000. They compiled a set of applications that were written in C into object-code, using a commercial compiler framework. The architecture simulator *tmsim* can simulate this object-code clock-cycle accurately. Both the compiler and the simulator are *retargetable* for a class of TriMedia architectures that they describe using a Machine Description File (MDF).

Živojnović et al. [26] used a Y-chart, as shown in Figure 10(c) to develop DSP processors. As a benchmark, they used a special set of C functions called DSPstone. They mapped the benchmarks onto the retargetable simulator called *SuperSim* using a retargetable version of the GNU GCC-compiler. They described a class of DSP-architectures using the special language *LISA*.

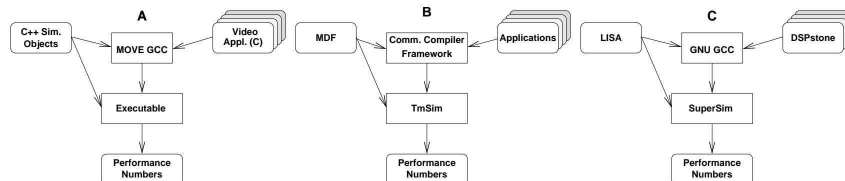

Fig. 10. Y-chart environments used in various design projects.

So, if the benchmark suite contains very different programs like a word processor application, a spreadsheet application and a compiler application, a general-purpose architecture results that is optimized for a broad range of applications. If, on the other hand, the set of applications contains only video applications, a dedicated processor architecture results that is optimized for video applications. The selection of the benchmarks is hence a key decision step in the design process.

In all the cases presented, designers make refinements to a well-known architecture template commonly referred to as *load-store architectures* [7]. For these architectures, good detailed models exist as well as good compiler frameworks. The model of computation underlying the C-language, e.g., imperative

languages, fits naturally with the model of architecture of load-store architectures. Consequently, designers of microprocessors use applications written in the C-language. To design architectures, they resort to tuning known compiler frameworks to include architectural changes, to change mapping strategies, or to rewrite applications.

7 Conclusions

We believe that programmable embedded systems will more and more be designed by means of an architecture template for which designers needs to select the proper set of parameter values. We assume that different architecture templates will emerge for different domains like mobile communication, automotive, and high-performance signal processing. The domain specific templates will become the models of architecture that match the typical characteristics of the applications that execute within those domains.

As a design approach for programmable embedded systems, we presented the Y-chart approach. It is a methodology in which designers use quantitative data that provides them with a sound basis on which to make decisions and motivate particular design choices. This leads to an environment in which a systematic exploration of the design space of the architecture template can be performed, leading to solid engineered systems. Since the design space of an architecture template is in general huge, a stepwise refinement of the design space is needed. This leads to the concept of a stack of Y-chart environments, in which each Y-chart models the system at a different level of abstraction.

In the design of programmable embedded systems, the current practice is still the golden point design (See Figure 7(a)). Quantifying architectural choices in architectures is unfortunately by no means current practice. Yet the RISC architecture development has unmistakably shown that quantifying design choices leads to well engineered architectures. As shown in this article, the design methodology of RISC processors and other application-specific processors can be cast into the presented Y-chart approach.

We believe, that the Y-chart approach presents a general and solid methodology for designing programmable embedded system architectures. This is reinforced by the fact that the methodology has already effected other research in embedded system design [27–29]. Furthermore, at Philips research, the Y-chart approach has led to the development of YAPI [30], which stands for *Y-chart application programming interface*. In addition, the Y-chart approach has influenced, and has been influenced by, the system level design work described in [31].

Before we can fully perform design space exploration at various levels of abstraction, there are still some tough research issues to be tackled. Especially performing mappings at high abstraction levels is difficult. For some dedicated systems, we can relay on applications written in 'C', compilers and architecture models developed in the realm of general-purpose processors to construct Y-chart environments. Nonetheless, we believe that new systems on a chip require

new architectural models, new application models, and mapping techniques. The new systems will for example exploit both task-level parallelism and instruction level parallelism. Furthermore, they should operate under real-time constraints and use heterogeneous architectural components (e.g. CPUs, DSPs, dedicated co-processors, distributed memories). The notion of models of computation and models of architectures should help us to pave the way to the exploration of systems at higher levels of abstraction.

Acknowledgement.

This research has been performed at both Philips Research and Delft University of Technology, as part of their “cluster program”. Philips Research, Ministry of Economic affairs, and Delft University of Technology have supported this research and are hereby acknowledged.

References

1. Claasen, T.: Technical and industrial challenges for signal processing in consumer electronics: A case study on TV applications. In: Proceedings of VLSI Signal Processing, VI. (1993) 3 – 11
2. Richards, M.A.: The rapid prototyping of application specific signal processors (RASSP) program: Overview and status. In: 5th International Workshop on Rapid System Prototyping, IEEE Computer Society Press (1994) 1–6
3. De Micheli, G., Sami, M.: Hardware/Software Co-Design. Volume 310 of Series E: Applied Sciences. NATO ASI Series (1996)
4. Kalavade, A., Subrahmanyam, P.: Hardware/software partioning for multi-function systems. In: Proc. of ICCAD'97. (1997) 516 – 521
5. Kienhuis, B., Deprettere, E., Vissers, K., van der Wolf, P.: The construction of a retargetable simulator for an architecture template. In: Proceedings of 6th Int. Workshop on Hardware/Software Codesign, Seattle, Washington (1998)
6. Kienhuis, B.A.: Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools. PhD thesis, Delft University of Technology, The Netherlands (1999)
7. Hennessy, J.L., Patterson, D.A.: Computer Architectures: A Quantitative Approach. second edn. Morgan Kaufmann Publishers, Inc. (1996)
8. Kienhuis, B., Deprettere, E., Vissers, K., van der Wolf, P.: An approach for quantitative analysis of application-specific dataflow architectures. In: Proceedings of 11th Int. Conference of Applications-specific Systems, Architectures and Processors (ASAP'97), Zurich, Switzerland (1997) 338 – 349
9. Gajski, D.: Silicon Compilers. Addison-Wesley (1987)
10. Balarin, F. abd Giusto, P., Jurecska, A., Passerone, C., Sentovich, E., Tabbara, B., Chiodo, M., Hsieh, H., Lavagno, L., Sangiovanni-Vincentelli, A.L., Suzuki, K.: Hardware-Software Co-Design of Embedded Systems: The POLIS Approach. Kluwer Academic Publishers (1997)
11. Lavenberg, S.S.: Computer Performance Modeling Handbook. Acadamic Press (1983)

12. van Gemund, A.J.: Performance Modeling of Parallel Systems. PhD thesis, Laboratory of Computer Architecture and Digital Techniques, Delft University of Technology (1996)
13. Lieverse, P., van der Wolf, P., Vissers, K., Deprettere, E.F.: A methodology for architecture exploration of heterogeneous signal processing systems. Journal of VLSI Signal Processing for Signal, Image and Video Technology **29** (2001) 197 – 207
14. Kruijtzer, W.: Tss: Tool for system simulation. IST Newsletter, Philips Research Laboratories (1997) 5 – 7 Issue 17.
15. Liao, S., Tjiang, S., Gupta, R.: An efficient implementation of reactivity for modeling hardware in the scenic design environment. In: Proceedings of DAC-97. (1997)
16. Lee, E.A., et al.: An overview of the Ptolemy project. Technical report, University of California at Berkeley (1994)
17. Chang, W.T., Ha, S., Lee, E.A.: Heterogeneous simulation - mixing discrete-event models with dataflow. VLSI Signal Processing **15** (1997) 127 – 144
18. van Berkel, K.: Handshake Circuits: an asynchronous architecture for VLSI programming,. Cambridge University Press (1993)
19. Vissers, K., Essink, G., van Gerwen, P., Janssen, P., Popp, O., Riddersma, E., Veendrick, J.: Architecture and programming of two generations video signal processors. In: Algorithms and Parallel VLSI Architectures III. Elsevier (1995) 167 – 178
20. Patterson, D.: Reduced instruction set computers. Comm. ACM **28** (1985) 8 – 21
21. Bose, P., Conte, T.M.: Performance analysis and its impact on design. IEEE Computer **31** (1998) 41 – 49
22. Hennessy, J., Heinrich, M.: Hardware/software codesign of processors: Concepts and examples. In Micheli, G.D., Sami, M., eds.: Hardware/Software Codesign. Volume 310 of Series E: Applied Sciences. NATO ASI Series (1996) 29 – 44
23. Camposano, R., Wilberg, J.: Embedded system design. Design Automation for Embedded Systems **1** (1996) 5 – 50
24. Corporaal, H., Mulder, H.: Move: A framework for high-performance processor design. In: Proceedings of Supercomputing, Albuquerque (1991) 692 – 701
25. Sijstermans, F., Pol, E., Riemens, B., Vissers, K., Rathnam, S., Slavenburg, G.: Design space exploration for future trimedia CPUs. In: ICASSP'98. (1998)
26. Živojnović, V., Pees, S., Schläger, C., Willems, M., Schoenen, R., Meyr, H.: DSP Processor/Compiler Co-Design: A Quantitative Approach. In: Proc. ISSS. (1996)
27. Rabaey, J., Potkonjak, M., Koushanfar, F., li, S., Tuan, T.: Challenges and opportunities in broadband and wireless communication designs. In: Proceedings of ICCAD. (2000)
28. Hekstra, G., La Hei, G., Bingley, P., Sijstermans, F.: Trimedia cpu64 design space exploration. In: ICCD. (1999)
29. Marculescu, R., Nandi, A.: Probabilistic application modeling for system-level performance analysis. In: Proceedings Design, Automation and Test in Europe (DATE'01), Munich, Germany (2001) 190–196
30. de Kock, E., Essink, G., Smits, W., van der Wolf, P., Brunel, J., Kruijtzer, W., Lieverse, P., Vissers, K.: Yapi: Application modeling for signal processing systems (2000)
31. Keutzer, K., Malik, S., Newton, A.R., Rabaey, J.M., Sangiovanni-Vincentelli, A.: System-level design: Orthogonalization of concerns and platform-based design. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **19** (2000) 1523–1543