From Algorithms To Parallel Architectures:
A Formal Approach

Khaled M. Elleitby
Computer Engineering Department
King Fahd University
Dhahran 31261
Saudi Arabia

Magdy A. Bayoumi
The Center For Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
U.S.A.

Abstract

In this paper, we introduce a formal approach for synthesis of parallel architectures. Four different forms are used to express the given algorithms: simultaneous recursion, recursion with respect to different variables, fixed nesting and variable nesting. Four different architectures for the same algorithm are obtained. As an example, a matrix-matrix multiplication algorithm is used to obtain four different optimal architectures. The different architectures of this example are compared in terms of area, time, broadcasting and required hardware. The approach is providing two main features: completeness and correctness.

1. Introduction

Formal high level synthesis of general architectures is an important design phase to ensure functional, correct and cost effective architectures. Recently, there have been several efforts in this direction [1-5], but many of these efforts have not included parallel architectures. There have been several synthesis approaches for synthesizing special class of arrays [6,7]. In this paper we present a formal system for synthesizing parallel architectures. The architectures produced by this system can be classified as uniprocessor architectures. To exploit the parallelism in a given algorithm the methodology has been generalized so that it can be applied to simultaneous recursion forms [10]. In this paper we shall extend the methodology by applying it to the following forms: recursion with respect to several variables, fixed nested recursion and variable nested recursion. Recursion with respect to several variables will be discussed in detail. For a complete discussion on all the forms please refer to [10]. The methodology provides two main features: completeness and correctness. Completeness means the ability to use the approach for any general algorithm. Correctness is achieved by using a set of transformations that are proved to be correct. A design example of matrix-matrix multiplication is used with each one of the forms to obtain a parallel architecture. These different architectures for this example are compared in terms of area and speed.

1.1. System Overview

Figure 1 shows the different components of our formal system. The system is composed from two subsystems: synthesis subsystem and user-interface subsystem. A new language Algorithm Specification Language (ASL) based on μ-recursive functions is used to specify the given algorithm. Transformation techniques are used to transform an algorithm specified in ASL to a realization language called RSL. Every construct in ASL has an isomorphic representation in RSL which is the basis of the automated transformation.

A logic programming environment based on Prolog, is employed as a user interface to the synthesis process. The logic programming environment supports specifying, simulating, and testing the target systems. Prolog provides homogeneity to the developed system as it supports hierarchical development and mixing of description at various hierarchical levels. For more details on the synthesis subsystem and the user interface subsystem please refer to [8-10]. In the next sub-section, due to space limitations, we shall present only the recursion with respect to several variables synthesis approach in detail.

2. Recursion with respect to...
several variables

If \( x_i \) \((1 \leq i \leq n)\) are \( n-1 \) place functions, \( x \) is \( n \) place functions, \( y \) is \( 2n \) place function and \( w_j \) are \( n \) place functions, then \( z \) is defined by ASL Code ASL1.

Transformation Algorithm to RSL

To transform the system of recursion with respect to several variables to RSL we implement each equation using the same method described in [9]. RSL1 is the RSL representation of the system:

Equation 1 is used to show that we use \( n \) registers to be initialized with the arguments \((\arg_{1}, \ldots, \arg_{n})\). Equation 2 means that the unit \( Suc \) which is a basic function has its inputs \( control_i \) \((1 \leq i \leq r)\) connected to the \( ready \) output of the unit computing \( x_i \) to be sure that \( I \) is not incremented until \( x_i \) is computed. Equation 3 is used to represent the fact that \( I \) is incremented every clock cycle using the \( Suc \) unit, and \( I \) is initialized to the value 1 using the register number \( n+1 \). Equation 4 determines the end of operation when \( I \) reaches the value \( m \). Equations 5,6,7 represent the composition operation in equations 1,2,3 of the ASL representation respectively. The architecture for the recursion with respect to several variables is shown in Figure 2. Similar analysis has been done for the other two approaches: fixed nested recursion and variable nested recursion [10].

Table 1 shows a comparison among these different forms of recursion in terms of architecture, broadcasting and complexity of the controller. The simultaneous recursion is the only form that gives a two dimensional array. All forms have broadcasting except the variable nesting. The controller of the variable nesting is complex compared with the other three forms.

3. Matrix-Matrix Multiplication Example

An example of matrix multiplication is introduced as an application of different forms of recursion. The architecture has two matrices \( A \) and \( B \) as inputs, and matrix \( C \) as an output. The multiplication is done in a recursive way and can be described by the following high level subroutine:

```plaintext
matrix_multiplication (A,B)
begin
for i=1 to n
  for j=1 to n
    begin
      \( C_{i,j,0} = 0 \)
      for k=1 to n
        \( C_{i,j,k} = C_{i,j,k-1} + A_{i,k} * B_{k,j} \)
      next k
    end
  next i
next j
end
```

For the ASL and RSL representation using the four recursive forms please refer to [10].

Figure 3 shows the architecture obtained for matrix multiplication for the case of recursive equations with several variables. The details of implementing the inner-product cell are shown in [9]. The architecture consists of \( N^2 \) inner-product cells. The number of cycles required to perform the multiplication is \( N \).

Figure 4 shows the architecture using recursion with respect to several variables. The architecture consists of \( N \) multiplication cells and one adder. The number of cycles required to perform the multiplication is \( N^4 \).

Figure 5 shows the architecture using fixed nesting recursion. The architecture consists of \( N \) inner-product cells. The number of cycles required to perform the multiplication is \( N^4 \).

Table 2 shows a comparison between different architectures of the matrix-matrix multiplication.

4. Conclusions

In this paper an formal approach for transforming different forms of recursion to parallel architectures has been introduced. Four different forms are used to express a given algorithm. Four optimal architectures for a matrix-matrix multiplication are compared. The developed approach represents the first step towards developing a high level synthesis system for general parallel architectures. It ensures correctness, but it does not address optimality which is considered as an important issue too. The developed approach has the following advantages:

[1] It is suitable for large problems since the transformation algorithm is linear.
It does not require to know the target architecture in advance.

The technique is fully automated.

The designer is not responsible for specifying the operations sequencing and communications among different units.

The approach is applicable to any general algorithm.

5. References


Acknowledgements: The first author wishes to thank King Fahd University of Petroleum and Minerals for support.
\[ z(0, \arg_2, \ldots, \arg_n) = x_1(\arg_2, \arg_3, \ldots, \arg_n) \]
\[ z(\arg_1 + 1, 0, \arg_3, \ldots, \arg_n) = x_1(\arg_1, \arg_3, \ldots, \arg_n) \]
\[ z(\arg_1 + 1, \arg_2 + 1, \ldots, \arg_{n-1} + 1, 0) = x_n(\arg_1, \arg_2, \ldots, \arg_{n-1}) \]
\[ z(\arg_1 + 1, \arg_2 + 1, \ldots, \arg_n + 1) = y(\arg_1, \ldots, \arg_n, z_1, \ldots, z_n) \]

\[ z_1 = z(\arg_1, w_1^{(1)}, w_2^{(1)}, \ldots, w_{n-1}^{(1)}) \]
\[ z_2 = z(\arg_1 + 1, \arg_2, w_1^{(2)}, w_2^{(2)}, \ldots, w_{n-2}^{(2)}) \]
\[ z_{n-1} = z(\arg_1 + 1, \ldots, \arg_{n-2} + 1, \arg_{n-1}, w_1^{(n-1)}) \]
\[ z_n = z(\arg_1 + 1, \ldots, \arg_n + 1, \arg_n) \]

**ASL1**: ASL code for recursion with respect to several variables

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Simultaneous</th>
<th>Several variables</th>
<th>Fixed nesting</th>
<th>Variable nesting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcasting</td>
<td>two dimensional</td>
<td>one dimensional</td>
<td>one dimensional</td>
<td>one dimensional</td>
</tr>
<tr>
<td>Complexity of controller</td>
<td>yes simple</td>
<td>yes simple</td>
<td>yes simple</td>
<td>no complex</td>
</tr>
</tbody>
</table>

**Table 1**: Comparison Between Different Forms of Recursion.

<table>
<thead>
<tr>
<th>Area</th>
<th>Simultaneous ( \theta(n^2) )</th>
<th>Several variables ( \theta(n^2) )</th>
<th>Nesting ( \theta(n^2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>( \theta(n^2) )</td>
<td>( \theta(n^2) )</td>
<td>( \theta(n^2) )</td>
</tr>
<tr>
<td>A*T</td>
<td>( \theta(n^2) )</td>
<td>( \theta(n^2) )</td>
<td>( \theta(n^2) )</td>
</tr>
<tr>
<td>Broadcasting</td>
<td>yes inner-product (n)</td>
<td>no multiplier (n)</td>
<td>inner-product (n)</td>
</tr>
<tr>
<td>Hardware</td>
<td>( \frac{1}{n^2} )</td>
<td>( \frac{1}{n^2} )</td>
<td>( \frac{1}{n^2} )</td>
</tr>
</tbody>
</table>

**Table 2**: Comparison Between Different Matrix-Matrix Multiplication Architectures.

361
\[ \text{Initp}(l, \text{arg}_1; 2, \text{arg}_2; \cdots; n, \text{arg}_n) \tag{1} \]

\[ \text{succ}_{\text{control}_1} = x_1(\text{arg})^{\text{ready}} \tag{2} \]

\[ \text{succ}_{\text{control}_n} = x_n(\text{arg})^{\text{ready}} \]

\[ I = \rho_1^{n+1} \text{succ}(I) \tag{3} \]

\[ \text{Ready} = \text{eq}(I, m) \tag{4} \]

\[ z(0, \text{arg}_2, \cdots, \text{arg}_n) = \text{Comp}(\text{arg}_2, \ldots, \text{arg}_n) \not= x_1 \]

\[ z(0, \text{arg}_2, \cdots, \text{arg}_n) = \text{Comp}(\text{arg}_2, \ldots, \text{arg}_n) \not= x_1 \]

\[ z(\text{arg}_1+1, 0, \text{arg}_3, \cdots, \text{arg}_n) = \text{Comp}(\text{arg}_1+1, 0, \text{arg}_3, \cdots, \text{arg}_n) \not= x_2 \]

\[ z(\text{arg}_1+1, 0, \text{arg}_3, \cdots, \text{arg}_n) = \text{Comp}(\text{arg}_1+1, 0, \text{arg}_3, \cdots, \text{arg}_n) \not= x_2 \]

\[ z(\text{arg}_1+1, \text{arg}_2+1, \cdots, \text{arg}_n-1, 0) = \text{Comp}(\text{arg}_1+1, \text{arg}_2+1, \cdots, \text{arg}_n-1+1 \not= x_n) \]

\[ z(\text{arg}_1+1, \text{arg}_2+1, \cdots, \text{arg}_n+1) = \text{Comp}(\text{arg}_1+1, \text{arg}_2+1, \cdots, \text{arg}_n+1) \not= z \]

\[ z(\text{arg}_1+1, \text{arg}_2+1, \cdots, \text{arg}_n+1, z_1, \ldots, z_n \not= y) \]

\[ z(\text{arg}_1+1, \text{arg}_2+1, \cdots, \text{arg}_n+1, z_1, \ldots, z_n \not= y) \]

\[ z_1 = \text{Comp}(\text{arg}_1, w_1^{(1)}, \ldots, w_{n-1}^{(1)} \not= z) \]

\[ z_1 = \text{Comp}(\text{arg}_1, w_1^{(1)}, \ldots, w_{n-1}^{(1)} \not= z) \]

\[ z_2 = \text{Comp}(\text{arg}_1+1, w_1^{(2)}, \ldots, w_{n-2}^{(2)} \not= z) \]

\[ z_2 = \text{Comp}(\text{arg}_1+1, w_1^{(2)}, \ldots, w_{n-2}^{(2)} \not= z) \]

\[ z_{n-1} = \text{Comp}(\text{arg}_1+1, \ldots, \text{arg}_n-1+1, w_1^{(n-1)} \not= z) \]

\[ z_{n-1} = \text{Comp}(\text{arg}_1+1, \ldots, \text{arg}_n-1+1, w_1^{(n-1)} \not= z) \]

\[ z_n = \text{Comp}(\text{arg}_1+1, \ldots, \text{arg}_n \not= z) \]

\[ z_n = \text{Comp}(\text{arg}_1+1, \ldots, \text{arg}_n \not= z) \]

\[ RSL_1: \text{RSL code for recursion with respect to several variables} \]

---

**RSL1: RSL code for recursion with respect to several variables**