An Array Processor Design Methodology for Hard Real-Time Systems

J.A.K.S. Jayasinghe, F. Moelaert El-Hadidy and O.E. Herrmann

University of Twente, P.O. Box 217
7500 AE Enschede, The Netherlands

Abstract

Many hard real-time systems need huge computing power and they are mostly designed by ad hoc methods. Array processors provide a viable means to achieve huge computing power and they can be designed systematically. This paper presents a systematic design methodology to design array processor based hard real-time systems.

1 Introduction

Real-time systems must produce not only logically correct results, but also meet timing constraints. Depending on the types of timing constraints, real-time systems are divided into two groups: *Hard real-time systems* and *Soft real-time systems* [1], [2]. A soft real-time system must produce computations as fast as possible such that a statistically described response time is satisfied. In a hard real-time system, computations must be finished before a given deadline.

Analogous to the status of VLSI design at its infancy, currently there is no scientific basis for hard real-time system design [2]. Though most state-of-the-art hard real-time systems have been designed by ad hoc methods, a scientific approach for hard real-time system design is essential as verification of the ad hoc designs are costly and error prone. Due to huge processing power requirements, almost all hard real-time systems need a multiprocessing environment. According to [2], a multiprocessor hard real-time system must possess the following features: *Homogeneity, Scalability, Survivability* and *Flexibility.*

Array processors consist of a set of modular processing elements (PEs) with spatially local communication, which makes them homogeneous and scalable. Survivability and flexibility can be introduced in the array processor design as well. Furthermore, systematic methods are used in array processor designing. These factors make array processor based hard real-time systems very attractive. The array processors operating with synchronous (asynchronous) communication are called systolic (wavefront) arrays. As the array processor contains modular PEs, only design problems associated with regular or partially-regular dependence graphs are considered for array processor design.

The rest of this paper is organized as follows. In Section 2, we briefly describe the widely used dependence graph approach and its limitations for real-time array processor design. In Section 3, our design methodology is presented. Finally, conclusions are drawn in Section 4.

2 Dependence Graph Based Array Processor Design and its Limitations

In this methodology, first an algorithm is developed in so called *single assignment code*, where each variable is only allowed to get a single value. Then the algorithm is represented in a graphical form by so called *dependence graph* (DG) [3]. The nodes of the DG are then mapped into an array processor. Construction of the single assignment description for large and complex problems is tedious and furthermore, they are associated with DGs described in higher dimensional Euclidean spaces. Therefore, manual mapping procedures are impractical as the visualization of the DG is tedious. Therefore, it is necessary to automate the mapping of the DG into an array processor. In literature, several techniques and software packages have been reported for the automation of the mapping like in [4], [5], [6], [7], but unfortunately, only regular DGs can be handled by these. Therefore, the current practice is to make the DG regular while the algorithm is written in single assignment form [8]. If the given problem is not associated with a regular DG, dummy operations can be added to get a regular DG. The DGs for large and complex problems are not regular in general and are very difficult to make regular by adding dummy operations. On the other hand, dummy nodes keep the PEs in the array processor busy unnecessarily. This could prevent the ability to meet hard real-time deadlines.

3 Structured Dependence Graph Based Array Processor Design

To simplify the construction of the *single assignment code*, we construct it hierarchically. This generates a set of DGs which are then combined to get the DG of the given problem. This DG is then projected into an abstract processor array using integer programming. Due to the generality of this approach, it can be used for partially-regular and regular DGs. Furthermore, it enables the projection of the DGs linearly as well as nonlinearly. In general, the abstract processor array resulting from a projection of a partially-regular DG contains processors whose behaviors are time varying. With the help of a set of tags, the abstract processor array is mapped into an array processor. These tags control the time-varying behavior, improve the regularity, survivability and flexibility of the array processor. In the following subsections, we describe these design steps briefly. More details are given in [9].

3.1 The Structured Single Assignment Code (S^2AC).

The S^2AC description consists of a set of hierarchical routines where each routine is described by a header and a body. Only a single assignment is made to every variable in each routine. We refer to the top-most routine as the level-0 routine and the routines in the next level as level-1 routines and so on. To simplify the construction of the DG, level-i routines are only allowed to call routines in the level-$(i+1)$. Only atomic operations are used in the last level routines. The following syntax is used to write the S^2AC description. All but the last level routines use data types *array* and *record* as defined in conventional structured programming languages. An *array* represents a set of data on which the same operation is performed. A *record* represents a set of data on which different operations are performed. The header of a routine consists of output variables, the name of the routine and input variables. The body of a routine consists of four fields: *type declaration field* (where the data types of input and output variables are declared), *initialization field* (where local variables of the routine are initialized), *variable assignment field* (where the values of variables are calculated by calling lower level routines or by performing atomic operations) and *output assignment field* (where output variables are updated). The second and last fields are optional. A formal description of the syntax of the S^2AC description is given in [9].

3.2 The Structured Dependence Graph (SDG) and Expansion

The SDG contains a DG for each routine in the S^2AC description. To indicate the hierarchy of the S^2AC description, the SDG is defined in Definition 3.4 with the aid of the following auxiliary definitions.

Definition 3.1 *Any edge that supplies (produces) data to (from) a DG is said to be an* **Input (Output)** **Edge** *of the DG. The node to (from) which the input (output) edge supplies (produces) data is said to be the* **Input (Output) Node**.

Definition 3.2 *Any set of parallel input and/or output edges of a DG is defined as an* **Edge Bundle**. *We use the symbol EB to denote an edge bundle.*

Definition 3.3 *An edge bundle EB is called an* **Input (Output) Edge Bundle** *if all members of the EB are input (output) edges.*

Definition 3.4 *A family of dependence graphs represented by $N+1$ sets $G^0, G^1, ..., G^N$ is defined as a* **Structured Dependence Graph** *if there exists a family of dependence graphs $g_k^i \in G^i$ $(0 \leq i < N)$ such that:*

1. *The nodes of g_k^i are labelled by a set of graphs $\{g_{m_1}^{(i+1)}, g_{m_2}^{(i+1)}, \ldots\}$ where $g_{m_l}^{(i+1)} \in G^{(i+1)}$ $(l = 1, 2, \ldots)$.*
2. *Each inbound edge connected to a node labelled by $g_m^{(i+1)}$ is also labelled by a unique input edge bundle of $g_m^{(i+1)}$.*
3. *Each outbound edge connected to a node labelled by $g_m^{(i+1)}$ is also labelled by a unique output edge bundle of $g_m^{(i+1)}$.*

Then, we construct a DG by the expansion of the SDG as defined below.

Definition 3.5 *If an SDG is converted into a single DG by recursively replacing all the labelled nodes by the relevant graph referred by its node label such that:*

1. *all the labelled input and output edges are replaced by a set of edges corresponding to the edges referred by the input and output edge bundles respectively,*
2. *all other labelled edges are replaced by a set of edges connecting the node where the ith edge in the output edge bundle is originated to the node where the ith edge in the input edge bundle is terminated,*

then the resultant DG is said to be an **Expansion** *of the SDG.*

3.3 The Canonical SDG

In array processor design, DGs containing only local-dependence edges are of importance. Furthermore, the projection of the DG becomes easy if we can construct the DG in a minimum dimensional Euclidean space. Therefore, we define a canonical SDG which will be expanded to create the DG which will be used for succeeding design steps.

Definition 3.6 *An SDG is said to be expandable by* **Abutment** *if the graphs referred by the node labels can be placed next to each other such that, for each labelled edge, the node where the ith edge in the output edge bundle is originated can be connected to the node where the ith edge in the input edge bundle is terminated without introducing any nonlocal-dependence edges in the resultant DG when the labelled edge is not an input or output edge. While placing two graphs next to each other, they are allowed to be rotated and/or mirrored to prevent the introduction of nonlocal-dependence edges.*

Definition 3.7 *An SDG is said to be a* **Canonical SDG** *if the following conditions are satisfied.*

1. *All the members of the SDG are defined in a common n-dimensional Euclidean space.*
2. *The expansion of the SDG can be done by abutment.*
3. *If the graph referred by a node label has to be rotated and/or mirrored during the expansion, then that node must be tagged with the information regarding how to rotate and/or mirror.*
4. *When an edge is labelled by an input and output edge bundle then:*
   - (a) *If the graphs referred by the terminal nodes of the edge are not tagged for rotation and/or mirroring, then the directions of the edges in both bundles must be the same.*
   - (b) *If the graphs referred by the terminal nodes of the edge are tagged for rotation and/or mirroring, then the directions of the edges in both bundles must be the same after the rotation and/or mirroring.*
5. *Each member of the SDG is defined in a minimum dimensional subspace of the common n-dimensional Euclidean space.*
6. *Each member of the SDG is defined such that the partial graph resulted by the expansion of the terminal nodes of any labelled edge is in a minimum dimensional Euclidean space when the edge is not an input or output edge.*

3.4 Integer Programming Formulation of the Scheduling and Projection of DGs

For a given n-dimensional DG, our goal is to project the DG into a lower dimensional abstract processor array such that timing constraints are not violated. For this, we solve two integer programming problems. For the scheduling problem, we assign a scaler s_i to each DG node indicating its execution time. For the projection problem, we assign an m-vector $\vec{p}_i$ to each DG node. Here $\vec{p}_i$ indicates the target position of the processor element (PE) where the function of the i^{th} DG node is performed.

3.4.1 The Scheduling Problem for Systolic Array Design

The number of delay registers introduced into the systolic array primarily depends on the scheduling of the DG nodes, and hence we try to get the best schedule in the first place. The number of delay registers introduced into the systolic array also depends on the projection, because sometimes one can reuse the same delay register several times. As the projection is still an unknown, we use the function

$$f_s = \sum_{for\ each\ DG\ edge} s_{destination\ node} - s_{source\ node} \qquad (1)$$

as our objective function. This will be an upper bound for the number of registers necessary. The duration of the time slot (or cycle time) is chosen according to the propagation delay of the slowest DG node.

To satisfy the precedence conditions, we introduce the following constraint for each edge in the DG:

$$s_{destination\ node} - s_{source\ node} \geq 1 \qquad (2)$$

We must schedule the DG such that the input and output timing constraints are met. Therefore,

$$s_{output\ node} \leq d_{output} \qquad (3)$$
$$s_{input\ node} \geq d_{input} \qquad (4)$$

where d_{output} and d_{input} are the deadlines and input arrival time expressed in terms of number of cycles. To reflect the (partial) regularity in the DG we use the following constraint:

$$s_{destination\ node} - s_{source\ node} = r_l \qquad (5)$$

by introducing a variable r_l for all the edges along the same direction connected to the same type of DG nodes. In practice, it is convenient to restrict the number of neighboring PEs to which a given PE can communicate to reduce the number of communication links necessary. We ensure this requirement by using the following constraint for each node k in the DG:

$$\text{For all } i \in N_k, \qquad \sum_{j \in N_k,\ j \neq i} g(s_i - s_j) \leq C - 1 \qquad (6)$$

where N_k represents the set of nodes connected to node k and C is the maximum number of neighboring PEs to which a PE can communicate. By defining $g(r)$ such that $g(0) = 1$ and $g(r) = 0$ for $r = \pm1, \pm2, \ldots$, we find whether node i and j have the same schedule time or not.

Apart from these constraints, we allow the designer to insert a set of optional constraints to ensure the projection of a set of DG nodes into the same PE. In this case, these nodes must be scheduled in different time slots which can be ensured by the following constraint:

$$\text{For all } i \in U_k, \qquad \sum_{j \in U_k,\ j \neq i} g(s_i - s_j) = 0 \qquad (7)$$

where U_k is the k^{th} user specified node set.

3.4.2 The Projection Problem for Systolic Array Design

Once the scheduling is known, we can reduce the number of PEs necessary for the systolic array by projecting several DG nodes into a single PE in an abstract processor array. Therefore, for the projection problem, the objective function must represent a measure of number of PEs. Let i and j be two neighboring nodes connected by an edge and $\vec{c}^T = [\underbrace{1\ 1\ \ldots\ 1}_{m}]$. As the i^{th} DG node is projected into a PE in the abstract processor array located at $\vec{p}_i$, DG nodes i and j will be projected into the same PE when $\vec{p}_i = \vec{p}_j$ $(i \neq j)$. Then we have $\vec{c}^T\vec{p}_i - \vec{c}^T\vec{p}_j = 0$. Let

$$f(k) = \begin{cases} 1 & \text{for } k = 0 \\ 2 & \text{for } k = \pm1, \pm2, \ldots \end{cases} \qquad (8)$$

Then,

$$\sum_{for\ each\ DG\ edge} f(\vec{c}^T\vec{p}_{source\ node} - \vec{c}^T\vec{p}_{destination\ node}) \qquad (9)$$

will be an upper bound for the number of PEs necessary for the systolic array. The projection must preserve the near-neighbor communication. Therefore, for all DG edges, we add the following constraints:

$$\begin{aligned} \vec{p}_{destination\ node} - \vec{p}_{source\ node} &\leq \vec{c} \\ \vec{p}_{destination\ node} - \vec{p}_{source\ node} &\geq -\vec{c} \end{aligned} \qquad (10)$$

To reflect the (partial) regularity in the DG we use the constraint,

$$\vec{p}_{destination\ node} - \vec{p}_{source\ node} = \vec{r}_l \qquad (11)$$

for the DG edges along the extreme boundaries of regular sub-DGs. Here, r_l is a unique variable for each boundary where regular sub-DGs of different types are connected. In the case that limited number of communication links are allowed, only certain components of $\vec{p}_{destination\ node} - \vec{p}_{source\ node}$ are allowed to be nonzero. For simplicity, we consider the case where communication links are only allowed along the coordinate axes. In this case we add the following constraint:

$$\vec{c}^T(\vec{p}_{destination\ node} - \vec{p}_{source\ node}) \leq 1 \qquad (12)$$

Furthermore, we cannot project two nodes into the same PE if they have the same schedule time. Therefore, for all DG node pairs having the same schedule time, we add the following constraint:

$$\vec{p}_i - \vec{p}_j \neq 0 \qquad (13)$$

If there are S_i DG nodes scheduled for the i^{th} time slot, then the above equation introduces $\sum_i S_i^2 - S_i$ constraints. The number of constraints can be dramatically reduced by exploiting the near-neighbor communication property. In that case, we can relax the above constraint for all nonadjacent node pairs, which cannot be projected into the same PE without violating the near-neighbor communication property.

In addition to the previous constraints, we can add the following optional constraints also. To prevent two DG nodes from being projected into the same PE, we can add the constraint, $\vec{p}_i - \vec{p}_j \neq 0$. On the other hand, the designer might prefer to project some specific nodes into a single PE. Constraints of the form $\vec{p}_i - \vec{p}_j = 0$ can be inserted to obtain such preferences. These optional constraints can be first left out and then they can be gradually inserted to eliminate undesirable features of the resulting design. We recommend to insert these optional constraints by inspecting the members of the SDG or automatically by specifying a set of predicates. If one is only interested in linear projections, the constraint $\vec{p}_{destination\ node} - \vec{p}_{source\ node} = \vec{k}_l$ can be introduced to each different edge direction.

Once the optimum values for $\vec{p}_i$ are known for all the DG nodes, an abstract processor array can be built as follows: For all DG nodes, define a PE at the location described by $\vec{p}_i$. For all DG edges, define a communication link from the PE location $\vec{p}_{source\ node}$ to $\vec{p}_{destination\ node}$ in the abstract processor array. For any DG edge, the number of delay registers required on the corresponding communication link on the abstract processor array is given by $s_{destination\ node} - s_{source\ node}$. Multiple communication links between two PEs can be merged together when the communication time slots of these links do not overlap and the number of delay registers on each link is the same.

3.4.3 The Scheduling and Projection Problems for Wavefront Array Design

According to [3], any systolic array can be converted into a wavefront array simply by replacing each synchronous communication link by an asynchronous communication link and replacing each delay register by an initial token. We adopt the same technique due to its simplicity. Therefore, in summary, the scheduling and projection problems for the wavefront array design are identical to those of systolic array design.

3.5 Tag Based Control

In general, an abstract processor array resulting from a projection of a partially-regular DG contains PEs performing time-varying functions. We map this abstract processor array into an array processor by designing a set of super-PEs whose functionality is controlled by *Tags*. A supper-PE is modular and it is capable of performing all the functions of a set of PEs in the abstract processor array. At each PE, we place a *residential-tag* to identify the different PE types. A set of *mobile-tags* is propagated through the array in a controlled fashion to control the time-varying behavior. The mobile tags are sent through the array via a so called *valid-tag-path* defined below.

Definition 3.8 *Any out-tree of the DG is said to be a* **Valid-Tag-Tree** *if the root node of the tree represents the virtual node where all the input edges of the DG are virtually connected and the terminal nodes of any edge in the tree except for the edges from the root node are scheduled in consecutive time slots*[1].

Definition 3.9 *The path defined by the projection of a valid-tag-tree into the abstract processor array is said to be a* **Valid-Tag-Path**.

As the functions performed by the PEs are controlled by tags, the functions performed by a faulty PE can be switched off and executed on a neighboring PE. This provides a survivability to the array processor. Different combination of tags are equivalent to different algorithms. Therefore, we can get a flexible array processor capable of executing a set of algorithms which can be mapped onto the selected topology simply by sending different combinations of tags.

4 Conclusions

An array processor design methodology suitable for hard real-time system is presented. Scheduling and Projection of the DG is solved using integer programming. By exploring the regularity of the DG we can solve the necessary IP problems in an efficient manner. This methodology provides a unified approach for linear and nonlinear projection of regular and partially-regular DGs.

References

[1] C.L. Liu and J.W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, in Hard Real-Time Systems, pp. 174-189, IEEE Computer Society Press, 1988.

[2] John A. Stankoric, *Real-time Computing Systems: The Next Generation*, in Hard Real-Time Systems, pp. 14-37, IEEE Computer Society Press, 1988.

[3] S.Y Kung, *VLSI Array Processors*, Prentice Hall, 1988.

[4] Sailesh K. Rao and Thomas Kaileth, *Regular Iterative Algorithms and their Implementation on Processor Arrays*, Proceedings of the IEEE, Vol. 76, NO. 3, pp. 259-269, March 1988.

[5] P. Quinton, *Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations*, Proceedings of te Symposium on Computer Architecture, pp. 208-214, 1984.

[6] W. Sang and J.A.B. Fortes, *Time Optimal Linear Schedules for Algorithms with Uniform Dependencies*, Proceedings of the International Conference in Systolic Arrays, pp. 393- , 1988.

[7] Dan I. Moldovan *ADVIS: A Software Package for the Design of Systolic Arrays*, IEEE Transaction on Computer-Aided Design, Vol CAD-6, No. 1, pp. 33-39, January 1987.

[8] E.T.L. Omizigt, *SYSTARS: A CAD Tool for the Synthesis and Analysis of VLSI Systolic/Wavefront Arrays*, in Proceedings of the International Conference in Systolic Arrays, pp. 383- , 1988.

[9] J.A.K.S. Jayasinghe, *An Array Processor Design Methodology for Hard Real-Time Systems*, Ph.D. Thesis, University of Twente, ISBN 90-9004031-5, 1991.

[1]The requirement of the consecutive schedules has been imposed to simplify the controller design. In fact this can be relaxed while keeping some form of regularity.