Dependence Analysis and Architecture Design for Bit-Level Algorithms

Weijia Shang
Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
sw@cacs.usl.edu

Benjamin W. Wah
Coordinated Science Laboratory
University of Illinois, Urbana-Champaign
Urbana, IL 61801
wah@manip.crhc.uiuc.edu

Abstract:. *In designing application-specific bit-level architectures and in programming existing bit-level processor arrays, it is necessary to expand a word-level algorithm into its bit-level form before dependence analysis can be performed. In this paper, we consider dependence structures of bit-level algorithms as functions of three components — dependence structures of word-level algorithms, dependence structures of the arithmetic algorithms implementing word-wise operations, and algorithm expansions. Based on these components, we can derive dependence structures of bit-level algorithms without using time consuming general dependence analysis methods. To illustrate our approach, we derive two dependence structures for bit-level matrix multiplication and apply a method developed earlier* [5,6,10] *to design two bit-level architectures. One of these architectures is* $O(p)$ *times faster than the best word-level architecture, where p is the word length. The speedup we found here is true in general because a bit in a bit-level architecture goes to the next processor for processing as soon as it is available.*

1. INTRODUCTION

Bit-level architectures can exploit parallelism at the bit-level that is often ignored in word-level processor arrays. This is true because a bit in such a structure does not have to wait for other bits to finish before propagating to the next processor for processing. In this paper we study the design of application-specific bit-level architectures and the programming of existing bit-level processor arrays.

A general method involves three steps. A word-level algorithm of the application can first be expanded into a bit-level algorithm [8]; this is followed by an analysis of the dependence relations of the bit-level algorithm; finally, based on the bit-level dependence structure, the algorithm is mapped to a bit-level processor array.

This paper addresses the dependence analysis of an expanded bit-level algorithm and proposes a method for deriving the bit-level dependence structure without using time consuming general dependence analysis procedures. We discuss the mapping of bit-level algorithms and the design of special-purpose bit-level architectures. We further show that bit-level architectures can be $O(p)$ times faster than the corresponding word-level structures, where p is the word length. The solution presented in this paper is an important step towards systematically programming or designing bit-level processor arrays.

Many methods have been proposed for deriving dependence structures of algorithms with nested loops [9]. These methods generally involve finding all integer solutions of a set of linear Diophantine equations, followed by a verification to see if the integer solutions are inside the *index set* or *iteration space* of the algorithm. In an exact analysis, the time complexity of these methods is exponential with respect to the number of nested loops (or the *algorithm dimension*).

The general dependence analysis methods discussed above can be applied to find bit-level dependence structures. However, we can substantially reduce the complexity in finding bit-level dependence structures if we exploit properties of bit-level algorithm expansions in our dependence analysis. For instance, we can view a bit-level dependence structure as a function of the corresponding word-level dependence structure, the dependence structures of the arithmetic algorithms implementing the word-wise operations (such as multiplication, division and addition), and the algorithm expansions. This allows us to derive a bit-level dependence structure without representing the algorithm at the bit-level and using general dependence-detection procedures.

The idea presented above is discussed in detail in Section 3. Basically, our approach is to expand a word-level algorithm based on its word-level dependence structure and the dependence structures of the underlying arithmetic algorithms. (Two of these algorithms are presented in Section 3.) Since many word-level algorithms involve a limited number of word-level arithmetic algorithms, the dependence structures of these algorithms need to be derived only once. For example, word-level algorithms, such as matrix multiplications, LU decompositions and convolutions, involve only a limited number of arithmetic algorithms for multiplication, addition and division (such as add-shift multiplication and carry-save multiplication).

Once the bit-level dependence structure is known, the next step is to either design a bit-level architecture based on the dependence structure, or map the dependence structure to a bit-level processor array. This can be carried out by an extension of the design method we have developed earlier [4,5,6,10]. We

Research of the first author was supported in part by Louisiana Education Quality Support Fund under contract number LEQSF(1991-93)-RD-A-42, and in part by the National Science Foundation under Grants MIP 91-10940. Research of the second author was supported by the Joint Services Electronics Program Contract N00014-90-J-1270.

المنارة للاستشارات
www.manaraa.com

illustrate our approach by showing the design of two special-purpose bit-level architectures for matrix multiplication. One of the architectures presented is time optimal; that is, it has the minimum total execution time and is $O(p)$ times faster than the best word-level architecture, where p is the word length. This improvement is expected to be true in general for bit-level architectures, as a bit can propagate to the next processor for further processing as soon as it is available.

This paper is organized into five sections. After defining basic terminology in Section 2, we show in Section 3 our bit-level algorithm dependence analysis. Section 4 shows how to design bit-level architectures based on the dependence structures obtained in Section 3. Two bit-level architectures are presented for matrix multiplication. Section 5 concludes the paper.

2. TERMINOLOGY AND DEFINITIONS

Throughout this paper, *sets*, *matrices* and *row vectors* are denoted by capital letters; *column vectors* are represented by lower-case symbols with an overbar; and *scalars* are shown as lower-case letters. The *transpose* of a vector $\bar{v}$ is denoted as $\bar{v}^T$. Vector $\bar{0}$ denotes a row or column vector whose entries are all zeroes. The dimension of vector $\bar{0}$ and whether it denotes a row or column vector are implied by the context in which it is used. The rank of matrix A is denoted *rank(A)*. The set of integers is denoted Z. The notation $|C|$ and $|\alpha|$ represents the cardinality of set C and the absolute value of scalar α, respectively. Let $\bar{v}$ and $\bar{u}$ be two vectors. Then $\bar{v} \geq \bar{u}$ means that every component of $\bar{v}$ is greater than or equal to the corresponding component of $\bar{u}$.

Algorithms considered in this paper are represented by a special kind of Fortran-like nested Do loops having the following form.

$$
\begin{array}{l}
DO\ (j_1=l_1,\ u_1;\ j_2=l_2,\ u_2;\ \ldots,\ j_n=l_n,\ u_n\ ) \\
\quad S_1(\bar{j}) \\
\quad S_2(\bar{j}) \\
\quad \cdots \\
\quad S_q(\bar{j}) \\
END
\end{array}
\tag{2.1}
$$

Column vector $\bar{j} = [j_1, j_2, \ldots, j_n]^T$ is the index vector (also called the *index point*). $S_1(\bar{j})$, $S_2(\bar{j})$, ..., $S_q(\bar{j})$ are q assignment statements in iteration $\bar{j}$ having the form $x_k(g(\bar{j})) = f(x_1(h_1(\bar{j})), \ldots, x_t(h_t(\bar{j})))$, where $1 \leq k \leq t$, and $g()$, $h_i()$, $i=1, \ldots, t$, are linear functions of $\bar{j}$. The lower and upper bounds of the ith nested loop, $1 \leq i \leq n$, are denoted by l_i and u_i, respectively. Algorithms with n nested Do loops are called *n-dimensional algorithms*.

In general nested Do loops, cross-iteration dependences may exist. If iteration $\bar{j}$ depends on iteration $\bar{j}'$, then this dependence can be described by a pair $(\bar{j}, \bar{d})$, where $\bar{d} = \bar{j} - \bar{j}'$ is the vector difference of the index vectors of these two iterations. Vector $\bar{d}$ is called a *dependence vector* and is said to be *valid* at index point $\bar{j}$. Assuming that iteration $\bar{j}$ depends on iteration $\bar{j}'$, there are three types of dependences [1]. The first type is called *flow dependence* (or read-after-write dependence) where an input variable of the computation in $\bar{j}$ is an output variable of the computation in $\bar{j}'$. The second is called *anti-dependence* (or write-after-read dependence) where an output variable of the computation in iteration $\bar{j}$ is an input variable of the computation in iteration $\bar{j}'$. The third is called *output dependence* (or write-after-write dependence) where an output variable of the computation in iteration $\bar{j}$ is an output of the computation in iteration $\bar{j}'$.

In this paper, we assume that every variable in the program is written only once during the entire execution of the algorithm; therefore, there is no output dependence. To illustrate this idea, consider the following word-level matrix multiplication algorithm.

Example 2.1 (matrix multiplication). Consider the matrix multiplication $Z=X \bullet Y$ as follows.

$$
\begin{array}{l}
DO\ (j_1=1,\ u;\ j_2=1,\ u;\ j_3=1,\ u) \\
\quad z(j_1, j_2) = z(j_1, j_2) + x(j_1, j_3)\, y(j_3, j_2) \\
END
\end{array}
$$

Variable $z(j_1, j_2)$, $j_1, j_2 = 1, \ldots, u$, is written more than once during the execution of the algorithm. This program can be transformed to the following equivalent one where every variable is written only once.

$$
\begin{array}{l}
DO\ (j_1=1,\ u;\ j_2=1,\ u;\ j_3=1,\ u) \\
\quad z(j_1,j_2,j_3) = z(j_1,j_2,j_3-1) + x(j_1,j_3)\, y(j_3,j_2) \\
END
\end{array}
\tag{2.2}
$$

where $z(j_1, j_2, 0) = 0$, $j_1, j_2 = 1, \ldots, u$, and the $z_{i,j}$ entry of the product matrix is $z(i, j, u)$, $i, j = 1, \ldots, u$. □

In program (2.2), datum $x(j_1, j_3)$ is needed as an input by the n computations at index points $[j_1, 1, j_3]^T$, $[j_1, 2, j_3]^T$, ..., $[j_1, u, j_3]^T$. In other words, we need to broadcast datum $x(j_1, j_3)$ to these n index points if these n computations were to be executed simultaneously. Usually, broadcasting is not preferred in VLSI implementations because it incurs additional area on a chip and longer clock cycles. If we do not allow broadcasts, then data can be pipelined to those computations that need the data using the method developed by Fortes and Moldovan [2]. After eliminating broadcasts, program (2.2) can be transformed into the following form [2].

$$
\begin{array}{l}
DO\ (j_1=1,\ u;\ j_2=1,\ u;\ j_3=1,\ u) \\
\quad x(\bar{j}) = x(\bar{j}-[0,\ 1,\ 0]^T) \\
\quad y(\bar{j}) = y(\bar{j}-[1,\ 0,\ 0]^T) \\
\quad z(\bar{j}) = z(\bar{j}-[0,\ 0,\ 1]^T) + x(\bar{j})\, y(\bar{j}) \\
END
\end{array}
\tag{2.3}
$$

where $x(\bar{j}) = x(j_1, j_2, j_3)$, $y(\bar{j}) = y(j_1, j_2, j_3)$ and $z(\bar{j}) = z(j_1, j_2, j_3)$. Intuitively, data $x(j_1, j_3)$ are pipelined along the j_2 axis through index points $[j_1, 1, j_3]^T$, $[j_1, 2, j_3]^T$, ..., $[j_1, u, j_3]^T$. Similarly, $y(j_3, j_2)$ are pipelined along the j_1 axis. Initially, $x(j_1, 0, j_3) = x_{j_1, j_3}$ and $y(0, j_2, j_3) = y_{j_3, j_2}$. The dependence structure of the matrix multiplication algorithm in (2.3) can also be obtained by using Banerjee's technique [1].

المنارة للاستشارات
www.manaraa.com

Consider a dependence vector $\bar{d}$. If for any two arbitrary index vectors $\bar{j}_1, \bar{j}_2 \in J$ such that $\bar{j}_2 - \bar{j}_1 = \bar{d}$ and that dependence vector $\bar{d}$ is valid at $\bar{j}_2$, then this dependence vector is *uniform*. If all dependence vectors are uniform, then the algorithm is a *uniform dependence algorithm*. For the purpose of this paper, an algorithm can be characterized by a triplet (J, D, E) where J is the index set, D is the dependence matrix containing all distinct dependence vectors as its columns, and E contains all different computations in all iterations. For uniform dependence algorithms, because all dependence vectors are valid at every index point, it is not needed to specify the points they are valid at. As an example, the matrix multiplication algorithm in (2.3) is a uniform dependence algorithm because dependence vectors $\bar{d}_1$, $\bar{d}_2$ and $\bar{d}_3$ are uniform and can be characterized by the triplet A = (J, D, E) where

$$J = \left\{ \begin{bmatrix} j_1 \\ j_2 \\ j_3 \end{bmatrix} : 1 \le j_1, j_2, j_3 \le u, j_1, j_2, j_3 \in Z \right\} \quad D = \begin{matrix} y & x & z \\ \end{matrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

and

$$E = \{ x(\bar{j}) = x(\bar{j} - \bar{d}_2), y(\bar{j}) = y(\bar{j} - \bar{d}_1), \\ z(\bar{j}) = z(\bar{j} - \bar{d}_3) + x(\bar{j})\, y(\bar{j}) \} .$$

The symbol on the top of each column in D indicates the variable that causes the dependence. This algorithm in (2.3) is *computationally uniform* because all iterations have the same computation.

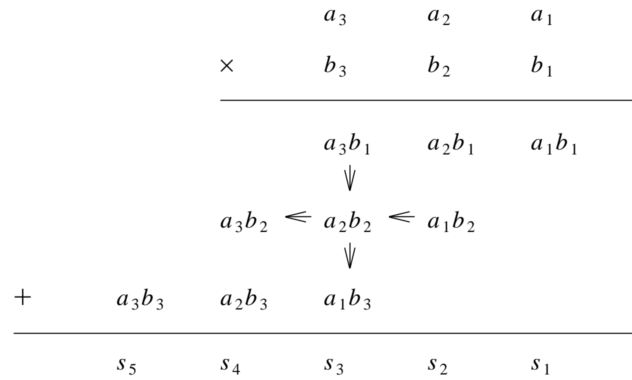
3. BIT-LEVEL DEPENDENCE ANALYSIS

This section presents our method for finding dependence structures of bit-level algorithms. Section 3.1 discusses dependence structures of arithmetic algorithms for the multiplication and addition of two integers. Section 3.2 shows two algorithm expansions for implementing word-wise operations by bit-wise operations. Finally, we show how to obtain the bit-level dependence structure directly from word-level dependence structures, dependence structures of arithmetic algorithms, and the corresponding algorithm expansion.

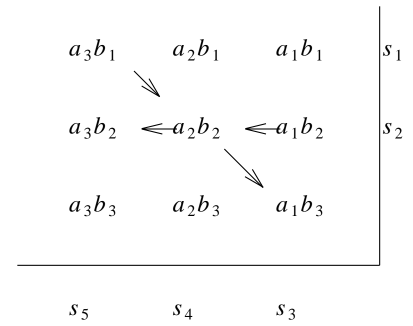
3.1. Dependence Structures of Arithmetic Algorithms

Consider the *add-shift* [3] arithmetic algorithm that multiplies two nonnegative integers $s = a \times b$, where $s = s_{2p-1}\, s_{2p-2} \ldots s_1$, $a = a_p\, a_{p-1} \ldots a_1$, and $b = b_p\, b_{p-1} \ldots b_1$. As is illustrated in Fig. 1a, the multiplication of two integers $s = a \times b$ can be implemented by adding p integers $(a_p \wedge b_i)(a_{p-1} \wedge b_i) \ldots (a_1 \wedge b_i)$, $i = 1, \ldots, p$, with the ith integer shifted $i-1$ positions to the left.

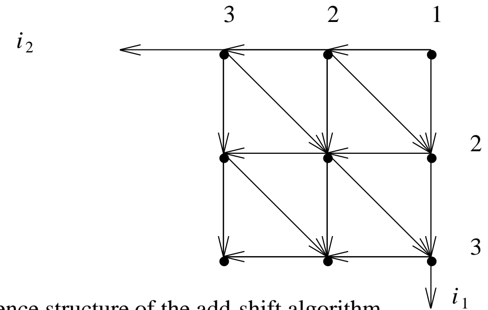
For instance, in computing point $a_2 b_2$ in Fig. 1a, one partial sum bit s and one carry bit c are to be produced using input variables a_2, b_2, the carry bit from the east (computing point $a_1 b_2$), and the partial sum bit from the north (computing point $a_3 b_1$). The output variables are the carry bit to be sent to the west (computing point $a_3 b_2$) and the partial sum bit to be sent to the south (computing point $a_1 b_3$).



(a) $s = a \times b$ implemented by the add-shift algorithm.



(b) The square obtained by reshaping the parallelogram in (a).



(c) Dependence structure of the add-shift algorithm.

Figure 1. The add-shift arithmetic algorithm.

If the parallelogram of the data distribution in Fig. 1a is reshaped to the square shown in Fig. 1b, then the operations performed are similar except that data are input and output in different directions. The two algorithms in Fig. 1a and 1b are equivalent. The general add-shift arithmetic algorithm as illustrated in Fig. 1b can be described by the following code.

$$\begin{aligned} &DO\ (i_1 = 1, p\, ; i_2 = 1, p) \\ &\quad c(\bar{i}) = g(a(i_2) \wedge b(i_1),\, c(i_1, i_2 - 1),\, s(i_1 - 1, i_2 + 1)) \\ &\quad s(\bar{i}) = f(a(i_2) \wedge b(i_1),\, c(i_1, i_2 - 1),\, s(i_1 - 1, i_2 + 1)) \\ &END \end{aligned} \tag{3.1}$$

www.manaraa.com

where $\bar{i} = [i_1, i_2]^T$, $c(\bar{i}) = c(i_1, i_2)$ is the carry bit, $s(\bar{i}) = s(i_1, i_2)$ is the partial sum bit, and Boolean functions g and f are defined as follows.

$$g(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_1)$$
$$f(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3 \,. \qquad (3.2)$$

Then initial values are $s(0, i_2) = 0$, $i_2 = 2, ..., p+1$, $s(i_1, p+1) = 0$, $i_1 = 1, ..., p-1$, and $c(i_1, 0) = 0$, $i_1 = 1, ..., p$. The final results are $s_i = s(i, 1)$ for $1 \le i \le p$, and $s_i = s(p, i-p+1)$ for $p < i \le 2p-1$.

Every variable in program (3.1) is written only once. Broadcasts exist because variable $a(i_2)$ is needed by index points $[1, i_2]^T$, ..., $[p, i_2]^T$, and $b(i_1)$ is needed by index points $[i_1, 1]^T$, ..., $[i_1, p]^T$. By applying Fortes and Moldovan's technique [2] for eliminating broadcasts, program (3.1) can be transformed into one without any broadcast as follows.

$$\begin{aligned}
&DO\ (i_1{=}1, p; i_2{=}1, p)\\
&\quad a(\bar{i}) = a(\bar{i}-\bar{\delta}_1)\\
&\quad b(\bar{i}) = b(\bar{i}-\bar{\delta}_2)\\
&\quad c(\bar{i}) = g(a(\bar{i}) \wedge b(\bar{i}),\ c(\bar{i}-\bar{\delta}_2),\ s(\bar{i}-\bar{\delta}_3))\\
&\quad s(\bar{i}) = f(a(\bar{i}) \wedge b(\bar{i}),\ c(\bar{i}-\bar{\delta}_2),\ s(\bar{i}-\bar{\delta}_3))\\
&END
\end{aligned} \qquad (3.3)$$

where $\delta_1 = [1,0]^T$, $\delta_2 = [0,1]^T$, and $\delta_3 = [1,-1]^T$.

In iteration $\bar{i}$, $a(\bar{i}-\bar{\delta}_1)$ produced in iteration $\bar{i}' = \bar{i}-\bar{\delta}_1$ is input to produce $a(\bar{i})$; hence iteration $\bar{i}$ depends on iteration $\bar{i}'$. This flow dependence is uniform and is described by the pair $(\bar{i}, \bar{\delta}_1)$, $\bar{i} \in J$. Similarly, for the statement generating $b(\bar{i})$, there is a uniform dependence described by the pair $(\bar{i}, \bar{\delta}_2)$, $\bar{i} \in J$. For statements generating $c(\bar{i})$ and $s(\bar{i})$ in iteration $\bar{i}$, variables $c(\bar{i}-\bar{\delta}_2)$ and $s(\bar{i}-\bar{\delta}_3)$ are needed as inputs, which are generated in iterations $\bar{i}' = \bar{i}-\bar{\delta}_2$ and $\bar{i}'' = \bar{i}-\bar{\delta}_3$, respectively. Hence there are two uniform dependences $(\bar{i}, \bar{\delta}_2)$ and $(\bar{i}, \bar{\delta}_3)$, $\bar{i} \in J$. Since all dependence vectors are uniform, this algorithm is a uniform dependence algorithm. In short, there are three distinct dependence vectors, and the add-shift algorithm in (3.3) is described by the triplet $A_{add-shift} = (J_{as}, D_{as}, E_{as})$ where

$$J_{as} = \{\bar{i}: 1 \le i_1, \text{ and } i_2 \le p,\ i_1, i_2 \in Z\}\,, \qquad (3.4)$$

$$D_{as} = [\bar{\delta}_1, \bar{\delta}_2, \bar{\delta}_3] = \begin{array}{c} \begin{matrix} a & b,c & s \end{matrix} \\ \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix} \end{array}$$

Note that δ_2 in D_{as} is involved in the computations of both $b(\bar{i})$ and $c(\bar{i})$.

Fig. 1c shows the index set and the dependence structure of the arithmetic algorithm in (3.3) when $p{=}3$. As an example, index point $[2, 2]^T$ represents the computation where a_2, b_2, carry $c(2,1)$ and partial sum $s(1, 3)$ are the four input bits. The computation is to sum three bits $a_2 \wedge b_2$, $c(2,1)$ and $s(1,3)$ to produce carry bit $c(2,2)$ to be sent to $[2, 3]^T$ and partial sum bit $s(2,2)$ to be sent to $[3,1]^T$. Dependence vector $[1,0]^T$ is due to pipelining a_2 from index point $[1, 2]^T$ and dependence vector $[0, 1]^T$ is due to pipelining b_2 from index point $[2, 1]^T$.

Due to space limitation, the dependence structure of an algorithm for adding two integers is not included here [7].

3.2. Algorithm Expansions and Bit-level Dependences

In this subsection we discuss algorithm expansions, namely, how to implement word-wise operations by bitwise operations. We study two expansions with the algorithm model in (2.1) restricted to the following form.

$$\begin{aligned}
&DO\ (j_1{=}l_1, u_1; j_2{=}l_2, u_2; ...; j_n{=}l_n, u_n)\\
&\quad x(\bar{j}) = x(\bar{j}-\bar{h}_1)\\
&\quad y(\bar{j}) = y(\bar{j}-\bar{h}_2)\\
&\quad z(\bar{j}) = z(\bar{j}-\bar{h}_3) + x(\bar{j}) \bullet y(\bar{j})\\
&END
\end{aligned} \qquad (3.5)$$

The triplet describing this word-level program is (J_w, D_w, E_w) where

$$D_w = \begin{bmatrix} \overset{x}{\bar{h}_1}, & \overset{y}{\bar{h}_2}, & \overset{z}{\bar{h}_3} \end{bmatrix} \quad J_w = \{\bar{j}: l_i \le j_i \le u_i, j_i \in Z, i=1,...,n\}. \qquad (3.6)$$

This model can describe applications such as matrix multiplication, convolution, matrix-vector multiplication, discrete cosine transform, and discrete Fourier transform. More general models are under investigation currently.

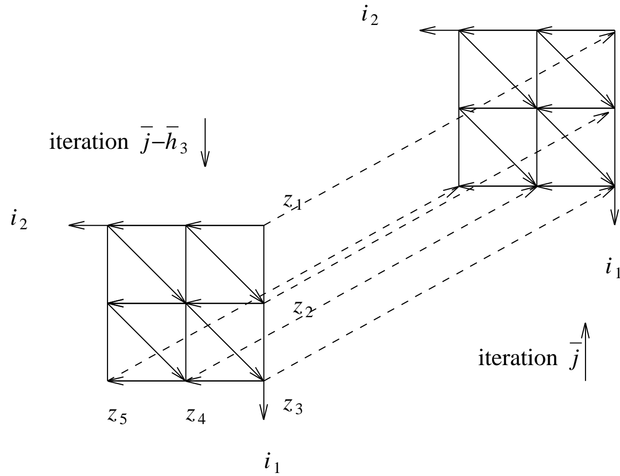
In each iteration of (3.5), two numbers $x(\bar{j})$ and $y(\bar{j})$ are multiplied, and the result added to the number $z(\bar{j}-\bar{h}_3)$ computed at index point $\bar{j}-\bar{h}_3$. Consider that each index point in J_w is replaced by a 2-dimensional index set J_{as} in Fig. 1c for multiplying $x(\bar{j})$ and $y(\bar{j})$. Then each index point $[\bar{j}^T, i_1, i_2]^T$ has $n+2$ indexes.

Fig. 2 illustrates two ways to add $z(\bar{j}-\bar{h}_3)$ to $x(\bar{j}) \bullet y(\bar{j})$. Let $x(\bar{j}) = x_p x_{p-1} ... x_1$, $y(\bar{j}) = y_p y_{p-1} ... y_1$ and $z(\bar{j}-\bar{h}_3) = z_{2p-1} z_{2p-2} ... z_1$. As is shown in Section 3.1, z_{2p-1}, z_{2p-2}, ..., z_1 are produced at boundary points $[(\bar{j}-\bar{h}_3)^T, i_1, i_2]^T$ where $i_1 = p$ or $i_2 = 1$. As is illustrated in Fig. 2a (Expansion II), the $2p-1$ bits of $z(\bar{j}-\bar{h}_3)$ are added to the product $x(\bar{j}) \bullet y(\bar{j})$ at boundary index points in iteration $\bar{j}$ where $i_1 = p$ or $i_2 = 1$; *i.e.*, at points $[\bar{j}^T, p, i_2]^T$, $i_2 = 1, ..., p$ and $[\bar{j}^T, i_1, 1]^T$, $i_1 = 1, ..., p-1$. As an example, at index point $[\bar{j}^T, p, 2]^T$, the bits that need to be summed are the partial sum bit from $[\bar{j}^T, p-1, 3]^T$, z_{p+1}, $x_2 \wedge y_p$, and the carry bit from index point $[\bar{j}^T, p, 1]^T$.

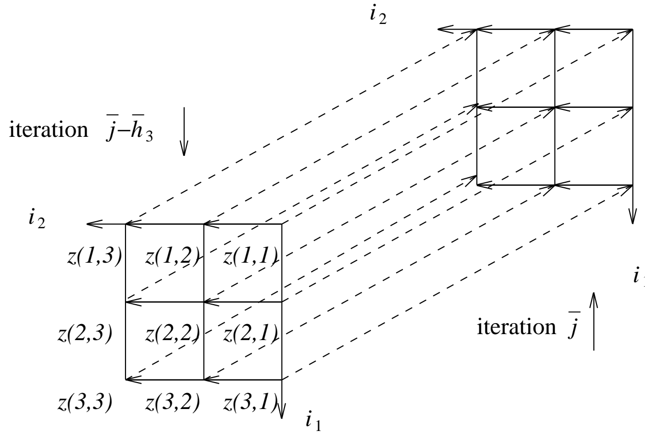
In reality, we do not have to add z_{2p-1}, ..., z_1 of $z(\bar{j}-\bar{h}_3)$ in order to produce $x(\bar{j}) \bullet y(\bar{j})$ as is shown in Fig. 2a. Instead, as is illustrated in Fig. 2b (Expansion I), we can add the partial sum bits $z(\bar{j}-\bar{h}_3, i_1, i_2)$, $i_1, i_2{=}1$, ..., p, of $z(\bar{j}-\bar{h}_3)$ generated at $[(\bar{j}-\bar{h}_3)^T, i_1, i_2]^T$ to $x(\bar{j}) \bullet y(\bar{j})$ at $[\bar{j}^T, i_1, i_2]^T$. In other words, partial sum bit $z(\bar{j}-\bar{h}_3, i_1, i_2)$ is not sent to $[(\bar{j}-\bar{h}_3)^T, i_1{+}1, i_2{-}1]^T$, but is sent to $[\bar{j}^T, i_1, i_2]^T$ instead.

As an example, let $p{=}3$ as is shown in Fig. 2b. At index point $[\bar{j}^T, 2, 2]^T$, three bits, the carry bit from index point $[\bar{j}^T, 2, 1]^T$, $z(\bar{j}-\bar{h}_3, 2, 2)$ from iteration $\bar{j}-\bar{h}_3$, and $x_2 \wedge y_2$ are summed to produce a new partial sum bit $z(\bar{j}, 2, 2)$ to be sent to

www.manaraa.com



(a) Expansion II: add z_5 ..., z_1 at southern and eastern boundaries.



(b) Expansion I: add partial sum bits of $z(j-h_3)$.

Figure 2: Two ways to add $z(j-h_3)$ to product $x(j) \cdot y(j)$.

index point $[(\bar{j}+\bar{h}_3)^T, 2,2]^T$, and a new carry bit $c(\bar{j},2,2)$ to be sent to index point $[\bar{j}^T, 2,3]^T$.

In short, each of the two ways described above for adding $z(\bar{j}-\bar{h}_3)$ to $x(\bar{j}) \cdot y(\bar{j})$ corresponds to an algorithm expansion for implementing word-wise operations by bit-wise operations. To get more insight on the dependence structures of expanded bit-level algorithms, consider the following simple 1-dimensional algorithm.

$$
\begin{aligned}
&DO\ (j=l,\ u) \\
&\quad x(j) = x(j-h_1) \\
&\quad y(j) = y(j-h_2) \\
&\quad z(j) = z(j-h_3) + x(j) \cdot y(j) \\
&END
\end{aligned}
\tag{3.7}
$$

Since this is 1-dimensional, j, h_1, h_2, and h_3 are scalars instead of vectors. The index set and the dependence structure of this 1-dimensional word-level algorithm in shown in Fig. 3a. Without loss of generality, we assume that $h_1 = h_2 = h_3$.

Fig. 3b shows the dependence structure where the p^2 partial sum bits of $z(j-h_3)$ are added to the corresponding partial sum bits of $x(j) \cdot y(j)$ (corresponding to Expansion II in Fig. 2b). Fig. 3c shows the dependence structure of the bit-level algorithm where the $2p-1$ final sum bits of $z(j-h_3)$ are added to $x(j) \cdot y(j)$ at boundary index points where $i_1 = p$ or $i_2 = 1$ (corresponding to Expansion I in Fig. 2a).

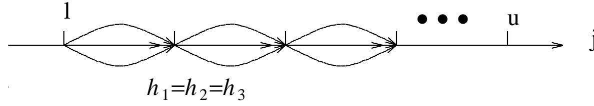
In Fig. 3b, each index point in Fig. 3a is replaced by the 2-dimensional index set for the multiplication of two integers in Fig. 1c. Hence, each index point in Fig. 3b has three indexes j, i_1, and i_2. The p bits of $x(j)$ are pipelined along the j axis at index points where $i_1 = 1$. In other words, the dependence caused by pipelining x_k, the kth bit of $x(j)$, is valid only at index points where $i_1 = 1$ and $i_2 = k$ and is, therefore, not uniform. All bits of $x(j)$ are also pipelined along axis i_1 in order to compute $x(j) \cdot y(j)$. The dependence caused by pipelining along direction j is described by the pair $([j, 1, i_2]^T, \bar{d}_1 = [h_1, 0, 0]^T)$, where $l \le j \le u$, and $1 \le i_2 \le p$. On the other hand, the dependence caused by pipelining along direction i_1 can be described by the pair $([j, i_1, i_2]^T, \bar{d}_4)$ where $i_1 \ne 1$ and $\bar{d}_4 = [0, 1, 0]^T$ (*i.e.*, dependence vector $\bar{\delta}_1$ in (3.4) is prefixed by a zero corresponding to the j axis). Dependence vector $\bar{d}_4$ is not uniform and is valid only at index points where $i_1 \ne 1$.

Similarly, the p bits of $y(j)$ are pipelined along axis j at index points where $i_2 = 1$, and the dependence caused by pipelining y_k, the kth bit of $y(j)$, is valid only at boundary index points where $i_1 = k$ and $i_2 = 1$. This dependence is, therefore, not uniform. All bits of $y(j)$ are also pipelined along axis i_2 in order to compute $x(j) \cdot y(j)$. The dependence caused by pipelining along direction j is described by the pair $([j, i_1, 1]^T, \bar{d}_2 = [h_2, 0, 0]^T)$, where $l \le j \le u$, and $1 \le i_1 \le p$. The dependence caused by pipelining along direction i_2 can be described the pair $([j, i_1, i_2]^T, \bar{d}_5)$, where $i_2 \ne 1$ and $\bar{d}_5 = [0, 0, 1]^T$ (*i.e.*, $\bar{\delta}_2$ in (3.4) is prefixed by a zero corresponding to axis j). Dependence vector $\bar{d}_5$ is not uniform and is valid only at points where $i_2 \ne 1$.

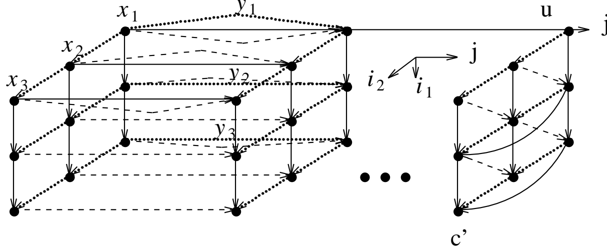
Variable $z(j-h_3)$ causes a flow dependence because it is to be added to $x(j) \cdot y(j)$ after it is generated. In Fig. 3b, because the p^2 partial sum bits $z(j-h_3, i_1, i_2)$, $i_1, i_2 = 1, \dots p$, of $z(j-h_3)$ produced at index points $[j-h_3, i_1, i_2]^T$, $i_1, i_2 = 1, \dots, p$, are sent to index points $[j, i_1, i_2]^T$, $i_1, i_2 = 1, \dots, p$, respectively, the flow dependence is described by the pair $(\bar{q}, \bar{d}_3)$ where $\bar{q} = [j, i_1, i_2]^T$, $i_1, i_2 = 1, \dots, p$, $j = l, \dots, u$, and $\bar{d}_3 = [h_3, 0, 0]^T$. This dependence is uniform. At index points where $j=u$, all partial sum bits have to be added. Partial sum bit $z(u, i_1, i_2)$ generated at index point $[u, i_1, i_2]^T$ will be sent to index points $[u, i_1+1, i_2-1]^T$ instead of index point $[j+h_3, i_1, i_2]^T$ as is in the cases when $j \ne u$. This flow dependence can be described by the pair $([u, i_1, i_2]^T, \bar{d}_6 = [0, 1, -1]^T)$ (where $\bar{d}_6$ is obtained from $\bar{\delta}_3$ in (3.4) by prefixing it by a zero corresponding to j axis). This dependence is not uniform and is valid only when $j = u$.

Carry bit c flows in parallel along with the i_2 axis. The corresponding dependence vector $\bar{d}_5 = [0, 0, 1]^T$ is not uniform

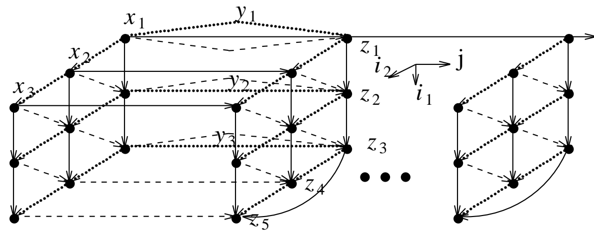
المنارة للاستشارات
www.manaraa.com



(a) Index set and dependence structure of program (3.7).



(b) Dependence structure using expansion I.



(c) Dependence structure using expansion II.

Figure 3. In (b) and (c) , x is pipelined along solid lines; y is pipelined along dotted lines and z flows along dashed lines.

---

and is not valid at index points where $i_2 = 1$ because at these points, carry c is zero and is not added. Therefore, the pair describing the dependence caused by c is $([j, i_1, i_2]^T, \bar{d}_5)$, $i_2 \neq 1$, which is the same as the one describing pipelining $y(j)$ along the i_2 axis.

In Fig. 3b, three bits are summed at some index points. In these cases, one carry bit and one partial sum bit have to be generated. At other index points, more than three bits have to be summed; hence, we need to generate at least two carry bits and one partial sum bit. For example, at index point $[u, 2, 2]^T$, we need to sum at least four bits: $x_2 \wedge y_2$, two partial sum bits $z(u-1, 2, 2)$ and $z(u, 1, 3)$, and carry bit $c(u, 2, 1)$. Three output bits should be generated: the partial sum bit $z(u, 2, 2)$ to be sent to index point $[u, 3, 1]^T$, and two carry bits $c(u, 2, 2)$ to be sent to index point $[u, 2, 3]^T$ and $c'(u, 2, 2)$ to be sent to index point $[u, 2, 4]^T$ (if $p = 3$, this carry goes out of the index set and is not useful; however, if $p > 3$, this carry is useful). If four of these input bits are one, carry c' will be one. If two and not more than three are ones, then carry c will be one. The dependence vector corresponding to the second carry c' is $\bar{d}_7 = [0, 0, 2]^T$, which is valid at index points where $j = u$ and $i_2 \neq 1, 2$, or $j = u$ and $i_1 \neq 1$.

In Fig. 3c (corresponding to Expansion I in Fig. 2a), all the dependence vectors can be obtained in a similar way. The two bit-level dependence structures in Fig. 3b and 3c have the same index set. Let J, D_I, and D_{II} be the index set, the dependence matrix for Fig. 3b, and the dependence matrix for Fig. 3c, respectively. Then

$$J = \{[j, i_1, i_2]^T : l \le j \le u,\ 1 \le i_1, i_2 \le p,\ j, i_1, i_2 \in Z\}$$

$$D_I = \left[\bar{d}_1, \bar{d}_2, \bar{d}_3, \bar{d}_4, \bar{d}_5, \bar{d}_6, \bar{d}_7\right] \tag{3.8}$$

$$= \begin{matrix} x & y & z & x & y,c & z & c' \\ \end{matrix}$$
$$= \begin{bmatrix} h_1 & h_2 & h_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 2 \end{bmatrix}$$
$$\begin{matrix} i_1{=}1 & i_2{=}1 & \bar{q} & i_1{\neq}1 & i_2{\neq}1 & j{=}u & \bar{q}_1 \end{matrix}$$

$$D_{II} = \left[\bar{d}_1, \bar{d}_2, \bar{d}_3, \bar{d}_4, \bar{d}_5, \bar{d}_6, \bar{d}_7\right] \tag{3.9}$$

$$\begin{matrix} x & y & z & x & y,c & z & c' \\ \end{matrix}$$
$$= \begin{bmatrix} h_1 & h_2 & h_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 2 \end{bmatrix}$$
$$\begin{matrix} i_1{=}1 & i_2{=}1 & \bar{q}_2 & i_1{\neq}1 & i_2{\neq}1 & \bar{q} & i_1{\neq}p \end{matrix}$$

where $\bar{q}_1 = [u, i_1, i_2]^T$, $i_1 \neq 1$ or $i_2 \neq 1, 2$, $\bar{q}_2 = [j, i_1, i_2]^T$, $i_1 = p$, or $i_2 = 1$, and $\bar{q}$ is an arbitrary index point in J. The information at the bottom indicates those index points where the corresponding dependence vector is valid at. For example, $\bar{d}_1$ is valid only at points where $i_1 = 1$ as is indicated by "$i_1 = 1$" at the bottom of $\bar{d}_1$. Vector $\bar{d}_3$ is uniform in Expansion I and $\bar{d}_6$ is uniform in Expansion II.

We now extend the above analysis on 1-dimensional algorithms to n-dimensional algorithms in (3.5). Let

$$\bar{q} = \begin{bmatrix} \bar{j} \\ \bar{i} \end{bmatrix} \text{ where } \bar{j} = [j_1, \ldots, j_n]^T \text{ and } \bar{i} = [i_1, i_2]^T, \tag{3.10}$$

and $\bar{d}_1 = [\bar{h}_1^T, 0, 0]^T$, $\bar{d}_2 = [\bar{h}_2^T, 0, 0]^T$, $\bar{d}_3 = [\bar{h}_3^T, 0, 0]^T$, $\bar{d}_4 = [\bar{0}, \bar{\delta}_1^{\,T}]^T$, $\bar{d}_5 = [\bar{0}, \bar{\delta}_2^{\,T}]^T$, $\bar{d}_6 = [\bar{0}, \bar{\delta}_3^{\,T}]^T$, and $\bar{d}_7 = [\bar{0}, 0, 2]^T$ where $\bar{h}_1$, $\bar{h}_2$ and $\bar{h}_3$ are as defined in (3.5) and $\bar{\delta}_1$, $\bar{\delta}_2$ and $\bar{\delta}_3$ are as defined in (3.4). Also let D_{as} and D_w be as defined in (3.4) and (3.6), respectively. Let J and D_I and D_{II} be the index set and dependence matrices of Expansions I and II described in Fig. 3b and 3c, respectively. Then the following theorem describes how D_I, D_{II} and J are related to the word-level dependence structure, the arithmetic algorithm, and algorithm expansions.

Theorem 3.1:

$$J = \left\{ \bar{q} = \begin{bmatrix} \bar{j} \\ \bar{i} \end{bmatrix} : \bar{j} \in J_w,\ \bar{i} \in J_{as} \right\}, \tag{3.11a}$$

$$D_I = \begin{bmatrix} D_w & \mathbf{0} & \bar{0} \\ \mathbf{0} & D_{as} & \bar{\delta}_4 \end{bmatrix} = \left[\bar{d}_1\, \bar{d}_2\, \bar{d}_3\, \bar{d}_4\, \bar{d}_5\, \bar{d}_6\, \bar{d}_7\right] \tag{3.11b}$$

$$\begin{matrix} x & y & z & x & y,c & z & c' \\ \end{matrix}$$
$$= \begin{bmatrix} \bar{h}_1 & \bar{h}_2 & \bar{h}_3 & \bar{0} & \bar{0} & \bar{0} & \bar{0} \\ \bar{0} & \bar{0} & \bar{0} & \bar{\delta}_1 & \bar{\delta}_2 & \bar{\delta}_3 & \begin{matrix} 0 \\ 2 \end{matrix} \end{bmatrix}$$
$$\begin{matrix} i_1{=}1 & i_2{=}1 & \bar{q} & i_1{\neq}1 & i_2{\neq}1 & j_n{=}u_n & \bar{q}_1 \end{matrix}$$

المنارة للاستشارات
www.manaraa.com

$$D_{II} = \begin{bmatrix} D_w \mathbf{0} & \bar{0} \\ \mathbf{0} & D_{as} \bar{\delta}_4 \end{bmatrix} = \begin{bmatrix} \bar{d}_1 & \bar{d}_2 & \bar{d}_3 & \bar{d}_4 & \bar{d}_5 & \bar{d}_6 & \bar{d}_7 \end{bmatrix} \quad (3.11c)$$

$$= \begin{array}{c} \\ \\ \\ \end{array}\begin{array}{ccccccc} x & y & z & x & y,c & z & c' \\ \left[\bar{h}_1\right. & \bar{h}_2 & \bar{h}_3 & \bar{0} & \bar{0} & \bar{0} & \left.\bar{0}\right] \\ \left[\bar{0}\right. & \bar{0} & \bar{0} & \bar{\delta}_1 & \bar{\delta}_2 & \bar{\delta}_3 & \left.\begin{matrix}0\\2\end{matrix}\right] \\ i_1{=}1 & i_2{=}1 & \bar{q}_2 & i_1{\neq}1 & i_2{\neq}1 & \bar{q} & i_1{=}p \end{array}$$

where $\bar{q}_1$ is defined such that ($i_1{\neq}1$ or $i_2{\neq}1,2$) and $j_n{=}u_n$, $\bar{q}_2$ is defined such that $i_1{=}p$ or $i_2{=}1$, $\bar{q}$ is an arbitrary point in J, $\bar{\delta}_4$ = $[0, 2]^T$, and $\mathbf{0}$ is a matrix with proper dimension and has all zero entries.

Proof. The proof is omitted due to space limitation [7]. □

Example 3.1. Consider the matrix multiplication in (2.3). Its dependence matrix and index set at the word-level is shown in (2.4). According to Theorem 3.1, the dependence matrix and index set of the corresponding bit-level matrix multiplication algorithm derived by Expansion II (Fig. 3c) are as follows.

$$D = \begin{array}{ccccccc} y & x & z & x & y,c & z & c' \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 2 \\ i_1{=}1 & i_2{=}1 & \begin{matrix} i_1{=}p \\ \text{or} \\ i_2{=}1 \end{matrix} & i_1{\neq}1 & i_2{\neq}1 & \bar{q} & i_1{=}p \end{array} \quad (3.12)$$

$$J{=}\{[j_1,j_2,j_3,i_1,i_2]^T{\in}Z^5{:}1{\le}j_1,j_2,j_3{\le}u,\ 1{\le}i_1,i_2{\le}p\} \quad (3.13)$$

Expansion II is slower than Expansion I because, as is indicated in Fig 3c, the computation at $\bar{j}$ has to wait for the final results at $\bar{j}{-}\bar{h}_3$. In Expansion I, partial sum bits in $\bar{j}{-}\bar{h}_3$ are sent to $\bar{j}$ and takes less time. Further, Expansion I is more computationally uniform because at all points, except when $j = u$ in Fig 3b, at most three bits are to be summed; in contrast, in Expansion II, four or five bits have to be summed on the hyperplane $i_1 = p$ (the southern boundary points in the 2-dimensional index set of the add-shift operation). This may cause unbalanced load distribution. More discussions on algorithm expansion can be found in the reference [7].

4. DESIGN OF BIT-LEVEL ARCHITECTURES

Based on the dependence structures presented in Section 3, we discuss in this section the design of bit-level architectures. After summarizing in Section 4.1 a design method we have developed earlier [5,6], we show in Section 4.2 the application of the design method to design two bit-level architectures for matrix multiplication.

4.1. Design Method

Definition 4.1 (Linear algorithm transformation): A *linear algorithm transformation* maps an n-dimensional algorithm (J, D, E) into a $(k-1)$-dimensional processor array according to the mapping:

$$\tau\colon J \rightarrow Z^k,\ \tau(\bar{j}) = T\bar{j},\ \forall \bar{j} \in J$$

where $T = \begin{bmatrix} S \\ \Pi \end{bmatrix} \in Z^{k \times n}$ is the *mapping matrix*, $S \in Z^{(k-1)\times n}$ is the *space mapping matrix*, and $\Pi \in Z^{1\times n}$ is the *time mapping vector* or *linear schedule vector*. The computation indexed by $\bar{j} \in J$ is executed at time $\Pi\bar{j}$ and at processor $S\bar{j}$. The mapping τ must satisfy the following conditions:

(1) $\Pi D > \bar{0}$.

(2) $SD = PK$ where $P \in Z^{(k-1)\times r}$ is the matrix of interconnection primitives of the target machine, and $K \in Z^{r\times m}$ is such that

$$\sum_{j=1}^{r} k_{ji} \le \Pi\, \bar{d}_i\,, \quad i=1, \ldots, m. \quad (4.1)$$

(3) $\forall \bar{j}_1, \bar{j}_2 \in J$, if $\bar{j}_1 \neq \bar{j}_2$, then $\tau(\bar{j}_1) \neq \tau(\bar{j}_2)$, or $T\bar{j}_1 \neq T\bar{j}_2$.

(4) The rank of T is equal to k, or $rank(T) = k$.

(5) The entries of T are relatively prime.

Condition 1 in Definition 4.1 preserves the partial ordering induced by the dependence vectors. If this condition is satisfied, then the computation indexed by $\bar{j} {\in} J$ is scheduled to execute only after the computations indexed by $\bar{j}{-}\bar{d}_i \in J$, $i = 1, \ldots, m$. In this case $\Pi D > \bar{0}$ and the dependence relation is, therefore, satisfied.

The matrix of interconnection primitives P describes the connection links of processors in the processor array. For an array with each processor connected to its four nearest eastern, southern, western and northern neighbors, it has four interconnection primitives $[0, 1]^T$, $[0, -1]^T$, $[1, 0]^T$ and $[-1, 0]^T$ and matrix $P = \begin{bmatrix} 0 & 0 & 1 & -1 \\ 1 & -1 & 0 & 0 \end{bmatrix}$. Condition 2 in Definition 4.1 guarantees that the space mapping can be implemented in a systolic architecture with interconnection primitive matrix P. The summation on the left hand side of inequality (4.1) is the number of times that the interconnection primitives have been used to pass a datum according to dependence vector $\bar{d}_i$ from its source to its destination. The item on the right is the time units between the source usage and the destination usage of that datum. Assuming that it takes one time unit for a datum to travel one interconnection primitive, then the inequality must be satisfied in order to have the datum arrive before it is used.

Condition 3 defines the condition for avoiding computational conflicts. If the condition were not true, that is, $\tau(\bar{j}_1) = \tau(\bar{j}_2)$, then the computations indexed by $\bar{j}_1$ and $\bar{j}_2$ are mapped to the same processor at the same time, and a conflict occurs. Condition 4 guarantees that the algorithm is to be mapped into a $(k-1)$-dimensional array but not a q-dimensional array, where $q < k-1$. When $rank(T) = q+1 < k$, there are exactly $q+1$ linearly independent rows in T, and all other rows of T are linear

المنارة للاستشارات
www.manaraa.com

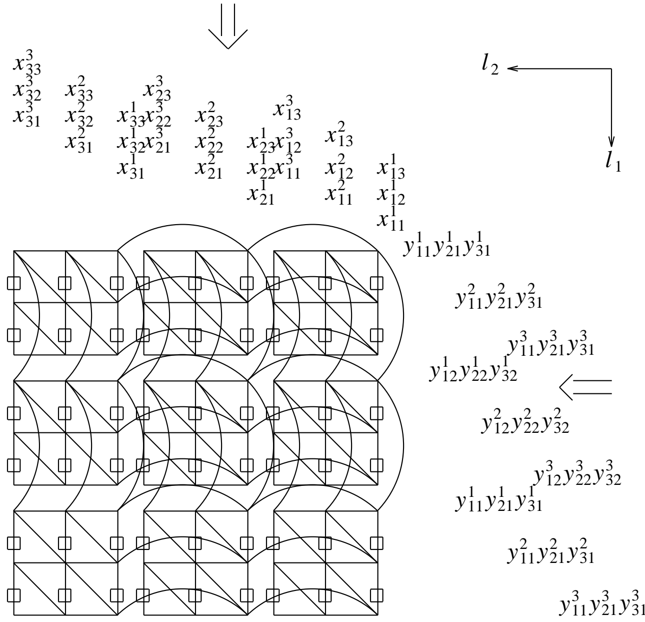


Figure 4. A bit-level processor array for matrix multiplication with $p=n=3$ corresponding to T in (4.2).

combinations of these $q+1$ linearly independent rows. Let T' be the matrix consisting of these $q+1$ linearly independent rows. T can now be transformed by a linear transformation to T', which means that the algorithm is actually mapped into a q-dimensional processor array. Condition 5 is used to guarantee that at any time during the execution, at least one processor is busy. For detailed description of the mapping model and optimization method, please see the references [5,6].

4.2. Bit-Level Architectures for the Matrix Multiplication

In this subsection, based on the dependence structure obtained by Expansion II, two bit-level architectures for matrix multiplication are presented. The dependence matrix and index set of the bit-level matrix multiplication by Expansion II are shown in (3.12) and (3.13), respectively

Theorem 4.5: The following mapping matrix

$$T = \begin{bmatrix} S \\ \Pi \end{bmatrix} = \begin{bmatrix} p & 0 & 0 & 1 & 0 \\ 0 & p & 0 & 0 & 1 \\ 1 & 1 & 1 & 2 & 1 \end{bmatrix} \tag{4.2}$$

is both feasible and time optimal.

Proof: The proof is omitted due to space limitation [7]. □

As is discussed in the reference [7], one feasible matrix of interconnection primitives and the corresponding K matrix are

$$P = \begin{bmatrix} p & 0 & 0 & 1 & 0 & 1 \\ 0 & p & 0 & 0 & 1 & -1 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.3}$$

The architecture described by mapping matrix T in (4.2) is shown in Fig. 4, where P and K are used and $p=u=3$. The timing and connections are specified completely by

$$TD = \begin{bmatrix} y & x & z & x & y,c & z & c' \\ p & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & p & 0 & 0 & 1 & -1 & 2 \\ 1 & 1 & 1 & 2 & 1 & 1 & 2 \end{bmatrix} \tag{4.4}$$

In Fig. 4, item x_{ij}^k (y_{ij}^k) represents the kth bit of x_{ij} (y_{ij}), Data x_{ij} flow from top to bottom and y_{ij} flow from right to left. Data z_{ij} are stationary and the final results are stored at the eastern and southern boundary points of each small block. There is a buffer on the interconnection primitive $[1, 0]^T$ because $S\bar{d}_4 = [1, 0]^T$ and $\sum_{t=1}^{6} k_{t4} = 1 < \Pi\bar{d}_4 = 2$. Note that interconnection primitives $[p, 0]^T$ and $[0, p]^T$ require long wiring with distance p.

The total execution time required by mapping matrix T in (4.2) is [6]

$$\begin{aligned} t &= \max\{\Pi(\bar{q}_1 - \bar{q}_2) : \bar{q}_1, \bar{q}_2 \in J\} + 1 \\ &= [1, 1, 1, 2, 1]([u,u,u,p,p]^T - [1,1,1,1,1]^T) + 1 \\ &= 3(u-1) + 3(p-1) + 1\,. \end{aligned} \tag{4.5}$$

The total number of processors required is

$$s = |\{\bar{l} : S\bar{q} = \bar{l},\ \bar{q} \in J\}| = u^2 p^2\,.$$

Consider another mapping matrix

$$T' = \begin{bmatrix} S \\ \Pi' \end{bmatrix} = \begin{bmatrix} p & 0 & 0 & 1 & 0 \\ 0 & p & 0 & 0 & 1 \\ p & p & 1 & 2 & 1 \end{bmatrix}. \tag{4.6}$$

Let

$$P' = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & -1 & 0 \end{bmatrix} \text{ and } K' = \begin{bmatrix} p & 0 & 1 & 0 & 0 & 0 \\ 0 & p & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4.7}$$

Then, $SD = P'K'$ and $\Pi'\bar{d}_i \geq \sum_{t=1}^{4} k'_{ti}$, $i = 1, \ldots, 7$. It can be shown similarly that T' is feasible; the corresponding architecture is shown in Fig. 5. The total execution time in this case is

$$\begin{aligned} t' &= \max\{\Pi'(\bar{q}_1 - \bar{q}_2) : \bar{q}_1, \bar{q}_2 \in J\} + 1 \\ &= [p, p, 1, 2, 1]([u,u,u,p,p]^T - [1,1,1,1,1]^T) + 1 \end{aligned} \tag{4.8}$$

المنارة للاستشارات
www.manaraa.com

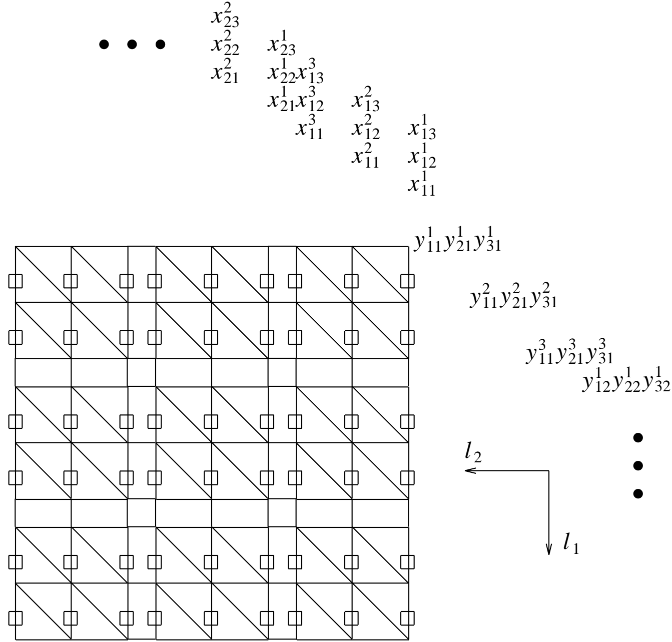


Figure 5. A bit-level architecture for matrix multiplication described by (4.6).

$$= (2p-1)(u-1)+3(p-1)+1$$

and the total number of processors is $(u\bullet p)^2$.

The main disadvantage of the design in Fig. 5 is that its total execution time is longer than that in Fig. 4. In this case, data x_{ij} and y_{ij} are pipelined at one speed instead of two different speeds as is shown in Fig. 4. However, long wires are not needed in Fig. 5, whereas in Fig. 4, long wires are used to pipeline x_{ij} and y_{ij} at different speeds.

We can compare the time optimal bit-level architecture in Fig. 4 with the best word-level architecture for matrix multiplication described in the literature [4]. The total execution time of the best word-level architecture for matrix multiplication with index set $J = \{[j_1, j_2, j_3]^T\colon 1 \le j_i \le u,\ j_i \in Z,\ i=1,2,3\}$ is $(3(u-1)+1)\bullet t_b$, where t_b is the time for multiplying two integers and adding two integers. Suppose the add-shift arithmetic algorithm is used to multiply two integers, then the multiplication time is $O(p^2)$, and the total time for word-level matrix multiplication is $O(p^2)\bullet(3(u-1)+1)$. If this number is compared with the total execution time in (4.5), then the speedup of our bit-level architecture over the word-level architecture described above is $O(p^2)$ if $u > p$ is assumed. Intuitively, a bit-level architecture is faster because, unlike in word-level architectures, a bit does not have to wait for all other bits to finish before going to next computation. In practice, faster arithmetic algorithms such as carry-save multiplication with complexity $t_b = O(p)$ can be used to multiply two integers. In this case the speedup of our bit-level architecture is $O(p)$.

5. CONCLUSIONS

In this paper, we propose to express the dependence structure of a bit-level algorithm as a function of its corresponding word-level dependence structure, the dependence structure of the arithmetic algorithm, and its algorithm expansion. Bit-level dependence structures can be obtained automatically from the above three parameters without using time consuming general dependence analysis methods. Based on the bit-level dependence structure, we illustrate the the design of two bit-level architectures for matrix multiplication. Our design shows that an optimal bit-level architecture can be $O(p)$ times faster than the corresponding word-level architecture, where p is the number of bits.

6. ACKNOWLEDGEMENTS

The authors would like to thank the students in the CMPS 639 course at USL for their helpful discussions. The authors are also indebted to L. Xu for suggesting the space mapping matrix S in (4.2), and to Z. Chen for drawing all the figures.

7. REFERENCES

[1] U. Banerjee. *Dependence Analysis for Supercomputing*, Kluwer Academic, 1988.

[2] J. A. B Fortes and D. I. Moldovan, "Data Broadcasting in Linearly Scheduled Array Processors," *Proc. 11th Annual Symp. on Computer Architecture*, 1984, pp. 133-149.

[3] K. Hwang, *Computer Arithmetic: Principles, Architecture, and Design,* John Wiley, New York, 1979.

[4] G.-J. Li and B. W. Wah, "The Design of Optimal Systolic Arrays," *IEEE Trans. Computers*, Vol. C-34, Jan. 1985, pp. 66-77.

[5] Z. Yang, W. Shang and J. A. B. Fortes, ‘‘Conflict-Free Scheduling of Nested Loop Algorithms on Lower Dimensional Processor Arrays,’’ *Proc. 6th IEEE Int’l Parallel Processing Symposium*, March 1992, Beverly Hills, CA, pp. 156-164.

[6] W. Shang and J. A. B. Fortes, ‘‘On Mapping of Uniform Dependence Algorithms into Lower Dimensional Processor Arrays,’’ *IEEE Trans. on Parallel and Distributed Systems*, Vol. 3, No. 3, May 1992, pp. 350-363.

[7] W. Shang and B. W. Wah, *Dependence Analysis and Architecture Design for Bit-Level Algorithms*, Technical Report 93-3-1, The Center for Advanced Computer Studies, Univ. of SW Louisiana, Lafayette, LA 70504, April, 1993.

[8] V. E. Taylor and J. A. B. Fortes, "Using RAB to Map Algorithms into Bit-level Systolic Arrays," *Proc. of Int’l Conf. on Supercomputing*, May 1987.

[9] Z. Xing and W. Shang, *Polynomial and Exact Data Dependence Analysis*, Technical Report 92-3-5, The Center for Advanced Computer Studies, Univ. of SW Louisiana, Lafayette, LA 70504, Dec., 1992.

[10] K. Ganapathy and B. W. Wah, "Synthesizing Optimal Lower Dimensional Processor Arrays," *Proc. Int’l Conf. on Parallel Processing*, Aug. 1992, pp. 96-103.

المنارة للاستشارات

www.manaraa.com