

# A Method for Partitioning UNITY Language in Hardware and Software

E. Barros

Depto de Informatica-CCEN-UFPE  
Caixa Postal 7851  
Cidade Universitaria - Recife - PE  
CEP 50732-970 - BRAZIL  
ensb@di.ufpe.br

W. Rosenstiel, X. Xiong

Forschungszentrum Informatik  
Haid-und-Neu-Straße 10 - 14  
76131 Karlsruhe  
Germany  
rosen@fzi.de, xiong@fzi.de

## Abstract

*In this paper we introduce a method to partition UNITY system specifications into software and hardware parts. This method considers different design possibilities and defines cost functions to find out the most suitable one under given design constraint in terms of hardware-software trade-off.*

## 1 Introduction

Hardware/software codesign has gained remarkable attention in recent years. A crucial point for the success of codesign is to find a proper way to partition given system specification under given design constraints into software and hardware parts.

There are two typical ways towards HW/SW partitioning. Hardware oriented one, as the one in Vulcan II from [GM92], generally starts from a hardware implementation. During the partitioning, those non-time-critical parts are often moved to software, with the given design constraints still being fulfilled. In software oriented approaches, taking the COSYMA system [EH92] for example, constructs violating the time constraints will mostly be shifted to hardware components.

Currently existing approaches mostly do not consider different design alternatives, and few of them explore the pipelining feasibility which can essentially enhance the parallelism.

In this paper we present an approach to partition UNITY. In comparison to other existing ones, it considers a large-scale implementation alternatives resulting from thorough examination of data dependency, different parallelism degrees and resource shar-

ing among individual UNITY constructs. Evaluation and final decision of alternatives are determined by a two stage clustering algorithm under given design constraints. Criteria like feasibility of pipelining, minimization of area/delay costs, reduction of synchronization/communication cost, and so on are decisive factors during the partitioning process.

## 2 UNITY for HW/SW-Codesign

UNITY denotes a theory for specifying parallel computations and a proof system. It consists of a notation for programming and a logic for reasoning about computations [CM88]. In this work it will only be used as program notation.

In comparison to other languages, UNITY provides some favorable features in terms of hardware/software-codesign. It supports specification of both synchronous and asynchronous behaviors, without having to tell how, where or when the behaviors should be executed. This grants designer independency on the target architecture. Furthermore, its simplicity makes exploration of hardware/software trade-off feasible in terms of complexity and computational cost. Finally and not lastly, its nondeterminism feature is useful for supporting the hardware representation.

### 2.1 Basic Structure

A UNITY program consists of 3 major parts, whereby declaration and initialization sections are self explanatory. The assignment section as program body is a set of multiple assignment statements with each being synchronous (linked by || sign) or asynchronous

(link by  $\parallel$  sign) to others. An execution starts from any state satisfying the initial condition, every statement is selected infinitely often (*fairness rule*) and nondeterministically executed till a state is reached where further execution will not change it (fix point).

- **Enumerate assignment:** Its general form:

$$\begin{aligned} x_1, \dots, x_n &:= e_1 1, \dots, e_1 n \text{ if } b_1 \\ &\sim \dots \\ &e_m 1, \dots, e_m n \text{ if } b_m \end{aligned}$$

Depending on the boolean conditions  $b_1, \dots, b_m$ , the right-hand expressions are evaluated and assigned synchronously to the left-hand variables. In case more than one boolean condition is true, all corresponding expression lists must have the same value to guarantee determinism of every assignment statement.

- **Parallel assignment:** It has the form  $A_1 \parallel \dots \parallel A_n$ , with each  $A_i$  being either an enumerated assignment, or a quantified assignment (defined below). During execution, all right-hand expressions are evaluated in parallel and assigned to the corresponding left-hand variables synchronously.

Following activity can either be specified with enumerated or parallel assignments: *Assign the smaller of  $A[i]$  and  $B[j]$  to  $C[k]$  and increment  $i$  by 1 if  $A[i] \leq B[j]$  and increment  $j$  otherwise*

$$\begin{aligned} C[k] &:= \min(A[i], B[j]) \\ \parallel \quad k &:= k + 1 \\ \parallel \quad i &:= i + 1 \text{ if } A[i] \leq B[j] \\ \parallel \quad j &:= j + 1 \text{ if } A[i] \geq B[j] \end{aligned} \quad (1)$$

or

$$\begin{aligned} C[k], i, j, k &:= A[i], i + 1, i, k + 1 \text{ if } A[i] > B[j] \\ &\sim B[j], i, j + 1, k + 1 \text{ if } A[i] \leq B[j] \\ &\sim B[j], i + 1, j + 1, k + 1 \text{ if } A[i] = B[j] \end{aligned} \quad (2)$$

- **Quantified assignment:** A special form of parallel assignment expressed by:  $\langle \parallel i_1, \dots, i_n : Q :: A \rangle$ . The *quantification*  $Q$  consists of definition for ranges of bound variables  $i_1, \dots, i_n$  and a boolean expression. The boolean expression may depend on values of program variables which can change during runtime as well as constants and bound variables.  $A$  may be any assignment. An instance of a quantification is a set of values of

bound variables which satisfies the boolean expression in  $Q$ . The execution of a quantified assignment is done by executing the body for each instance of bound variables in parallel.

- **Quantified statement list:** Its syntax is given by  $\langle \parallel i_1, \dots, i_n : Q :: S \rangle$ . Here  $Q$  is defined in a similar way as above. Its instance number here must however not vary during runtime. The body  $S$  represents a single or a list of statements. Quantified statement list may also be seen as shorthand notation for a list of statements  $S_1 \parallel \dots \parallel S_k$ .

Following UNITY program piece employs both quantified assignments and quantified statement list to define an identity matrix at fixed point.

$$\begin{aligned} \langle \parallel \quad 0 \leq i \leq N :: U[i, i] &:= 1 \\ \parallel \langle \parallel j : 0 \leq j \leq N \wedge i \neq j :: U[i, j] &:= 0 \rangle \rangle \end{aligned}$$

## 2.2 Overview of Our UNITY Approach

### 2.3 The Overview

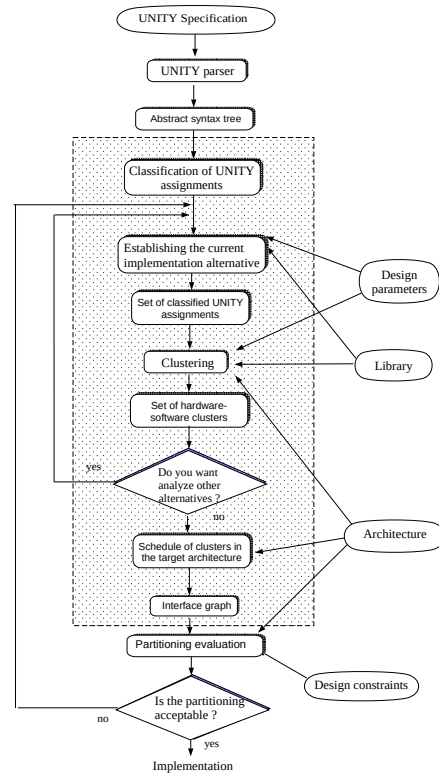


Figure 1: Overview of the UNITY approach

First the UNITY specification is classified. An implementation alternative is then selected as reference

for the clustering process. Guided by well defined optimization criteria, the two stage clustering produces hardware and software clusters. Intermittent results are evaluated by allocation of resulting clusters to the target architecture. If the resulting partitioning is unacceptable, the process will be repeated. The final result must meet the given design constraints.

## 2.4 Target Architecture

Since the style of the target architecture considerably influences performance and area behaviors, decision made for partitioning will be heavily affected by the selected scheme of target architecture. We adopt in our approach a simple model. In this model, control is centralized in CU (Central Unit), a general reprogrammable processor serving as the software processor, and the processing units (PUs) are dedicated processors (ASICs, FPGAs) used to implement the hardware clusters. The CU and PUs work on the master-slave model and communicate with each other via single memory through a common bus. In spite of its simplicity, this model, cited in many publications, can deal with a great amount of applications. A figure, which illustrates the allocation of the resulting hardware and software clusters to this architecture, can be later seen at the end of the clustering description.

## 3 Classification of UNITY

### 3.1 Definition of Partitioning Unit

Since our partitioning resides at the assignment level, we define the smallest indivisible unit in UNITY, termed as *element*, as the partitioning unit. An element is a single assignment part of an enumerated assignment with only one single right side. Equations (1), (2) contain 4 and 3 elements respectively.

### 3.2 Classification

To detect different alternatives, comprehensive study and analysis of individual element concerning its characterization (type of its affiliated assignment) and its relations to other elements (data dependency, synchronization), termed "classification", are necessary. For this purpose, for each element five attributes will be established, and discussion for each attribute about its role in the partitioning will be conducted.

**AR: Asynchronous Relationship** Elements in distinct statements are said to have this relationship. Element consuming variables (*client*) produced by oth-

ers (*server*), or consuming and producing variables (*clientserver*) are said to be *data dependent*. Though data dependent asynchronous elements can not run in parallel totally, they are good candidates for pipelining. Otherwise they will be sequentially executed and share functional unit (FU). Data independent asynchronous elements can run in parallel. The implementation alternatives result from the so called *class values* attached to each attribute. They indicate for each element its relationship with other elements, type of its assignment, and reflects in this way its implementation alternatives: parallel/pipelining (default class values) or sequential.

**SIPA: Synchronism Inside Parallel Assignment** Its class values indicate whether an element belongs to a parallel assignment. Moreover, they capture data dependency between elements within the same parallel assignment. The assignment  $x := y || y := x$  swaps the values of the variables  $x$  and  $y$ . It consists of two elements classified as both having *clientserver/sequential* class values.

Data dependency between synchronous elements is called *anti-dependency (read-before-write)*. Synchronization for such kind of "waiting for write before the common variable is read by all", is pretty expensive, if anti-dependent elements are allocated to different clusters. Thus, anti-dependent elements should possibly be clustered together.

**SIQA Synchronism Inside Quantified Assignment** The purpose of this attribute is twofold: 1) capture to which degree of parallelism a quantified assignment should be executed; 2) detect anti-dependency among assignments within the same quantified assignment.

The first (default) value implies the totally parallel implementation while the second partially parallel ones (with varying degrees). In fact, we have combined with each "second" class value different flags indicating if the quantified assignment will be serialized at the level of assignments or quantification. In the first case, elements are performed sequentially using a single hardware component. In the second case, elements are performed in parallel, where each selected quantification is performed in a sequential loop. This flexibility allows analysis of all possible levels of parallelism, as well as all combinations of these for all elements. It also applies to the nested quantifications.

Analysis of SIPA is very helpful in terms of partitioning. By allocating data dependent elements within the same quantified assignment to the same FU, synchronization cost can be strongly reduced. Moreover, the flexibility of analyzing design alternatives with different parallelism degrees enables reusability

of resources by allocating parts of quantifications with same structure and behavior to same FU. By this means, optimal area/delay trade-off can be achieved. More on this is later.

**ME: Mutual Exclusion** Discovery of ME can improve performance since conditions for mutually exclusive elements can be evaluated in parallel. Furthermore, designer has the flexibility to select between parallel and sequential evaluation of mutual exclusive conditions. This helps to find good trade-off by assessing the achieved benefit.

**Mp: Multiplicity** With this attribute we can distinguish simple enumerated assignments from multiple ones. Detecting multiple kind of statements and allocating them to a superscalar processor, e.g. a RISC processor, can improve performance without additional area cost, thus makes good use of resources.

All the three elements in equation (2) are for example multiple. Implementing multiple assignments sequentially will lead to another implementation alternative with a different degree of parallelism.

### 3.3 Selection of Alternatives

As pointed out before, the classification results in implementation alternatives. Our approach allows each time, in case several runs of the clustering process are necessary, an alternative to be selected for the upcoming clustering process, either by the designer, or by default (parallel or pipelining), or guided by cost functions (in case of SIQA). The cost function attempts to find the most suitable current class values to balance both the parallelism degree of synchronous elements and the delay of asynchronous elements, in case they can be pipelined, so that area/delay trade-off is optimized. For more detail on the cost function, please refer to [Bar93].

## 4 Partitioning

### 4.1 Introduction and Overview

Multi-stage clustering technique has two major advantages over traditional single stage one:

- reducing the circumstance of weighing each criterion against others, since criteria can be separated and distributed over several stages
- handling objects as group rather than individuals. This feature makes clustering more appropriate for high-level partitioning where often a great amount of objects are considered

Multi-stage clustering technique has been used in several synthesis systems for data path design and architecture partitioning [Lag89]. Figure 2 and the following description show how we make use of this technique in our approach. To begin with, a function is

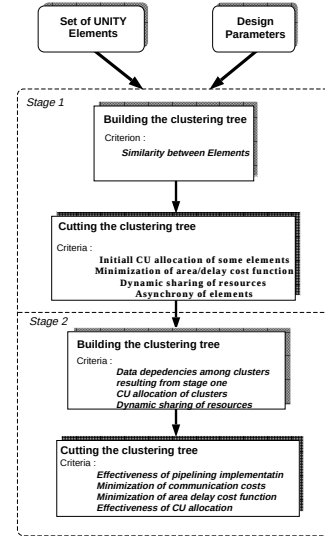


Figure 2: Overview of our clustering process

defined according to the classification of elements to measure the extent of their “similarity” among each other, according to the types of the current attributes attached to this element, as well as selected current class values of each attribute. This results in a distance matrix, with its help a cluster tree can be build.

Then, a cost function is defined to reflect the criteria for finding the most proper outline of the cluster tree, i.e. initial allocation of clusters into CU (software component) unit (either guided by cost functions or selected by the designer), resource sharing if it minimizes the area/delay cost, and separation of asynchronous elements for possible pipelining.

At stage two, clusters resulting from stage one are used as clustering objects to build cluster tree. To gauge the closeness among clusters, asynchronous clusters are defined to be distant from each other to make possible pipelining feasible. Synchronous clusters with anti-dependency are kept together to reduce synchronization cost as pointed before. Factors concerning CU allocation and dynamic sharing are also considered in the cost function. Also, some inherited feature (resource sharing) from stage one has to be maintained in the cost function. To outline cost function explores the feasibility of pipelining under consideration of the area/delay cost.

## 4.2 Setup of the Cluster Tree

After the classification, an implementation alternative (current class value) for each element is selected as reference for the clustering. To build the cluster tree, distances among elements will be calculated according to the following guidelines:

- Elements of similar parallelism degree are kept closer to each other
- Elements classified to share same FU are kept together
- Synchronous elements with anti-dependency are kept close to minimize synchronization cost
- Asynchronous elements not sharing common FU are kept separate to enable more parallelism
- Mutually exclusive elements running in parallel are kept separate
- Elements of similar multiplicity are kept closer for a better resource utilization

Except for the first criteria which deals with the types of assignments that the elements belong to, all the other criteria are related to current class values of individual attribute. The distance function is defined as:  $D(e_1, e_2) = D_{AssignmentType}(e_1, e_2) + D_{ClassValue}(e_1, e_2)$ . Distances of all elements build a distance matrix, from which a cluster tree can be built. The algorithm for building a cluster tree from a distance matrix can be found in [Joh67].

## 4.3 Cut the Cluster Tree

Criteria for detection of the cutline are already described in the introduction. The cost function is defined as:  $f_{cut1}(s, c) = f_{Asyn1}(s, c) + f_{CUAlloc1}(s, c) + f_{AreaDelay}(s, c)$ , where  $s$  is the clustering sequence of cluster tree  $s$ .  $f_{Asyn1}(s, c)$  suggests that asynchronous elements sharing common FU be kept in a cluster ( $f_{Asyn1}(s, c) = 1$ ) while those entitled to pipelining and those assigned to different FUs be kept separate ( $f_{Asyn1}(s, c) = \infty$ ).

$f_{CUAlloc1}(s, c)$  causes elements pre-allocated to CU to be separate from elements assigned to hardware. Good candidates for CU allocation hereby are elements belonging to statement with the least assignments (to reduce the synchronization cost). Among these elements, multiple ones are more favored since they could be executed in single clock cycle in a superscalar processor. Eventually, elements conformed

to both the above conditions and at the same time minimizing the area/delay cost are allocated to CU.

Finally,  $f_{AreaDelay}(s, c)$  evaluates the area/delay cost concerning both dynamic and static sharing.

### Algorithm for finding the cutline at stage 1

- $s = 0$
- **until** ( $f_{cut1}(s + 1, c) > f_{cut1}(s, c')$ ) **or** ( $c$  consists of single cluster)
  - do**  $s \leftarrow s + 1$ ;
  - place cut line at the level of clustering  $s$
  - od**

## 4.4 Clustering at Stage 2

According to the criteria given in the introduction, a new distance matrix for the clusters (resulting from stage 1) can be established and from it a cluster tree can be set up.

The goal of clustering at the second stage is to determine whether asynchronous clusters will be implemented as pipelining, since pipelining is also accompanied by additional delays. First, a cost function is defined to detect the suitable cluster sets. Then, benefit of pipelining implementation concerning performance improvement will be analyzed by further cost function. Finally, a cost function serving as guidance for finding the cutline will take both the factors of performance speedup by pipelining and minimization of area/delay cost into account.

To find the most suitable cutting, the cutline is initially placed at the top of the cluster tree and then gradually moved downwards, resulting in more and more clusters. The search process continues in this manner until the cost function will not be reduced. Figure 5 for the matrix inverse example illustrates this process. The generated clusters will be finally allo-

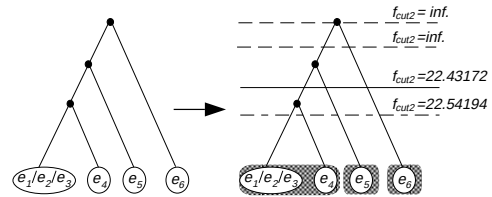


Figure 3: The cluster tree before/after cutting

cated to the underlying target architecture:

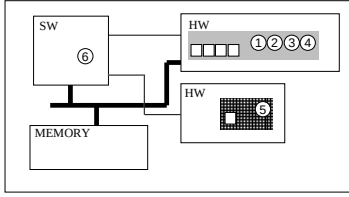


Figure 4: Allocation of clusters

## 5 Exploration of the Design Space

For the evaluation's purpose, we have applied the algorithm to different examples: a multi-variable PID controller, matrix inverse, priority queue ... etc.

For the sake of completeness, we examined all possible classifications to cover all possible implementations for each selected application, in terms of resource sharing, balance of the parallelism degree, implementation with or without pipelining, and so on. Also, pre-allocation of distinct elements into CU is considered. Figure 5 shows the result for the matrix in-

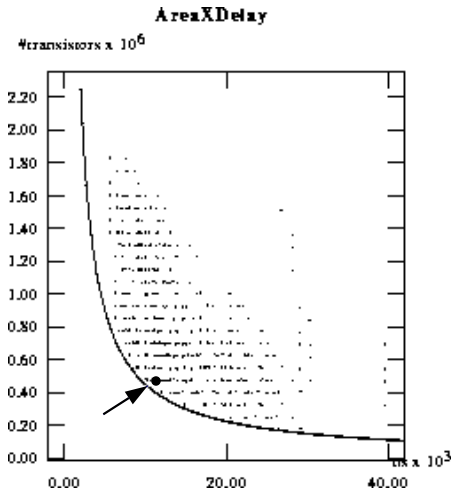


Figure 5: Exploration of the Design Space

verse example. The curve indicates the optimal results according to the chosen weight factors and in the area/delay cost function. Scattered points over the curve are area/delay values resulting from other partitioning alternatives. The arrow in the figure indicates one of the solutions found by our approach with some specified area/delay trade-off and it is very close to the optimal curve. Experiments with other examples have also been very successful.

## 6 Conclusion and Outlook

We have presented a partitioning method for UNITY language. This method explores a wide spectrum of design space by considering different design alternatives. The partitioning is gradually done and refined in a two stage clustering process.

Like any other language UNITY has both strong and weak points. Besides its many merits like direct expression of synchrony, its major drawbacks are lack of control information and the absence of hierarchy etc.

Our method is originally designed for UNITY. However, its major idea can also be conveyed to other specification languages. One possible expansion may be use of the specification mechanism based on the UNITY concept of synchrony and asynchrony while including mechanism for specifying control information and sequential execution. Another possibility is to find some already existent specification mechanism (e.g. C++, C, VHDL), which are powerful enough to specify more complex systems. Our future work will mainly consider this aspect.

## References

- [Bar93] E. Barros. *Hardware/Software Partitioning using UNITY*. PhD thesis, University of Tübingen, July 1993.
- [CM88] K. Chandy and J. Misra. *A Foundation of Parallel Programs Design*. Prentice-Hall, 1988.
- [EH92] R. Ernst and J. Henkel. Hardware-Software Codesign of Embedded Controllers Based on Hardware Extraction. In *Handouts of the International Workshop on Hardware-Software Co-Design*, Estes Park, Colorado, October 1992.
- [GM92] R. Gupta and G. De Micheli. System-level Synthesis Using Re-programable Components. In *Proceedings of EDAC*, pages 2–7, Brussels, Belgium, March 1992.
- [Joh67] S.C. Johnson. Hierarchical clustering schemes. In *Psychometrika* 32, pages 241–254, 1967.
- [Lag89] E.D. Lagnese. *Architectural Partitioning for System Level Design of Integrated Circuits*. PhD thesis, Carnegie Mellon University, March 1989.