The Journal of VLSI Signal Processing—Systems for Signal, Image, and Video Technology, 19, 51–77 (1998)

A Systolic Design Methodology with Application to Full-Search Block-Matching Architectures

YEN-KUANG CHEN AND S.Y. KUNG
Princeton University

Received May 21, 1997; Revised November 5, 1997

Abstract. We present a systematic methodology to support the design tradeoffs of array processors in several emerging issues, such as (1) high performance and high flexibility, (2) low cost, low power, (3) efficient memory usage, and (4) system-on-a-chip or the ease of system integration. This methodology is algebraic based, so it can cope with high-dimensional data dependence. The methodology consists of some transformation rules of data dependency graphs for facilitating flexible array designs. For example, two common partitioning approaches, LPGS and LSGP, could be unified under the methodology. It supports the design of high-speed and massively parallel processor arrays with efficient memory usage. More specifically, it leads to a novel *systolic cache* architecture comprising of shift registers only (cache without tags). To demonstrate how the methodology works, we have presented several systolic design examples based on the block-matching motion estimation algorithm (BMA). By multiprojecting a 4D DG of the BMA to 2D mesh, we can reconstruct several existing array processors. By multiprojecting a 6D DG of the BMA, a novel 2D systolic array can be derived that features significantly improved rates in data reusability (96%) and processor utilization (99%).

1. Introduction

The rapid progress in VLSI technology will soon reach more than 100 million transistors in a chip, implying tremendous computation power for many applications, e.g., real-time multimedia processing. Many important design issues emerge for the hardware design for these applications:

1. High performance and high flexibility
2. Low cost, low power, and efficient memory usage
3. System-on-a-chip or the ease of system integration
4. Fast design turn-around

The challenge is that many of these design issues discord with each other.

In addressing these critical issues, we present a systematic methodology to support the design of a broad scope of array processors. This allows us to design and evaluate diverse designs easily and quickly. This algebraic methodology can handle algorithms with high-dimensional data dependency. It can exploit a high degree of data reusability and thus it can design high performance processor arrays with high efficiency in memory usage.

In this paper, we focus on the block-matching motion estimation algorithm (BMA) [6] as an example. The basic idea of the BMA is to locate a displaced block, which is most similar to the current block, within the search area in the previous frame as shown in Fig. 1. Various criteria have been presented for the BMA. The most popular one is to find the least sum of the absolute difference (SAD) as

$$\text{Motion Vector} = \arg\min_{[u,v]}\{SAD[u,v]\}$$

$$SAD[u,v] = \sum_{i=1}^{n}\sum_{j=1}^{n} \mid s[i+u, j+v] - r[i,j] \mid$$

$$-p \le u \le p,\ -p \le v \le p$$

where n is the block width and height, p is the absolute value of the maximum possible vertical/horizontal motion, $r[i,j]$ is the pixel intensity (luminance value)

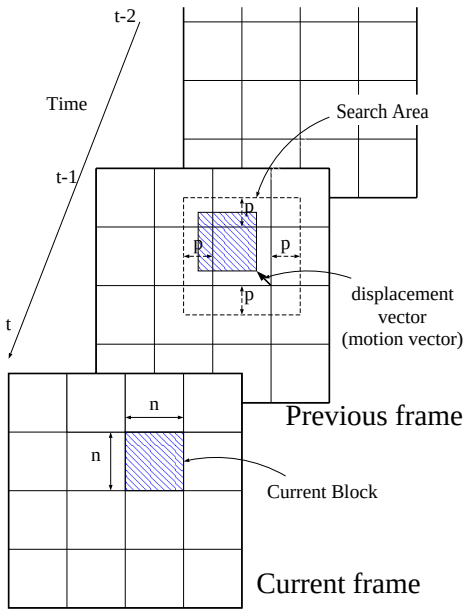

Fig. 1. In the process of the block-matching motion estimation algorithm, the current frame is divided into a number of non-overlapping current blocks, which are n pixels $\times$ n pixels. Each of the current blocks will be compared with $(2p+1) \times (2p+1)$ different displaced blocks in the search area of the previous frame.

in the current block at (i, j), $s[i+u, j+v]$ is the pixel intensity in the search area in the previous frame, and (u, v) represents the candidate displacement vector.

The BMA is extremely computationally intensive in current video coding [7, 15]. For example, a SAD for a block of 16×16 pixels requires 512 additions. For search range $\{-32, \cdots, +32\} \times \{-32, \cdots, +32\}$, there are 4225 SADs, and hence, 2.16×10^6 additions. For a video with 720 pixels $\times$ 480 pixels $\times$ 30 frames per second, 88×10^9 additions per second would be required for a real-time MPEG-1 video coding. In order to tackle such a computationally demanding problem in real-time, putting massively parallel processing elements (PEs) together as a computing engine, like systolic array, is often mandatory.

Such fully utilized processing power can process a tremendous amount of data. In the example, each pixel in the previous frame will be revisited thousands of times. If each visit involves a memory fetch, it would imply an extremely short memory read cycle time (32 ps) for real-time motion estimation of CCIR 601 pictures. So far, state-of-the-art memories are far beyond such demand. In order to make the data flow keep up with the processing power, memory access localities must be exploited. Particularly, data reusability plays a critical role in the systolic design of many important applications.

In order to find a good tradeoff point between several conflicting design goals, a systematic/comprehensive design methodology must be used. Since most multimedia signal processing algorithms have the following features: localized operations, intensive computation, and matrix operation, high-level mapping methodologies are proving very efficient. (For the reader's convenience, in the Appendices, we review the basic systolic design notations and methodology.)

1.1. Previous Approaches for Systolic BMA Design

Because the BMA for a single current block is a 4-dimensional algorithm (as shown in Appendix A.1), it is impossible to get a 2D or 1D system implementation by one projection. Conventionally, the BMA is decomposed into subparts, which (1) are individually defined over index spaces with dimensions less than or equal to three and (2) are suitable to perform the canonical projection. The *functional decomposition* method simplifies the multi-dimensional time schedule and projection problem [5, 10, 16, 20]. For example, one such decomposition is to take u out first and consider it later as follows:

$$SAD[v] = \sum_{i=1}^{n} \sum_{j=1}^{n} \mid s[i, j+v] - r[i, j] \mid$$

$$-p \leq v \leq p$$

As a result, we can get several existing DGs as shown in Fig. 2.

There are many arrays in [10, 16] that can be derived by canonical projecting of the 3D DG shown in Fig. 2. However, most of the designs require a huge amount of memory bandwidth. For example, the design shown in Fig. 3(a) can be derived by projecting the DG in Fig. 2 along the v-direction. This design needs 16 byte data per cycles. Without sufficient memory bandwidth, the PEs are idle most of the time. Hence, most of these designs are not practical.

Another method (called *index fixing*) fixes one the loop index at a time over and over. When two or fewer loop indices remain, the remaining algorithm can be easily transformed into systolic design [4, 5, 10, 16]. For example, the design in Fig. 3(a) can also be derived by fixing the index of the u and v of the 4-dimensional DG.

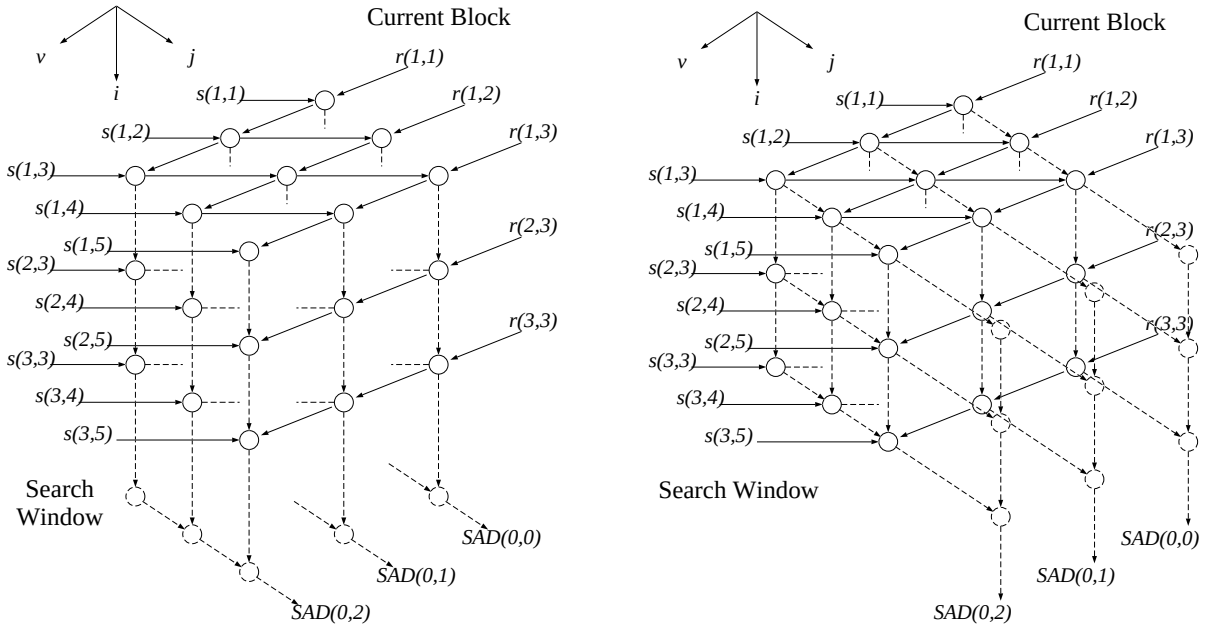

Fig. 2. Two 3D DG examples of the BMA [2, 10, 16].

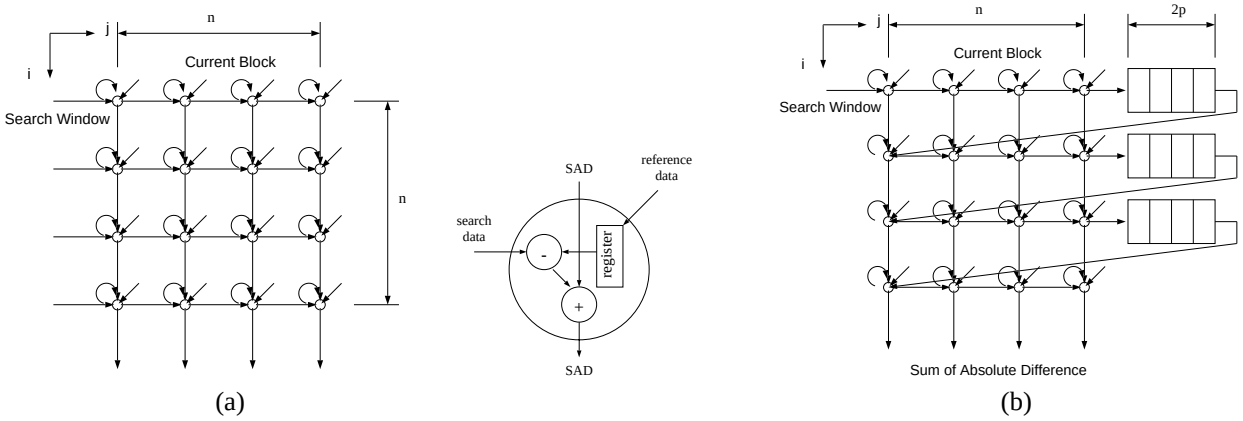

Fig. 3. Previous array design examples. (a) Projected without buffers. (b) Projected with buffers [8].

A breakthrough design that greatly reduces the I/O bandwidth by exploiting *data reusability* is shown in [8] (cf. Fig. 3(b)). It carries some extra buffers. The advantage of this design is that the data are input serially such that the hunger of the I/O is greatly reduced. The amount of the input data per operation is only 1 byte. Furthermore, shift registers instead of random access memories are used here such that the control is easier, the buffer area is smaller, and the data access rate is higher. Moreover, because the search windows of the current blocks overlap each other, a simple FIFO (based on this design) is proposed to capture more data reusability and thus further reduce the I/O bandwidth [14].

However, the design shown in Fig. 3(b) is one of the designs that is blamed for inefficiency because of unnecessary computations. The inefficiency comes from the following problem: In order to have only one I/O port for the whole array, the data running through the whole array must be unified. Hence, in this design, some processor may receive some useless data and do some unnecessary computation (or without doing real computation) [1, 8]. The utilization rate $= \dfrac{(2p+1)^2}{(n+2p)^2}$.

Later, a 2D array design prevents some unnecessary data running through every PE by inputting the data from two memory ports [1]. It not only needs low I/O bandwidth but can also achieve high computational power.

A transformation of snapshot (called *slice and tile*) is employed to produce different forms of DGs [2]. There will be a reduction of one dimension in the DG. For example, an original 3D BMA would become a 2D DG. After that, canonical single projection approaches can be used. This technique can re-design most of the existing architectures in graphs. However, the memory organization must be designed via a careful bookkeeping system on the information about the interface between subparts.

1.2. Overview of this Work

In this paper, we present a systematic methodology, multiprojection, to support the design of a broad scope of array processors. Many previous approaches, such as *functional decomposition*, *index fixing*, and *slice and tile*, can be regarded as its special cases.

We also propose several useful rules essential for the implementation of multiprojection. For instance, by applying LPGS (locally parallel globally sequential) or LSGP (locally sequential globally parallel) during the multiprojection, the design can enjoy expandabilities without compromising the data reusability. Other rules for reducing the number of buffers are also made available. The rules may be adopted to improve computational power and flexibilities and reduce I/O requirement and control overhead.

We shall demonstrate how the multiprojection can achieve this goal, based on a systolic design example of the BMA. Our methodology is applied to design (1) massively parallel systolic architectures and (2) fast *systolic cache* architectures for the MPEG application.

2. Multiprojection Methodology for Optimal Systolic Design

Conventional single projection can only map an n-dimensional DG directly onto an $(n-1)$-dimensional SFG. However, due to current VLSI technology constraint, it is hard to implement a 3D or 4D systolic array. In order to map an n-dimensional DG directly onto an $(n-k)$-dimensional SFG without DG decomposition, a multi-dimensional projection method is introduced [11, 17, 18, 24].

The projection method, which maps an n-dimensional DG to an $(n-1)$-dimensional SFG, can be applied k times and thus reduces the dimension of the array to $n-k$. More elaborately, a similar projection method can be used to map an $(n-1)$-dimensional SFG into an $(n-2)$-dimensional SFG, and so on. This scheme is called *multiprojection*.

The *functional decomposition*, *index fixing*, and *slice and tile* are the special cases of the multiprojection. Multiprojection can not only obtain the DGs and SFGs from functional decomposition but can also obtain other 3D DGs, 2D SFGs, and other designs that are difficult to be obtained from other methods.

Multiprojection is introduced here to design array processors which satisfy most of the following design criteria: (1) increase the computational power, (2) reduce the I/O requirement, (3) reduce the control overhead, and (4) have some expandabilities. For example, a localized recursive algorithm for block matching is derived so that the original 6D BMA is transferred into 3D algorithm [22]. (We will see why the BMA is 6-dimensional later in Section 2.1 and Section 4.3.) After that, it is derived into two designs—a 1D systolic array and a 2D semi-systolic array. Both of the arrays are reported to achieve an almost 100% utilization rate. Nevertheless, since the original 6D is folded into 3D, the designs have more constraints. The former one requires a massive amount of I/O ports. The latter one is only useful when the size of the current block (n) is equal to twice of the search range ($2p$) and requires a massive amount of data broadcasting.

2.1. High Dimensional Algorithm

Before we jump into the discussion of the multiprojection, it is advisable to introduce the concept of high-dimensional algorithms first. An algorithm is said to be n-dimensional if it has n-depth recursive loops in nature. For example, a block-matching algorithm for the whole frame is 6-dimensional as shown Fig. 4(a). The indices x, y, u, v, i, j contribute the algorithm into 6D.

It is very important to respect the *read-after-read* data dependency. If a datum could be read time after time by hundreds of operations and those operations are put closely together, then a small cache can get rid of a large amount of external memory accesses.

```
for (x = 0; x < Nh; x++)
    for (y = 0; y < Nv; x++)
    {
        for (u = -p; u <= p; u++)
            for (v = -p; v <= p; v++)
            {
                SAD = 0;
                for (i = 1; i <= n; i++)
                    for (j = 1; j <= n; j++)
                        SAD = SAD + |s[x*n+i+u, y*n+j+v] - r[x*n+i, y*n+j]|;

                if (Dmin > SAD)
                {
                    Dmin = SAD;
                    MV[x, y] = [u, v];
                }
            }
    }
```

(a)

```
for (a = 0; a < Nh * Nv; a++)
{
    x = a div Nv;
    y = a mod Nv;
    for (b = 0; b < (2*p+1) * (2*p+1); b++)
    {
        u = b div (2*p+1) - p;
        v = b mod (2*p+1) - p;
        for (c = 0; c < n * n; c++)
        {
            i = c div n + 1;
            j = c mod n + 1;
            .
            .
        }
    }
}
```

(b)

```
for (j1 = 0; j1 < 2; j1++)
    for (j2 = 1; j2 <= n / 2; j2++)
        SAD = SAD + | s[x*n+i+u, y*n+j1*n/2+j2+v] - r[x*n+i, y*n+j1*n/2+j2] |;
```

(c)

Fig. 4. (a) The 6D BMA, where N_v is the number of current blocks in the vertical direction, N_h is the number of current blocks in the horizontal direction, n is the block size, and p is the search range. The indices x, y, u, v, i, j contribute the algorithm into 6D. The inner four loops are exactly those is shown in Fig. 22. (b) A 3D BMA that folds two loops in (a) into one loop. (c) On the other hand, a 7D BMA (x, y, u, v, i, j_1, j_2 7-dimension) can be constructed by modifying the inmost loop index `j` of the original algorithm into two indices `j` and $\mathtt{j}_2$.

Since `s[x*n+i+u, y*n+j+v]` will be read time after time for different x, y, u, v, i, j combinations, this algorithm is 6D.

One the other hand, if we ignore the read-after-read data dependency, the DG has only two-dimensional *read-after-write* dependency based on variable `SAD`. Although the DG become lower dimensional, it would be harder to track the data reusability and reduce the amount of memory accesses.

Transformation to Lower Dimension. As shown in Fig. 4(b), two loops are folded into one loop to make the algorithm become less-dimensional [22].

The DG becomes 3-dimensional because there are only 3 loop indices. The number of projections in multiprojection become less and it is easier to optimize the scheduling. However, in this modified algorithm, the operation regarding `(u,v+1)` must be executed directly after the operation regarding `(u,v)`. It makes the algorithm become less flexible. Efficient, expandable, and low I/O designs are harder to achieve. Besides, the folding of 6D DG will make it benefit less from some useful graph transformation as shown in Section 3.

Transformation to Higher Dimension. We can also construct some artificial indices to make a lower-dimensional DG problem become higher-dimensional DG. For example, the inmost loop of the original algorithm could be modified as shown in Fig. 4(c).

The indices x, y, u, v, i, j_1, j_2 transform this algorithm into a 7-dimensional concept. This approach is not generally recommended because the number of steps for multiprojection increases in order to have the low-dimension design. However, this method provides an option of execution in the order of $j = \{1, N/2+1, 2, N/2+2, \ldots\}$ instead of $j = \{1, 2, \ldots, N/2, N/2+1, \ldots\}$ (simply exchanging the order of the $\mathtt{j}_1$ loop and the $\mathtt{j}_2$ loop). As we will see later in Section 3.7, LSGP and LPGS partitioning can be carried out via multiprojection after a DG is transformed into an artificial higher-dimensional DG.

2.2. Algebraic Formulation of Multiprojection

The process of multiprojection could be written as a number of single projections using the same algebraic formulation as introduced in Appendix A.1. In this section, we explain how to project the $(n-1)$-dimensional SFG to an $(n-2)$-dimensional SFG. The potential difficulties of this mapping are (1) the presence of delay edges in the $(n-1)$-dimensional SFG, and (2) the delay management of the edges in the $(n-2)$-dimensional SFG.

Double-Projection. For simplicity, we first introduce how to have a 2D SFG for a 4D DG by the multiprojection.

Step 1 We project the 4D DG into a 3D SFG by projection vector $\vec{d}_4$ (4×1 column vector), projection matrix $\mathbf{P}_4$ (3×4 matrix), and scheduling vector $\vec{s}_4$ (4×1 column vector) with three constraints: (1) $\vec{s}_4^T \vec{d}_4 > 0$, (2) $\mathbf{P}_4 \vec{d}_4 = 0$, and (3) $\vec{s}_4^T \vec{e}_i \geq 0 \quad \forall i$. The computation node $\underline{c}$ (4×1) in 4D DG will be mapped into the 3D SFG by

$$\begin{bmatrix} T_3(\underline{c}) \\ \underline{n}_3(\underline{c}) \end{bmatrix} = \begin{bmatrix} \vec{s}_4^T \\ \mathbf{P}_4 \end{bmatrix} \underline{c}$$

The data dependence edges will be mapped into the 3D SFG by

$$\begin{bmatrix} D_3(\vec{e}_i) \\ \vec{m}_3(\vec{e}_i) \end{bmatrix} = \begin{bmatrix} \vec{s}_4^T \\ \mathbf{P}_4 \end{bmatrix} \vec{e}_i$$

Theorem 1. $D_3(\vec{e}_i) \neq 0$ *for any* $\vec{m}_3(\vec{e}_i) = 0$.

Proof: For $\vec{m}_3(\vec{e}_i) = 0$, $\vec{e}_i$ is proportional to $\vec{d}_4$. For example, $\vec{e}_i = \alpha \vec{d}_4$ ($\alpha \neq 0$). The basic constraint $\vec{s}_4^T \vec{d}_4 > 0$ implies $\alpha \vec{s}_4^T \vec{d}_4 \neq 0$; therefore, $D_3(\vec{e}_i) = \vec{s}_4^T \vec{e}_i \neq 0$. □

Step 2 We project the 3D SFG into a 2D SFG by projection vector $\vec{d}_3$ (3×1 column vector), projection matrix $\mathbf{P}_3$ (2×3 matrix), and scheduling vector $\vec{s}_3$ (3×1 column vector) with three constraints: (1) $\vec{s}_3^T \vec{d}_3 > 0$, (2) $\mathbf{P}_3 \vec{d}_3 = 0$, and (3) $\vec{s}_3^T \vec{m}_3(\vec{e}_i) \geq 0$ $\forall \vec{e}_i$ for broadcasting data. Or, $\vec{s}_3^T \vec{m}_3(\vec{e}_i) > 0 \; \forall \vec{e}_i$ for non-broadcasting data.
The computation node $\underline{n}_3(\underline{c})$ (3×1) in the 3D SFG, which is mapped from $\underline{c}$ (4×1) in the 4D DG, will be mapped into the 2D SFG by

$$\begin{bmatrix} T_2'(\underline{c}) \\ \underline{n}_2(\underline{c}) \end{bmatrix} = \begin{bmatrix} \vec{s}_3^T \\ \mathbf{P}_3 \end{bmatrix} \underline{n}_3(\underline{c})$$

The data dependence edges in the 3D SFG will further be mapped into the 2D SFG by

$$\begin{bmatrix} D_2'(\vec{e}_i) \\ \vec{m}_2(\vec{e}_i) \end{bmatrix} = \begin{bmatrix} \vec{s}_3^T \\ \mathbf{P}_3 \end{bmatrix} \vec{m}_3(\vec{e}_i)$$

Step 3 We can combine the results from the previous 2 steps. Let allocation matrix $\mathbf{A} = \mathbf{P}_3 \mathbf{P}_4$ and scheduling vector $\mathbf{S}^T = \vec{s}_3^T \mathbf{P}_4 + M_4 \vec{s}_4^T$. ($M_4 \geq 1 + (N_4 - 1)\vec{s}_3^T \vec{d}_3$ where N_4 is the maximum number of nodes along the $\vec{d}_3$ direction in the 3D SFG.)

- Node mapping:

$$\begin{bmatrix} T_2(\underline{c}) \\ \underline{n}_2(\underline{c}) \end{bmatrix} = \begin{bmatrix} \mathbf{S}^T \\ \mathbf{A} \end{bmatrix} \underline{c}$$

where $\underline{n}_2(\underline{c}) = \mathbf{A}\underline{c}$ means where the original computational node $\underline{c}$ is mapped. $T_2(\underline{c}) = \mathbf{S}\underline{c}$ means when the computation node is to be executed.

- Edge mapping:

$$\begin{bmatrix} D_2(\vec{e}_i) \\ \vec{m}_2(\vec{e}_i) \end{bmatrix} = \begin{bmatrix} \mathbf{S}^T \\ \mathbf{A} \end{bmatrix} \vec{e}_i$$

where $\vec{m}_2(\vec{e}_i) = \mathbf{A}\vec{e}_i$ means where the original data dependency relationship is mapped. $D_2(\vec{e}_i) = \mathbf{S}^T\vec{e}_i$ means how much time delay should be in the edge $\vec{m}_2(\vec{e}_i)$.

Constraints for Data and Processor Availability. Every dependent datum comes from previous computation. To ensure data availability, every edge must have at least one unit of delay if the edge is not broadcasting some data.

Theorem 2. **Data Availability.** *$D_2(\vec{e}_i) = \mathbf{S}^T\vec{e}_i \geq 0$ if $\vec{e}_i$ is for broadcasting data. $D_2(\vec{e}_i) = \mathbf{S}^T\vec{e}_i > 0$ if $\vec{e}_i$ is not for broadcasting data.*

Proof:

$$\begin{aligned} D_2(\vec{e}_i) &= \mathbf{S}^T\vec{e}_i \\ &= (\vec{s}_3^T\mathbf{P}_4 + M_4\vec{s}_4^T)\vec{e}_i \\ &= \vec{s}_3^T\mathbf{P}_4\vec{e}_i + M_4\vec{s}_4^T\vec{e}_i \\ &\geq \vec{s}_3^T\mathbf{P}_4\vec{e}_i \\ &\quad \text{(from the constraint (3) in step 1)} \\ &> 0 \quad \text{(or, } \geq 0\text{)} \\ &\quad \text{(from the constraint (3) in step 2)} \end{aligned}$$

□

Two computational nodes that are mapped into a single processor could not be executed at the same time. To ensure processor availability, $T_2(\underline{c}_i) \neq T_2(\underline{c}_j)$ must be satisfied for any $\underline{c}_i \neq \underline{c}_j$ and $\underline{n}_2(\underline{c}_i) = \underline{n}_2(\underline{c}_j)$.

Theorem 3. **Processor Availability.** *$T_2(\underline{c}_i) \neq T_2(\underline{c}_j)$ for any $\underline{c}_i \neq \underline{c}_j$ and $\underline{n}_2(\underline{c}_i) = \underline{n}_2(\underline{c}_j)$.*

Proof: For any $\underline{n}_2(\underline{c}_i) = \underline{n}_2(\underline{c}_j)$

$\Rightarrow \mathbf{P}_3\underline{n}_3(\underline{c}_i) - \mathbf{P}_3\underline{n}_3(\underline{c}_j) = 0$

$\Rightarrow \underline{n}_3(\underline{c}_i) - \underline{n}_3(\underline{c}_j)$ is proportional to $\vec{d}_3$.

$\Rightarrow \underline{n}_3(\underline{c}_i) - \underline{n}_3(\underline{c}_j) = \mathbf{P}_4(\underline{c}_i - \underline{c}_j) = \alpha\vec{d}_3$

Since N_4 is the maximum number of nodes along the $\vec{d}_3$ direction in the 3D SFG, $\alpha \in \{0, \pm1, \pm2, \ldots, \pm(N_4 - 1)\}$.

$$\begin{aligned} &T_2(\underline{c}_i) - T_2(\underline{c}_j) \\ &\quad = \mathbf{S}^T(\underline{c}_i - \underline{c}_j) \\ &\quad = (\vec{s}_3^T\mathbf{P}_4 + M_4\vec{s}_4^T)(\underline{c}_i - \underline{c}_j) \\ &\quad = \vec{s}_3^T\mathbf{P}_4(\underline{c}_i - \underline{c}_j) + M_4\vec{s}_4^T(\underline{c}_i - \underline{c}_j) \\ &\quad = \alpha\vec{s}_3^T\vec{d}_3 + M_4\vec{s}_4^T(\underline{c}_i - \underline{c}_j) \end{aligned}$$

1. If $\mathbf{P}_4\underline{c}_i = \mathbf{P}_4\underline{c}_j$, then $\alpha = 0$ and

$$\begin{aligned} T_2(\underline{c}_i) - T_2(\underline{c}_j) &= M_4\vec{s}_4^T(\underline{c}_i - \underline{c}_j) \\ &\neq 0 \quad \text{(by Theorem 1)} \end{aligned}$$

2. If $\mathbf{P}_4\underline{c}_i \neq \mathbf{P}_4\underline{c}_j$, then $\alpha \in \{\pm1, \ldots, \pm(N_4 - 1)\}$
   (a) If $\vec{s}_4^T(\underline{c}_i - \underline{c}_j) = 0$, then

$$\begin{aligned} T_2(\underline{c}_i) - T_2(\underline{c}_j) &= \alpha\vec{s}_3^T\vec{d}_3 \\ &\neq 0 \quad \text{(by the basic constraint of step 2)} \end{aligned}$$

   (b) If $\vec{s}_4^T(\underline{c}_i - \underline{c}_j) \neq 0$, then by assuming $\vec{s}_4^T(\underline{c}_i - \underline{c}_j) > 0$ without losing generality, we have

$$\begin{aligned} &T_2(\underline{c}_i) - T_2(\underline{c}_j) \\ &\quad = \alpha\vec{s}_3^T\vec{d}_3 + M_4\vec{s}_4^T(\underline{c}_i - \underline{c}_j) \\ &\quad \geq \alpha\vec{s}_3^T\vec{d}_3 \\ &\qquad +(1 + (N_4 - 1)\vec{s}_3^T\vec{d}_3)\vec{s}_4^T(\underline{c}_i - \underline{c}_j) \\ &\quad = (\alpha + (N_4 - 1)\vec{s}_4^T(\underline{c}_i - \underline{c}_j))\vec{s}_3^T\vec{d}_3 \\ &\qquad +\vec{s}_4^T(\underline{c}_i - \underline{c}_j) \\ &\quad \geq (\alpha + (N_4 - 1))\vec{s}_3^T\vec{d}_3 + \vec{s}_4^T(\underline{c}_i - \underline{c}_j) \\ &\qquad (\because \vec{s}_4^T(\underline{c}_i - \underline{c}_j) \geq 1) \\ &\quad \geq 0 + \vec{s}_4^T(\underline{c}_i - \underline{c}_j) \\ &\qquad (\because \alpha + N_4 - 1 \geq 0) \\ &\quad > 0 \end{aligned}$$

   If $\vec{s}_4^T(\underline{c}_i - \underline{c}_j) < 0$, then let $\underline{c}_i' = \underline{c}_j$ and $\underline{c}_j' = \underline{c}_i$. The condition $T_2(\underline{c}_i') \neq T_2(\underline{c}_j')$ for any $\underline{c}_i' \neq \underline{c}_j'$ and $\underline{n}_2(\underline{c}_i') = \underline{n}_2(\underline{c}_j')$ holds. So, the proof will.

Q.E.D. from 1, 2(a), and 2(b). □

Multiprojection n-Dimensional DG into k-Dimensional SFG.

Step 1 Let the n-dimensional SFG define as the n-dimensional DG. That is, $\underline{n}_n(\underline{c}_x) = \underline{c}_x$ and the $\vec{m_n}(\vec{e_i}) = \vec{e_i}$.

Step 2 We project the l-dimensional SFG into a $(l-1)$-dimensional SFG by projection vector $\vec{d_l}$ ($l \times 1$), projection matrix $\mathbf{P}_l$ ($(l-1) \times l$), and scheduling vector $\vec{s_l}$ ($l \times 1$) with basic constraint $\vec{s_l}^T \vec{d_l} > 0$, $\mathbf{P}_l \vec{d_l} = 0$, and $\vec{s_l}^T \vec{m_l}(\vec{e_i}) \geq$ (or $>$) $0 \forall \vec{e_i}$.

The computation node $\underline{c}_i$ ($l \times 1$) and the data dependence edge $\vec{m_l}(\vec{e_i})$ ($l \times 1$) in l-dimensional SFG will be mapped into the $(l-1)$-dimensional SFG by

$$\underline{n}_{l-1}(\underline{c}_i) = \mathbf{P}_l \underline{n}_l(\underline{c}_i) \quad (1)$$

$$\vec{m}_{l-1}(\vec{e_i}) = \mathbf{P}_l \vec{m}_l(\vec{e_i}) \quad (2)$$

Step 3 After $(n-k)$ projections, the results can be combined. The allocation matrix will be

$$\mathbf{A} = \mathbf{P}_k \mathbf{P}_{k+1} \cdots \mathbf{P}_n \quad (3)$$

The scheduling vector will be

$$\begin{aligned} \mathbf{S}^T \;=\; & \vec{s}_{k+1}^T \mathbf{P}_{k+2} \mathbf{P}_{k+3} \cdots \mathbf{P}_n \\ & + M_{k+2}\, \vec{s}_{k+2}^T\, \mathbf{P}_{k+3} \mathbf{P}_{k+4} \cdots \mathbf{P}_n \\ & + M_{k+2} M_{k+3}\, \vec{s}_{k+3}^T\, \mathbf{P}_{k+4} \mathbf{P}_{k+5} \cdots \mathbf{P}_n \\ & \vdots \\ & + M_{k+2} M_{k+3} \cdots M_n\, \vec{s}_n \end{aligned} \quad (4)$$

where $M_l \geq 1 + (N_l - 1)\vec{s_{l-1}}^T \vec{d}_{l-1}$ and N_l is the maximum number of nodes along the $\vec{d}_{l-1}$ direction in the l-dimensional SFG. Therefore,

- Node mapping will be:

$$\begin{bmatrix} T_k(\underline{c}_i) \\ \underline{n}_k(\underline{c}_i) \end{bmatrix} = \begin{bmatrix} \mathbf{S}^T \\ \mathbf{A} \end{bmatrix} \underline{c}_i \quad (5)$$

- Edge mapping will be:

$$\begin{bmatrix} D_k(\vec{e_i}) \\ \vec{m_k}(\vec{e_i}) \end{bmatrix} = \begin{bmatrix} \mathbf{S}^T \\ \mathbf{A} \end{bmatrix} \vec{e_i} \quad (6)$$

Constraints for Processor and Data Availability. If no transmittance property is assumed, every edge must have at least one delay because every dependent data is come from previous computation. It is easy to show that data availability is satisfied, i.e., $D_k(\vec{e_i}) > 0 \forall i$.

Following the same proof of Theorem 3, one can easily show processor availability is also satisfied., i.e., $T_k(\underline{c}_i) \neq T_k(\underline{c}_j)$ for any $\underline{c}_i \neq \underline{c}_j$ and $\underline{n}_2(\underline{c}_i) = \underline{n}_2(\underline{c}_j)$.

2.3. Optimization in Multiprojection

After projection directions are fixed, the structure of the array is determined. The remaining part of the design is to find a scheduling that can complete the computation in minimal time under processor and data availability constraint. That is,

$$\min_{\mathbf{S}} \left(\max_{\underline{c}_x, \underline{c}_y} \{ \mathbf{S}^T (\underline{c}_x - \underline{c}_y) \} \right)$$

under the following constraints:

1. $\mathbf{S}^T \vec{e_i} > 0 \quad \forall \vec{e_i}$ (Data Availability)
2. $\mathbf{S}^T \underline{c}_i \neq \mathbf{S}^T \vec{c_j} \quad \forall \underline{c}_i \neq \underline{c}_j, \mathbf{A}\underline{c}_i = \mathbf{A}\underline{c}_j$ (Processor Availability)

A method using quadratic programming techniques is proposed to tackle the optimization problem [26]. However, it takes non-polynomial time to find the optimal solution. A polynomial-time heuristic approach, which uses the branch-and-bound technique and tries to solve the problem by linear programming, is also proposed [25].

Here, we propose another heuristic procedure to find a near optimal scheduling in our multiprojection method. In each single projection, from i-dimension to $(i-1)$-dimension, find an $\vec{s_i}$ by

$$\vec{s_i} = \arg\min_{\vec{s}} \left\{ \max_{\underline{n}_i(\underline{c}_x), \underline{n}_i(\underline{c}_y)} \left\{ \vec{s}^T \left[\underline{n}_i(\underline{c}_x) - \underline{n}_i(\underline{c}_y) \right] \right\} \right\} \quad \forall \underline{c}_x, \underline{c}_y \in \text{DG} \quad (7)$$

under the following constraints:

1. $\vec{s_i}^T \vec{d_i} > 0$
2. $\vec{s_i}^T \vec{m_i}(\vec{e_j}) \geq 0 \; \forall j$ if $(i-1)$-dimension is not the final goal.
   $\vec{s_i}^T \vec{m_i}(\vec{e_j}) > 0 \; \forall j$ if $(i-1)$-dimension is the final goal.

This procedure will find a linear scheduling vector in polynomial time, when the given processor allocation function is linear. Although we have no proof of optimization yet, several design examples show our method can provide optimal scheduling when the DG is shift-invariant and the projections directions are along the axes. (Nevertheless, it will still be an NP-hard problem for all possible processor allocation and time allocation functions.)

Table 1. Graph transformation rules for equivalent DGs. Note that the *transmittent data*, which are used repeatedly by many computation nodes in the DG (see Appendix A.2), play a critical role here.

Rules	Apply to	Function	Advantages
Assimilarity	2D transmittent data	Keep only one edge and delete the others in the 2nd dimension	Save links
Summation	2D accumulation data	Keep only one edge and delete the others in the 2nd dimension	Save links
Degeneration	2D transmittent data	Reduce a long buffers to a single register	Save buffers
Reformation	2D transmittent data	Reduce a long delay to a shorter one	Save buffers
Redirection	Order independent data (e.g., transmittent or accumulation data)	Opposite the edge	Save problems on negative edges

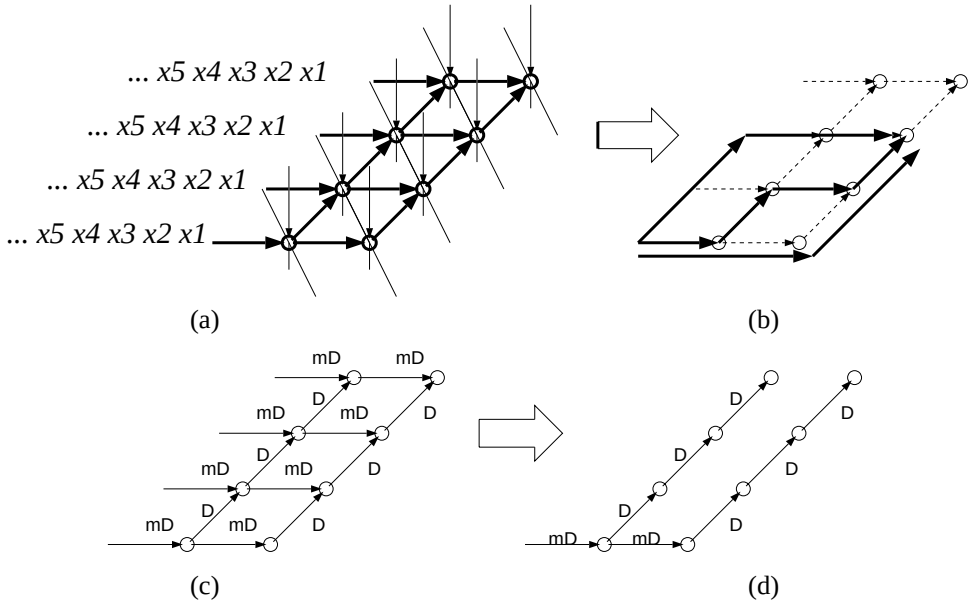

Fig. 5. (a) A high-dimensional DG, where a datum is transmittent to a set of nodes by the solid 2D mesh. (b) There are several paths via which the datum can reach a certain node. (c) During the multiprojection, the dependencies in different directions get different delay. (d) Because the data could reach the nodes by two possible paths, the *assimilarity rule* is applied to this SFG. Only one of the edges in the second dimension is kept. Without changing the correctness of the algorithm, a number of links and buffers are reduced.

3. Equivalent Graph Transformation Rules

In Appendix A.2 and Section 2.1, some transformation rules of the DG are introduced. In order to have better designs, we also provide some graph transformation rules that can help us reduce the number of connections between processors, the size of buffer, or the power consumption. Table 1 shows a brief summary of the rules.

3.1. Assimilarity Rule

As shown in Fig. 5, the assimilarity rule can save some links without changing the correctness of the DG. If a datum is transmittent to a set of operation/computation nodes in the DG/SFG by a 2D (or higher-dimensional) mesh, then there are several possible paths via which the datum can reach a certain node. For example, in the BMA, the `s[i+u,j+v]`

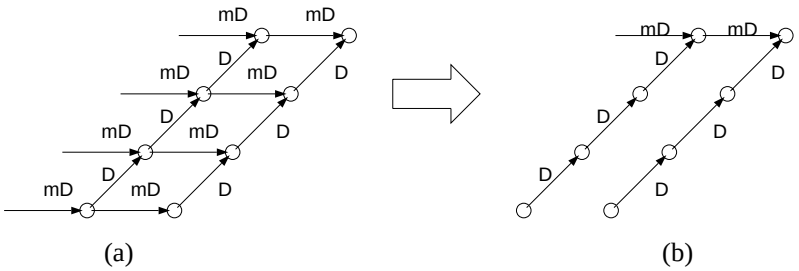

Fig. 6. (a) A datum is the summation of a set of nodes by a 2D mesh in an SFG. During the multiprojection, the dependencies in different directions get different delay. (b) Without changing the correctness of the algorithm, only one of the edges in the second dimension is kept. By the *summation rule*, a number of links and buffers are reduced.

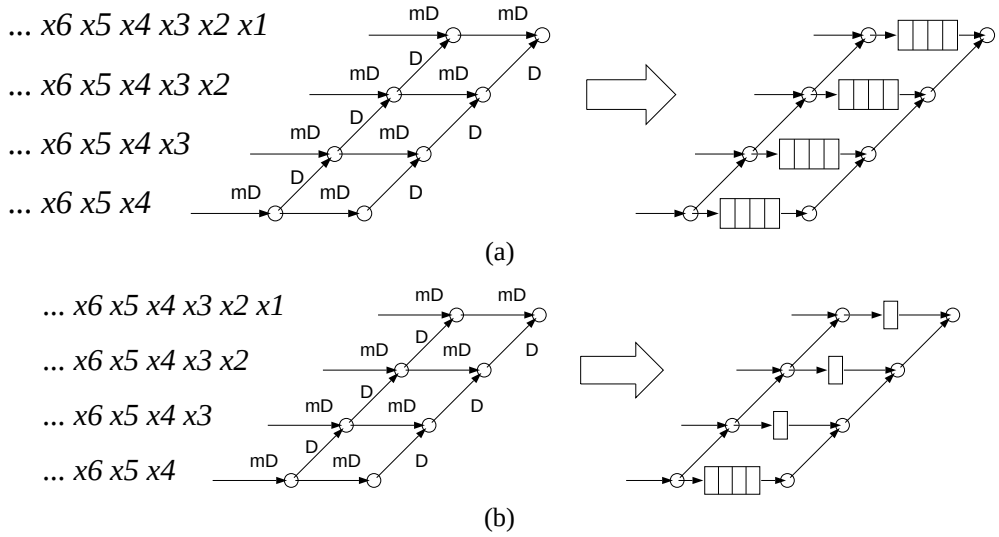

Fig. 7. (a) When transforming an SFG description to a systolic array, the conventional delay management uses $(m-1)$ registers for m units of delay on the links. (b) If the data sets of two adjacent nodes overlap each other, the *degeneration rule* suggests that only a register is required because the other data could be obtained by the other direction.

can be passed by `s[(i+1)+(u-1),j+v]` via loop i, or by `s[i+u,(j+1)+(v-1)]` via loop j. Keeping only one edge in the second dimension is sufficient for the data to reach everywhere.

The procedure of keeping only one edge for a set of edges can save a great number of interconnection buffers. Usually, this rule is applied after the final SFG is obtained. In this way, we can get rid of edges with longer delay and more edges.

One of the major drawbacks of this assimilarity rule is that every node must use the same set of data before this rule can be applied. It is not true for any algorithm that uses a 2D mesh to transmittent the data. Generally speaking, the data set of a node greatly overlaps with the data set of the other nodes but not identically. In order to reduce the connection edges, we can make all the nodes process the same set of data artificially (i.e., ask the nodes to do some useless computations) and then apply this rule.

3.2. Summation Rule

As shown in Fig. 6, the summation rule can save some links without changing the correctness of the DG. Because summation is associative, the order of the summation can be changed. If output is obtained by aggregating a 2D (or higher-dimensional) mesh of computational nodes, we can accumulate the partial sum in one dimension first, then accumulate the total from the partial sum in the second dimension afterward. For example, in the BMA, the SAD[u,v] is the 2D summation of $\mid s[i+u, j+v] - r[i,j] \mid$ over $1 \leq i, j \leq n$. We can accumulate the difference over index i first, or over

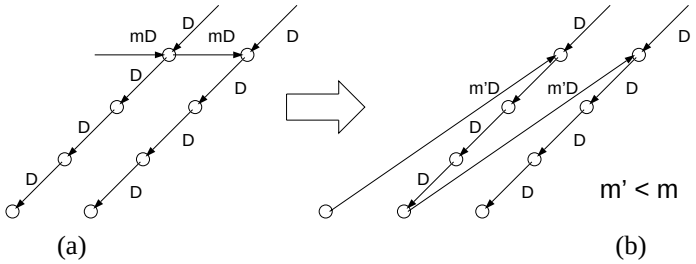

Fig. 8. (a) A high-dimensional DG, where a datum is transmittent to a set of nodes by a 2D mesh, is projected into an SFG. During the multiprojection, the dependencies in different directions get different delay. Because the data could reach the nodes by more than two possible paths, the assimilarity rule is applied to this SFG. Only one of the edges in the second dimension is kept. (b) The delay (i.e., the number of buffers) could be further decreased when the *reformation rule* transforms the original 2D mesh into a tilted mesh.

index j first (cf. Fig 2). We should calculate the data in the direction with fewer buffers first, then rigorously calculate the data in the other direction later.

3.3. Degeneration Rule

The degeneration rule reduces the data link when data are transmittent through a 2D (or higher-dimensional) mesh when (1) each node has its own data set and (2) the data sets of two adjacent nodes overlap each other significatly. One way to save the buffer is to let the overlapping data transmittent from one dimension thoroughly (like that in the assimilarity rule) and let the non-overlapping transmittent from the other dimension(s) (unlike that in the assimilarity rule). In the second dimension, it is only necessary to keep non-overlapping data. Fig. 7 shows that only a register is required because the other data could be obtained by the other direction.

3.4. Reformation Rule

For 2D or higher-dimensional transmittent data, the structure of the mesh is not rigid. For example, in the BMA, the `s[i+u, j+v]` can be passed by `s[(i+k)+(u-k), j+v]` via loop i and by `s[i+u, (j+k)+(v-k)]` via loop j for $1 \leq k \leq n$. For a different k, the structure of the 2D transmittent mesh is different. The final delay in the designed SFG will be different. As a result, we should choose k, depending on the required buffer size. Generally speaking, the shorter the delay, the fewer the buffers.

For example, Fig. 8(a) shows a design after applying the assimilarity rule. Only a long delayed edge was left. Moreover, the data are transmittent to the whole array. So, we detour the long delayed edge, make use of the delay in the first dimension, and get the design show in Fig. 8(b), where the longest delay is now shorter.

3.5. Redirection Rule

Because some operations are associative (e.g., summation data, transmittent data), the arcs in the DG are reversible. The arcs are reversed to help the design. For example, the datum `s[(i+1)+(u-1), j+v]` is passed to `s[i+u, j+v]` via loop i in the BMA. After mapping the DG to a SFG, the delay on the edge is negative. Conventionally, negative delay is not allowed and we must find another scheduling vector $\vec{s}$. This rule tells us to move the data in the opposite direction (passing the `s[i+u, j+v]` to `s[(i+1)+(u-1), j+v]`) instead of re-calculating the scheduling vector (cf. Fig. 9).

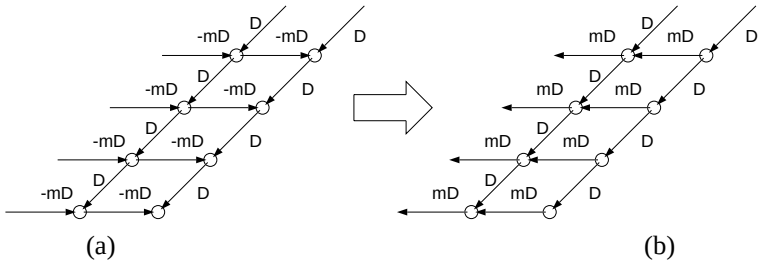

Fig. 9. (a) Generally speaking, an SFG with a negative delay is not permissible. (b) However, if the the dependencies have no polarization, then we apply the *redirection rule* to direct the edges with negative delay to the opposite direction. After that, the SFG become permissible.

3.6. Design Optimization vs. Equivalent Transformation Rules

All these rules do not modify the correctness of the implementation, but could accomplish some degree of design optimization.

1. The assimilarity rule and the summation rule have no influence on the overall calculation time. However, these two rules reduce the buffers and links. Generally speaking, these two rules are applied after the SFG is yielded.
2. The degeneration rule does not influence the overall calculation time. It is applied when one would like to transform the SFG into hardware design. It helps the reduction of the buffers and links. However, extra control logic circuits are required.
3. The reformation rule and the redirection rule will have influence on the scheduling problem because these two rules can make some prohibited scheduling vectors become permissible.

These rules help the design optimization but also make the optimization process harder. Sometimes, the optimization process will become a reiterative procedure which consists of (1) scheduling optimization and (2) equivalent transformation.

3.7. Locally Parallel Globally Sequential and Locally Sequential Globally Parallel Systolic Design by Multiprojection

In Appendix A.4, LPGS and LSGP have been introduced briefly. In this section, we delineate a unified partitioning and scheduling scheme for LPGS and LSGP into our multiprojection method. The advantage of this unified partitioning model is that various partitioning methods can be achieved by choosing projection vectors. The systematic scheduling scheme can explore more inter-processor parallelism.

Equivalent Graph Transformation Rules for Index Folding. A unified re-indexing method is adopted to fold original DG into a higher-dimensional DG but with a smaller size in a chosen dimension. Then, our multiprojection approach is applied to obtain the LPGS or LSGP designs. The only difference between LPGS and LSGP under our uniform approaches is the order of the projection. Our approach is even better in deciding the scheduling because our scheduling is automatically inherited from multiprojection scheduling instead of hierarchical scheduling.

Index Folding. In order to map an algorithm into a systolic array by LPGS or LSGP, we propose a re-

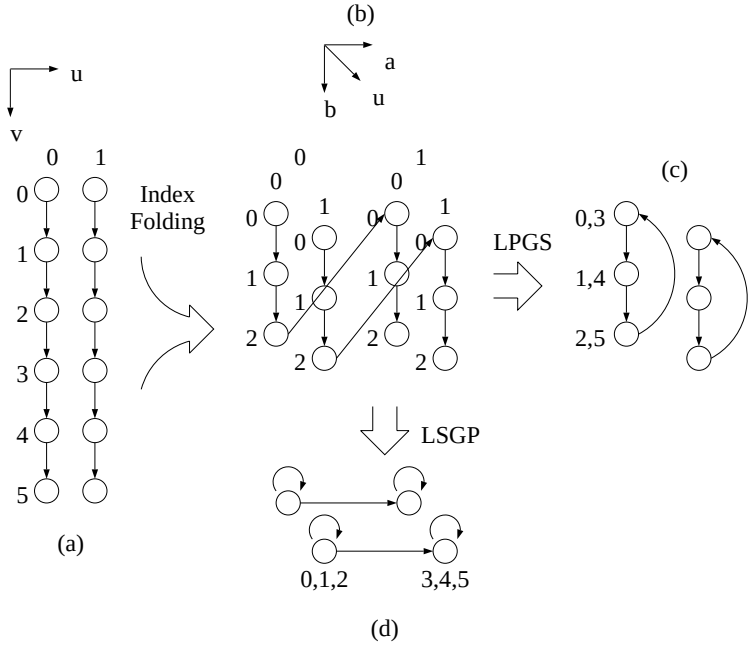

Fig. 10. (a) shows a 2×6 DG. (b) shows an equivalent $2 \times 3 \times 2$ DG after index folding. (c) an LPGS partitioning when we project the 3D DG along the a direction. (d) an LSGP partitioning when we project the 3D DG along the b direction.

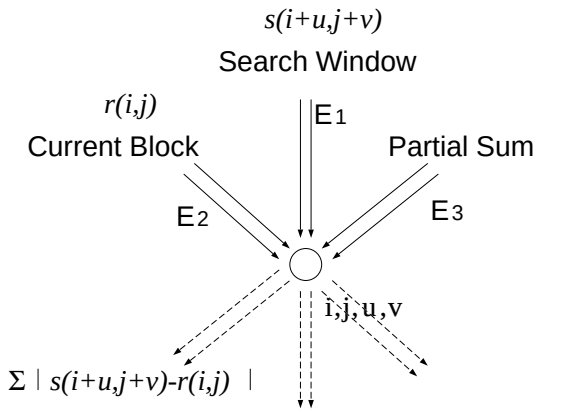

Fig. 11. A core in the 4D DG of the BMA. There are $n \times n \times (2p+1) \times (2p+1)$ nodes in the DG. The node i, j, u, v represents the computation `SAD[u, v] = SAD[u, v] + |s[i + u, j + v] - r[i, j] |`. We denote $\vec{E}_1$ as the data dependency between computation nodes for `s[i + u, j + v]`. Because `s[i + u, j + v]` can come from two possible directions: (1) `s[(i-1) + (u+1), j + v]` or (2) `s[i + u, (j-1) + (v+1)]`, $\vec{E}_1$ can be $(1, 0, -1, 0)$ and $(0, 1, 0, -1)$. By the same token, $\vec{E}_2$—the data dependency of the current block—could be $(0, 0, -1, 0)$ and $(0, 0, -1, 0)$. $\vec{E}_3$, which accumulates the difference, could be $(1, 0, 0, 0)$ and $(0, 1, 0, 0)$. The representation of the DG is not unique; most of the dependence edges can be redirected because of data transmittance.

indexing method for the computational nodes into a higher-dimensional DG problem.

An example is shown in Fig. 10. We want to map a 2×6 DG into a smaller 2D systolic array. Let u, v be the indices ($0 \le u \le 1$, $0 \le v \le 5$) of the DG.

First, we will re-index all the computational nodes (u, v) into (u, a, b). The 2D DG becomes a 3D DG ($2 \times 2 \times 3$) where an a means 3 units of v, a b means 1 unit of v, and $0 \le a \le 1$, $0 \le b \le 2$. Then, a node at (u, a, b) in the 3D DG is equivalent to the node at $(u, (3a + b))$ in the original 2D DG.

After this, by multiprojection, we can have the following two partitioning methods:

1. **LPGS**
   If we project the 3D DG along the a direction, then the nodes that are close to each other in the v direction will be mapped into the different nodes. That is, the computation nodes are going to be executed in parallel. This is an LPGS partitioning.
2. **LSGP**
   If we project the 3D DG along b, then the nodes that are close to each other in the v direction will be mapped into the same node. That is, the computation nodes are going to be executed in a sequential order. This is an LSGP partitioning.

Note that we must be careful about the data dependency after transformation. One unit of original v will be 0 unit of a and 1 unit of b when the dependence edge does not move across different packing segments. (In the example, a packing segment consists of all the computation nodes within three units of sequential v. That is, the packing boundary is when 3 divides v.) One unit of the v is 1 unit of the a and -2 unit of the b when the dependence edge crosses the packing boundary of the transformed DG one time.

4. Systolic Designs for Full-Search Block-Matching Algorithms by Multiprojection Approach

4.1. 4D DG of BMA

As Fig. 22 shows the pseudo code of the BMA of a single current block, Fig. 11 shows a core in the 4D DG of the BMA for a current block. The operations of taking difference, taking absolute value, and accumulating residue are embedded in a 4-dimensional space i, j, u, v. The indeices i and j ($1 \le i, j \le n$) are the indices of the pixels in a current block. The indices u and v ($-p \le u, v \le p$) are the indices of the potential displacement vector. The actual DG would be a 4-dimensional repeat of the same core. Although it is more difficult to visualize the actual DG, it is fairly straightforward to manipulate algebra on the core and thus manipulate multiprojection.

We use $\vec{E}_1$ to denote the data dependency of the search window. The `s[i+u, j+v]` will be used repeatedly for (1) different i, j, (2) same $i + v$, and (3) same $j + u$. Therefore, $\vec{E}_1$ is a 2-dimensional reformable mesh. One possible choice is $(1, 0, -1, 0)$ and $(0, 1, 0, -1)$. The `r[i, j]` will be used repeatedly for different u, v. Hence, $\vec{E}_2$, the data dependency of the current block, could be $(0, 0, -1, 0)$ and $(0, 0, -1, 0)$. The summation can be done in i-first order or j-first order. $\vec{E}_3$, which accumulates the difference, could be $(1, 0, 0, 0)$ and $(0, 1, 0, 0)$. *The representation of the DG is not unique; most of the dependence edges can be redirected because of data transmittance.*

Constructing Previous Designs. As mentioned before, our multiprojection can cover most of the previous design methods. Here is the first example.

The following is the 4D DG of the BMA:

Search Window ($\vec{E}_1$)	1, 0, -1, 0	$D_4 = 0$
	0, 1, 0, -1	$D_4 = 0$
Current Blocks ($\vec{E}_2$)	0, 0, -1, 0	$D_4 = 0$
	0, 0, 0, -1	$D_4 = 0$
Partial Sum of SAD ($\vec{E}_3$)	1, 0, 0, 0	$D_4 = 0$
	0, 1, 0, 0	$D_4 = 0$

After our first projection with $\vec{d}_4^T = (0, 0, -1, 0)$, $\vec{s}_4^T = (0, 0, -1, 0)$, and

$$P_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

the SFG will be

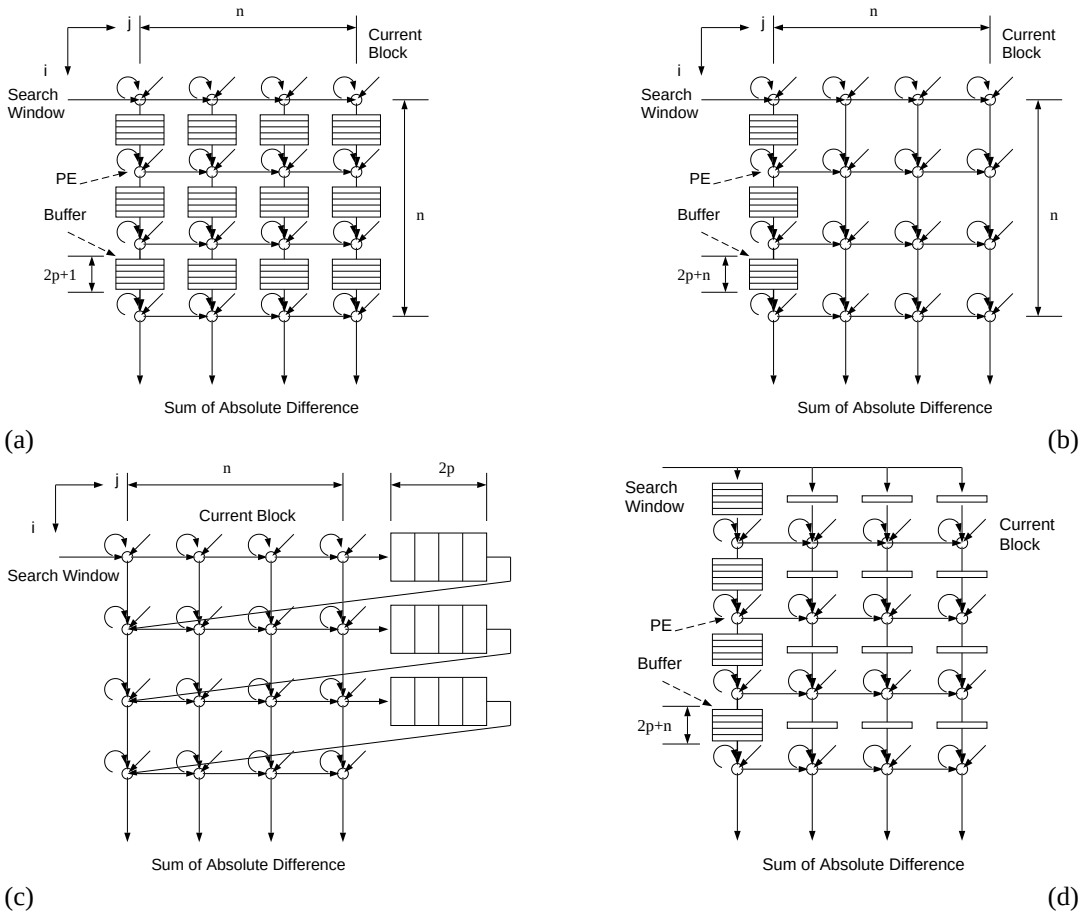

Fig. 12. (a) A 2D BMA systolic design from double-projecting the 4D DG using Eq. (9). (b) The design after the assimilarity rule is applied. (c) The design after the reformation rule is applied (cf. Fig. [8]). (d) The design by applying the degeneration rule. Its timing diagram is shown in Fig. 13.

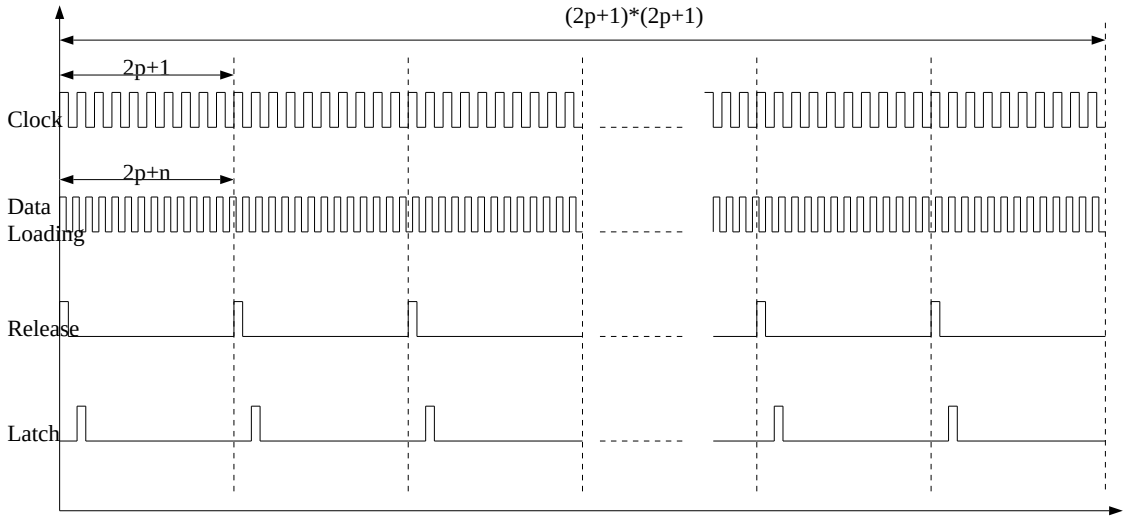

Fig. 13. The timing diagram of the design in Fig. 12(d).

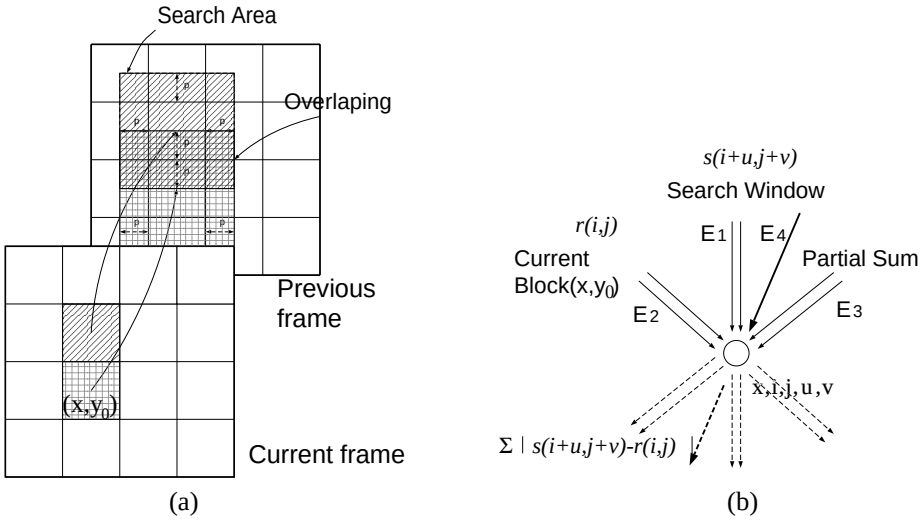

Fig. 14. (a) The data sets of different current blocks indicates the possibilities of the data reuse. (b) The 5D DG of the BMA.

Search Window ($\vec{E}_1$)	1, 0, 0	$D_3 = 1$
	0, 1, 1	$D_3 = 0$
Current Blocks ($\vec{E}_2$)	0, 0, 0	$D_3 = 1$
	0, 0, 1	$D_3 = 0$
Partial Sum of SAD ($\vec{E}_3$)	1, 0, 0	$D_3 = 0$
	0, 1, 0	$D_3 = 0$

If we discard any edges that have delay, then $\vec{E}_1 = (0,1,1)$, $\vec{E}_2 = (0,0,1)$, $\vec{E}_3 = (0,1,0)\&(1,0,0)$. We construct the 3D DG shown in Fig. 2. And, we also construct many previous designs based on the 3D DG.

If we keep the edges that have delays, then we can reconstruct the design in [8] (cf. Fig. 3(b)) by projecting the SFG one more time with $\vec{d}_3^T = (0,0,1)$, $\vec{s}_3^T = (1,0,1)$, and

$$P_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

To ensure processor availability,

$$M \geq 1 + (N-1)(\vec{s}_3 \cdot \vec{d}_3) \tag{8}$$

where N is the maximal number of nodes along the $\vec{d}_3$-direction in the SFG. Because the index u ranges from $-p$ to p, N is $2p+1$. Hence, $M = 2p+1$ and

$$\begin{cases} \mathbf{A} = \mathbf{P_3}\mathbf{P_4} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \\ \mathbf{S}^T = \vec{s}_3^T\mathbf{P_4} + M\vec{s}_4^T = [1, 0, -2p-1, -1]^T \end{cases} \tag{9}$$

We have

Search Window ($\vec{E}_1$)	1, 0	$D_2 = 2p+2$
	0, 1	$D_2 = 1$
Current Blocks ($\vec{E}_2$)	0, 0	$D_2 = 2p+1$
	0, 0	$D_2 = 1$
Partial Sum of SAD ($\vec{E}_3$)	1, 0	$D_2 = 1$
	0, 1	$D_2 = 0$

as Fig. 12(a) shows the design.

Design Via Assimilarity and Reformation Rule. This design has a huge amount of buffers although it can catch considerable data reusability. In order to reduce the number of buffers, we can apply the *assimilarity rule*, as suggested in Section 3.1. We make all the nodes process the same set of data (`s[-p+1, -p+1]`, ..., `s[p+n, p+n]`), and delete most of the link in the second dimension, as shown in Fig. 12(b). We further apply the *reformation rule* to make the delay smaller, and get the design shown in Fig. 12(c), which is identical to the design proposed in [8].

In terms of I/O bandwidth requirements, this design is superior to many other designs because the data are input serially and the I/O bandwidth is reduced by one order of magnitude. Shift registers instead of random access memories are used here. Thus, the control is easier, the buffer area is smaller, and the data access rate is higher. (The I/O rate of the current block is only 6% of the rate f the search window. It is relatively easy to manage the data flow of the current date. Therefore, we focus on the I/O requirement of the search window in this paper.)

However, because of artificial unifying of the input data, some unnecessary data must go through every PE. So, the utilization rate is only 66% when $n = 1$ and $p = 32$.

Design Via Degeneration Rule. Another approach to save buffer for Fig. 12(a) is to apply the *degeneration rule*. As shown in Fig. 12(d), this design can also save a number of buffers as well as keep the processor busy. It has a 77% total utilization rate (include the loading

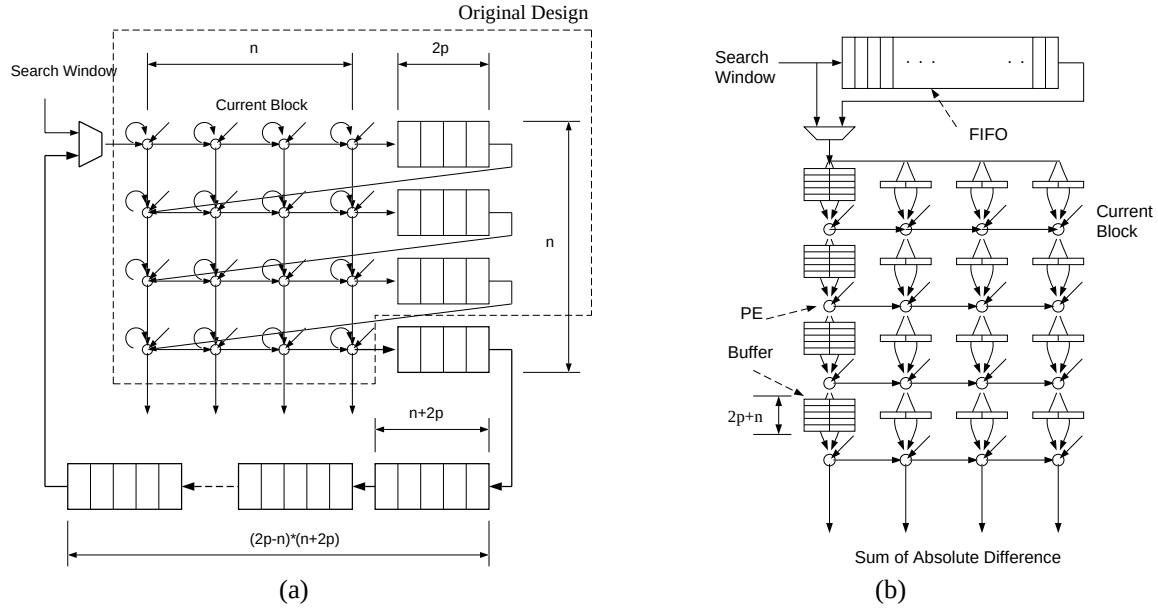

Fig. 15. (a) The design, proposed in [14], can be re-delivered by multiprojecting the 5D DG of the BMA with the assimilarity rule and the reformation rule. (b) A new design can be devised by multiprojecting the 5D DG of the BMA with the degeneration rule.

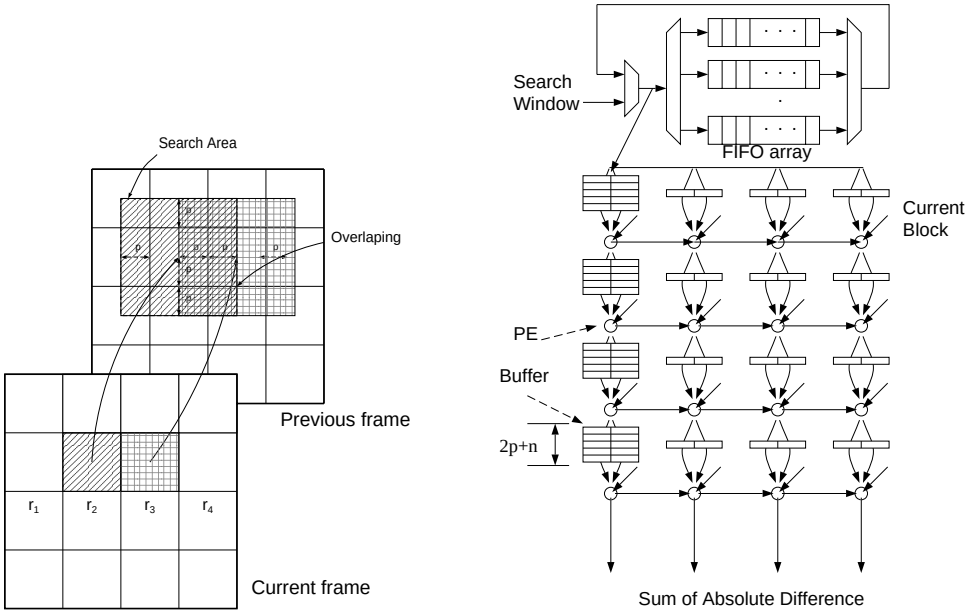

Fig. 16. (a) The data sets of different current blocks (in row-major order) indicates different possibilities of the data reuse. (b) The design with data input in the order of row major. Its timing diagram is shown in Fig. 17.

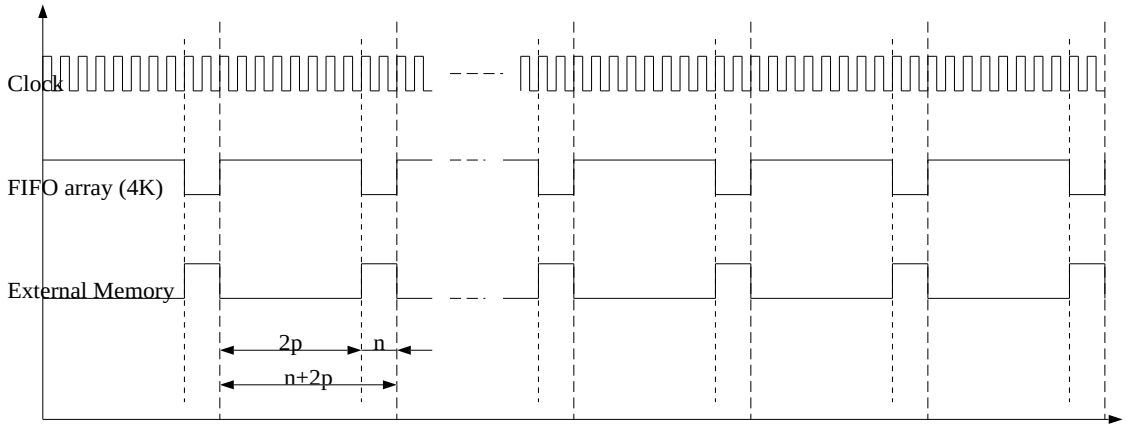

Fig. 17. The timing diagram of the design in Fig. 16(b).

phase and computation), and use only one I/O port for search window. Its timing diagram is shown in Fig. 13.

4.2. Multiprojecting 5D DG of BMA

Increasing the reusability of the data can reduce the I/O and, hence, increase the overall performance. This motivates the introduction of the 5D DG of the BMA. As shown in Fig. 14, two contiguous current blocks may share some parts of the search window.

Let x, y define the indices of the current blocks in a frame. In the 5D design, we fix y at a constant value. $\vec{E}_4$ is new. $\vec{E}_4$ passes the data of the search window shared by the current blocks of a same y. $\vec{E}_1$, $\vec{E}_2$, $\vec{E}_3$ are the same as before; more specifically, $\vec{E}_1$ passes the data of the search window for a given current block.

If we project the 5D DG along x, u, v direction and apply the assimilarity and the reformation rule

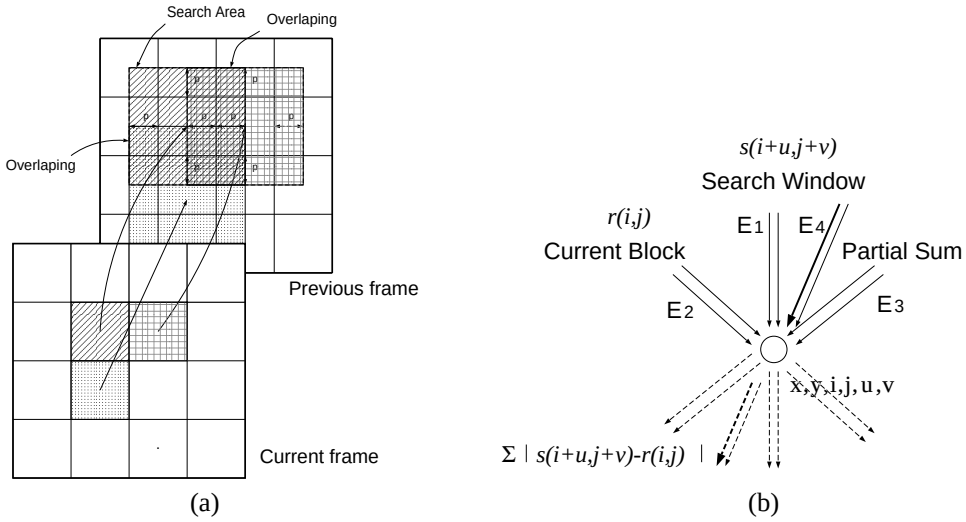

Fig. 18. (a) The data reusability between current blocks. (b) The core of the 6D DG of the BMA. (The core will be repeated when $0 \le x \le N_v$, $0 \le y \le N_h$, $1 \le i, j \le n$, $-p \le u, v \le p$.) $\vec{E}_1 = (0, 0, 1, 0, -1, 0)$ and $(0, 0, 0, 1, 0, -1)$. $\vec{E}_2 = (0, 0, 0, 0, -1, 0)$ and $(0, 0, 0, 0, -1, 0)$. $\vec{E}_3 = (0, 0, 1, 0, 0, 0)$ and $(0, 0, 0, 1, 0, 0)$. $\vec{E}_4 = (1, 0, 0, 0, -n, 0)$ and $(0, 1, 0, 0, 0, -n)$.

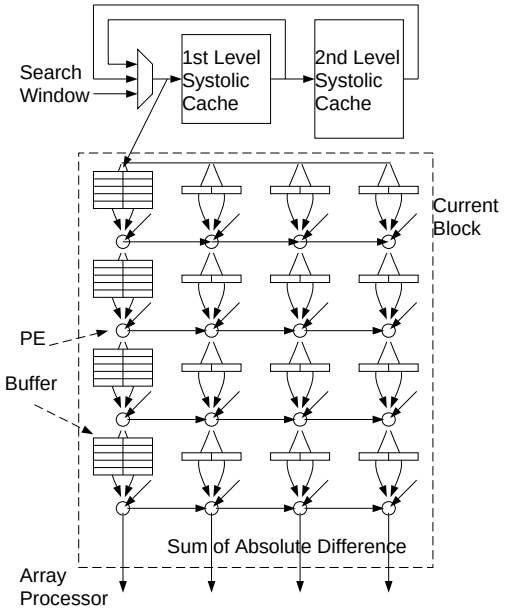

Fig. 19. The design by multiprojecting the 6D DG of the BMA with the degeneration rule. The basic structure of the processor array is the same as 5D design. Its systolic cache is detailed in Fig. 20.

to it, we have the same design as proposed in [14] (cf. Fig. 15(a)). By adding some buffers in the chip, we can reuse a major part of the search window without reloading it. The ratio of reused data: $\frac{2p \times (n+2p)}{(n+2p) \times (n+2p)}$. When $n = 16, p = 32$, the ratio amounts to about 80% while 4KB on-chip buffer is added. However, this design would share the same problem, a low utilization rate, as that in [8] (cf. Fig. 3(b)).

Fig. 15(b) shows the design the after the degeneration rule is applied to 5D DG. It has a 99% total utilization rate (include the loading phase and the computation phase), and uses only one I/O port for search window.

Row-Major 5D DG of BMA. In the previous design, we assume that the BMA will be performed in the column major of the current blocks. However, in MPEG codec, current blocks are coded in the order of the row major. In order to work with current MPEG codec, the previous column-major systolic design may require an extra buffer to save the motion vector information.

In order to avoid the extra buffer, the data that overlapped between the current blocks in the row major (cf. Fig. 16(a)) is also considered. Because the memory designed for the buffer is in row-major, the data reused between two current blocks become piecewise continuous. Its correspondent design and timing diagram are shown in Fig. 16(b) and 17.

4.3. Multiprojecting 6D DG of BMA

As the full-frame BMA is 6D (cf. Fig. 4), Fig. 18 shows the 6D DG of the BMA. Let x, y define the indices of the current blocks in a frame. $\vec{E}_1, \vec{E}_2, \vec{E}_3$ are the same as above. The new feature is that $\vec{E}_4$ now represents inter-block usability shifted in both x and y indices.

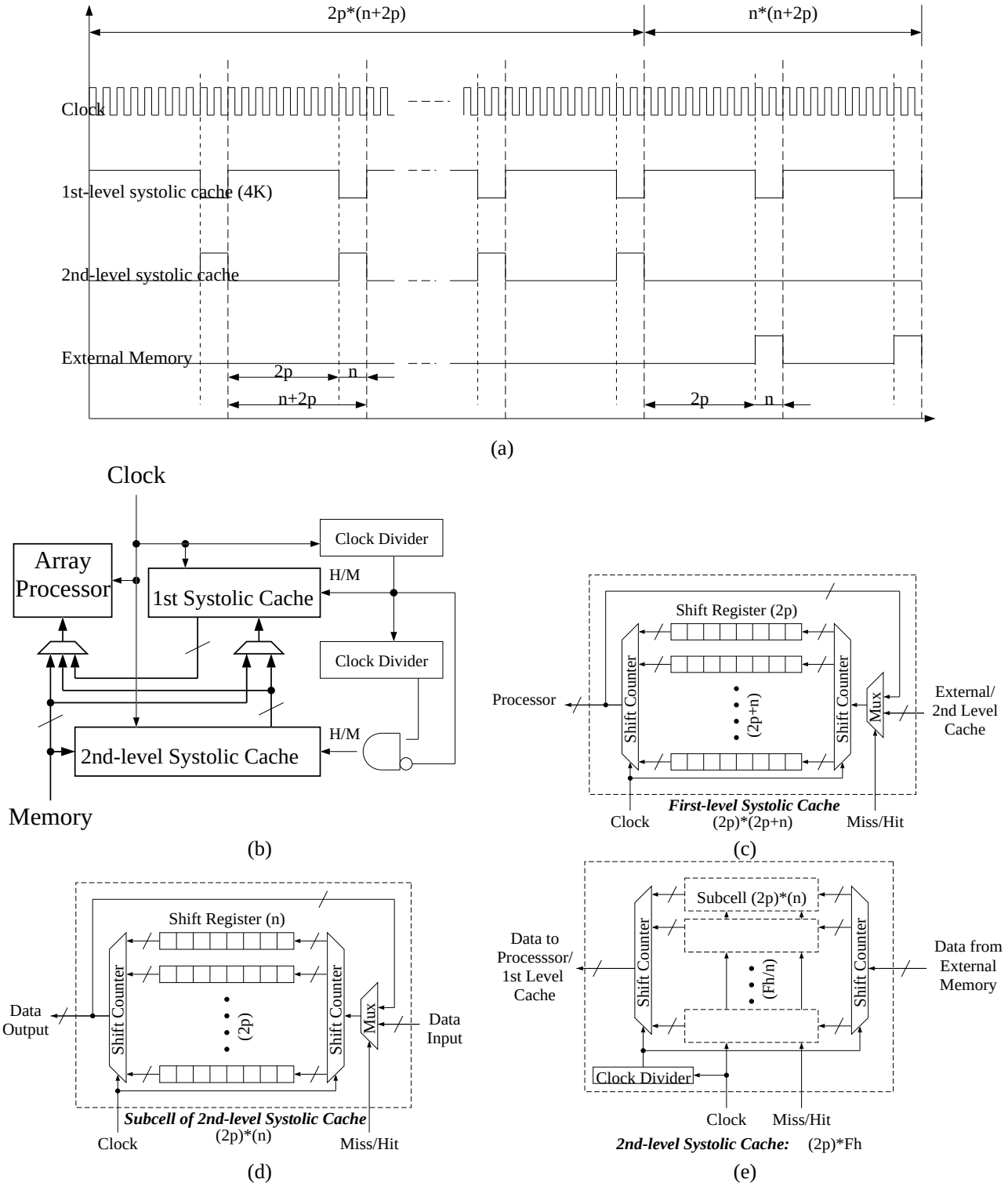

Fig. 20. The *systolic cache* of the design shown in Fig. 19: (a) Its timing diagram. (b) The overall picture. (c) The *first-level* systolic cache. (d) A *subcell* of second-level systolic cache. (e) The *second-level* systolic cache.

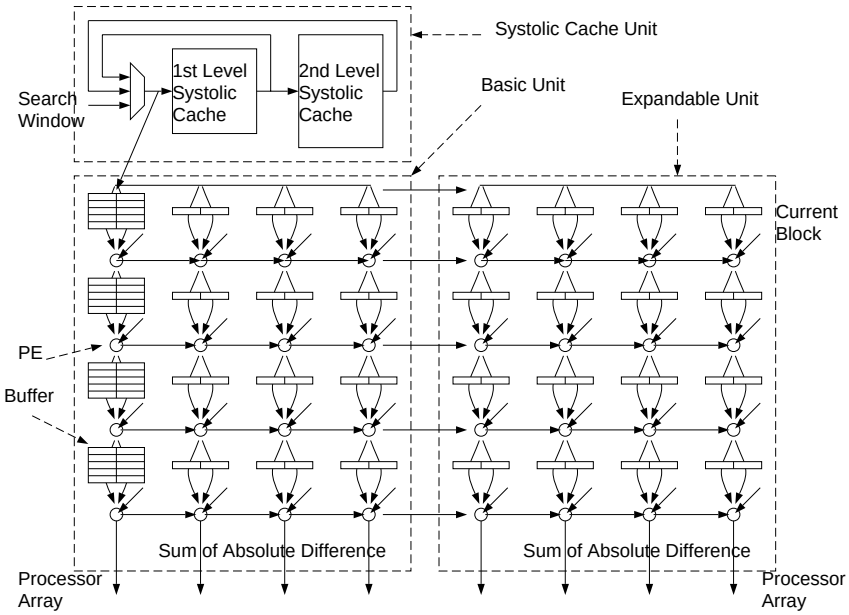

Fig. 21. A seamless design of expandable array processors (cf. Fig 19).

Table 2. A comparison of several designs. Our algebraic design methodology can handle algorithms with high-dimensional data dependency and thus exploit the maximum degree of data reusability. Our design from multiprojection the 6D DG of the BMA can achieve 99% total utilization rate of the PEs and 96% data reusability rate of the search window.

	Advantage	Disadvantage
Our design from 4D DG (By degeneration rule, Fig. 12)	Only one I/O port	81% total utilization rate
Our design from 5D DG (By degeneration rule, Fig. 15)	Only one I/O port 99% total utilization rate	80% data reusability rate
Our design from 6D DG (By degeneration rule, Fig. 21)	Only one I/O port 99% total utilization rate 96% data reusability rate Expandable	

Special Supporting Memory/Cache/Buffer Design. Since it is hard to hold all the data in the same chip that holds the processor array, a small cache is important. Because the memory access pattern is very regular in the full search BMA, there is a predetermined way for best replacement policy of the cache. Eventually, we can get rid of the tags for the cache between the main memory and processing unit because we know (1) where the data should go, (2) which data should be replaced, and (3) where we should fetch the data. Based on this idea, we can design a so-called *systolic cache*—a pre-fetch external cache.

Fig. 19 shows the extended systolic design for the row-major 6D DG. The schematic design of the *systolic cache* to support such a row-major 6D DG design is detailed in Fig. 20. If the width of a frame F_h is 1024 ($F_h = N_h \times n$) and half of the search window size p is 32, then the size of that cache will be $2p \times F_h = 64$K cache.

LPGS and LSGP for Expandable Design. In addition to the overlapping between search windows of different current blocks, another important property is that there

```
for (u = -p; u <= p; u++)
    for (v = -p; v <= p; v++)
    {
        SAD[u, v] = 0;
        for (i = 1; i <= n; i++)
            for (j = 1; j <= n; j++)
                SAD[u, v] = SAD[u, v] + |s[i + u, j + v] - r[i, j] |;
    }

for (u = -p; u <= p; u++)
    for (v = -p; v <= p; v++)
        if (Dmin > SAD[u, v])
        {
            Dmin = SAD[u, v];
            MV = [u, v];
        }
```

(a)

```
for (u = -p; u <= p; u++)
    for (v = -p; v <= p; v++)
    {
      SAD[u, v, 0, n] = 0;
      for (i = 1; i <= n; i++)
      {
          SAD[u, v, i, 0] = SAD[u, v, i - 1, n];
          for (j = 1; j <= n; j++)
              SAD[u, v, i, j] = SAD[u, v, i, j-1] + |s[i+u, j+v] - r[i, j] |;
      }
    }

for (u = -p; u <= p; u++)
    for (v = -p; v <= p; v++)
      if (Dmin > SAD[u, v, n, n])
      {
          Dmin = SAD[u, v, n, n];
          MV = [u, v];
      }
```

(b)

Fig. 22. (a) The pseudo code of the BMA for a single current block. This pseudo code is exactly the inner four loops as shown in Fig. 4(a). (b) A single assignment code for the BMA. Every element in `SAD[u, v, i, j]` array will be assigned to a value only once—as the name come from.

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
    {
        R[u, -p-1, i, j] = r[i, j];
        S[u, -p-1, i, j] = s[u+i, -p-1+j];
    }

for (v = -p; v <= p; v++)
{
    SAD[u, v, 0, n] = 0;
    for (i = 1; i <= n; i++)
    {
        SAD[u, v, i, 0] = SAD[u, v, i - 1, n];
        for (j = 1; j <= n; j++)
        {
            R[u, v, i, j] = R[u, v-1, i, j];
            S[u, v, i, j] = S[u, v-1, i, j+1];
            SAD[u,v,i,j] = SAD[u,v,i,j-1] + | S[u,v,i,j] - R[u,v,i,j] |;
        }
    }
}
```

Fig. 23. A example of the localized recursive BMA. The variable `s[u+i, u+j]` and `r[i, j]` in the inner three loop of the single assignment code shown in Fig. 22(b) are replaced by locally-interconnected array `S[u,v,i,j]` and `R[u,v,i,j]` respectively.

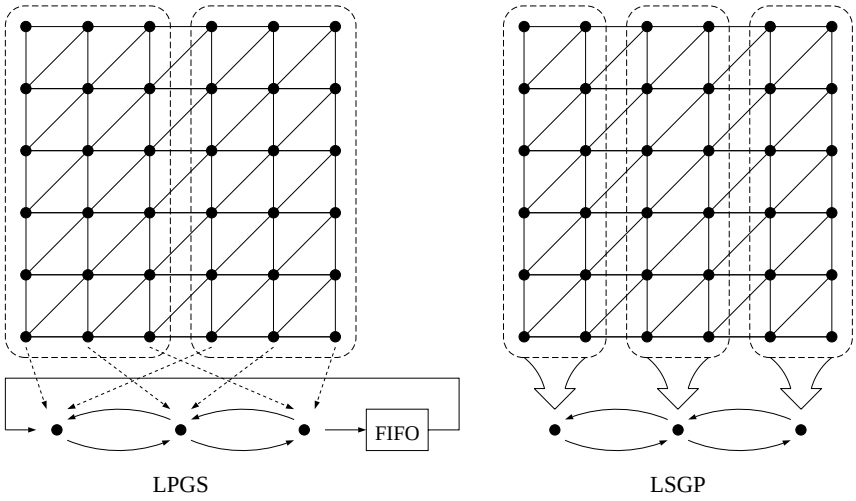

Fig. 24. There are two methods for mapping the partitioned DG to an array: locally parallel globally sequential (LPGS) and locally sequential globally parallel (LSGP).

is **no overlap and no gap** between search windows of different current blocks at any time. The search window data departing one array can be used immediately by another array. The reusable data are taken over naturally by the next array without extra buffers or special links. This design has very high expandabilities. The chips can be cascaded easily without performance lost as shown in Fig. 21.

5. Conclusions

In this work, we concentrate on an algebraic multiprojection methodology, capable of manipulating an algorithm with high-dimensional data dependence, to design the special data flow for highly reusable data.

Multiprojecting the 6D DG of the BMA can give us high performance processor array designs with minimum supporting buffers (cf. Table 2). We can achieve very high data reusability rates by simple buffers, e.g.,

shift registers or cache without tags. The data in the search-window are reused as many times as possible in the SAD computations at different search-positions. Therefore, the problem of the input bandwidth for the search-area data can be alleviated.

It is desirable to have a chip flexible for different block-sizes and search-ranges so that it can be used in a variety of application systems. The size of buffers and their scheduling could be derived automatically when array processors are designed via multiprojection.

In addition, the expandability of the array processor design is very important for some practical implementations. The multiprojection can give us the expandability not only for single chip solution but also for the chip array design.

This work has also been extended to operation placement and scheduling in fine-grain parallel architectures [3]. Because this method exploits cache and communication localities, it results in highly efficient parallel codes.

Appendix

A.1. Common Systolic Design Approaches

Several useful transformation techniques have been proposed for mapping the algorithm into parallel and/or pipeline VLSI architecture [11]. There are 3 stages in common systolic design methodology: the first is dependence graph (DG) design, the second is mapping the DG to a signal flow graph (SFG), and the third is design array processor based on the SFG.

More precisely, a DG is a directed graph, $G =< V, E >$, which shows the dependence of the computations that occur in an algorithm. Each operation will be represented as one node, $\underline{c} \in V$, in the graph. The dependence relation will be shown as an arc, $\vec{e} \in E$, between the corresponding operations. A DG can be also considered as the graphical representation of a single assignment algorithm. Our approach to the construction of a DG will be based on the space-time indices in the recursive algorithm: Corresponding to the space-time index space in the recursive algorithm, there is a natural lattice space (with the same indices) for the DG, with one node residing on each grid point. Then the data dependencies in the recursive algorithm may be explicitly expressed by the arcs connecting the interacting nodes in the DG, while its functional description will be embedded in the nodes. A high-dimensional looped algorithm will lead to a high-dimensional DG. For example, the BMA for a single current block is a 4-dimensional recursive algorithm [22].

A complete SFG description includes both functional and structural description parts. The functional description defines the behavior within a node, whereas the structural description specifies the interconnection (edges and delays) between the nodes. The structural part of an SFG can be represented by a finite directed graph, $G =< V, E, D(E) >$ since the SFG expression consists of processing nodes, communicating edges, and delays. In general, a node, $\underline{c} \in V$, represents an arithmetic or logic function performed with zero delay, such as multiplication or addition. The directed edges $\vec{e} \in E$ model the interconnections between the nodes. Each edge $\vec{e}$ of E connects an output port of a node to an input port of some node and is weighted with a delay count $D(\vec{e})$. The delay count is determined by the timing and is equal to the number of time steps needed for the corresponding arcs. Often, input and output ports are refereed to as sources and sinks, respectively.

Since a complete SFG description should include both functional description (defines the behavior within a node) and structural description (specifies the interconnection—edges and delays—between the nodes), we can easily transform an SFG into a systolic array, wavefront array, SIMD, or MIMD. Therefore, most research is on how to transfer a DG to an SFG in the systolic design methodology.

There are two basic considerations for mapping from a DG to an SFG:

1. **Placement:** To which processors should operations be assigned? (A criterion might be to minimize communication/exchange of data between processors.)
2. **Scheduling:** In what ordering should the operations be assigned to a processor? (A criterion might be to minimize total computing time.)

Two steps are involved in mapping a DG to an SFG array. The first step is the processor assignment. Once the processor assignment is fixed, the second step is the scheduling. The allowable processor and schedule assignments can be quite general; however, in order to derive a regular systolic array, linear assignments and scheduling attract more attention.

Processor Assignment. Processor assignment decides which processor is going to execute which node in the DG. A processor could carry out the opera-

tions of a number of nodes. For example, a projection method may be applied, in which nodes of the DG along a straight line are assigned to a common processing element (PE). Since the DG of a locally recursive algorithm is regular, the projection maps the DG onto a lower dimensional lattice of points, known as the processor space. Mathematically, a linear projection is often represented by a projection vector $\vec{d}$. The mapping assigns the node activities in the DG to processors. The index set of nodes of the SFG are represented by the mapping

$$\mathbf{P} : I^n \rightarrow I^{n-1}$$

where I^n is the index set of the nodes of the DG, and I^{n-1} is the Cartesian product of (n-1) integers. The mapping of a computation $\underline{c}_i$ in the DG onto a node $\underline{n}$ in the SFG is found by:

$$\underline{n}(\underline{c}_i) = \mathbf{P}\underline{c}_i$$

where $\underline{n}(\cdot)$ denotes the mapping function from a node in the DG to a node in the SFG. and the processor basis **P**, denoted by an $(n-1) \times n$ matrix, is orthogonal to $\vec{d}$. Mathematically,

$$\vec{d}^T\mathbf{P} = 0$$

This mapping also maps the arcs of the DG to the edges of the SFG. The set of edges $\vec{m}(\vec{e})$ into each node of the SFG is derived from the set of dependence edges $\vec{e}$ at each point in the DG by

$$\vec{m}(\vec{e}_i) = \mathbf{P}\vec{e}_i$$

where $\vec{m}(\cdot)$ denotes the mapping function from an edge in the DG to an edge in the SFG.

In this paper, bold face letters (e.g., **P**) represent matrices. Overhead arrows represent an n-dimensional vector, written as an $n \times 1$ matrix, e.g., $\vec{e}_i$ (a dependency arc in the DG) and $\vec{m}(\vec{e}_i)$ (an SFG dependency edge that comes for the $\vec{e}_i$). An n-tuple (a point in n-dimensional space), written as an $n \times 1$ matrix, is represented by underlined letters, e.g., $\underline{c}_i$ (a computation node in the DG) and $\underline{n}(\underline{c}_i)$ (an SFG computation node that comes from $\underline{c}_i$).

Scheduling. The projection should be accompanied by a scheduling scheme, which specifies the sequence of the operations in all the PEs. A schedule function represents a mapping from the n-dimensional index space of the DG onto a 1D scheduling time space. A linear schedule is based on a set of parallel and uniformly spaced hyper-planes in the DG. These hyper-planes are called equi-temporal hyper-planes—all the nodes on the same hyper-plane must be processed at the same time. Mathematically, the schedule can be represented by a schedule vector (column vector) $\vec{s}$, pointing to the normal direction of the hyper-planes. The scheduling of a computation $\underline{c}$ in the DG on a node $\underline{n}$ in the SFG is found by:

$$T(\underline{c}) = \vec{s}^T\underline{c}$$

where $T(\cdot)$ denotes the timing function of a node in the DG to the execution time of the processor in the SFG.

The delay $D(\vec{e})$ on every edge is derived from the set of dependence edges $\vec{e}$ at each point in the DG by

$$D(\vec{e}_i) = \vec{s}^T\vec{e}_i$$

where $D(\cdot)$ denotes the timing function of an edge in the DG to the delay of the edge in the SFG.

Permissible Linear Schedules. There is a partial ordering among the computations, inherent in the algorithm, as specified by the DG. For example, if there is a directed path from node $\underline{c}_x$ to node $\underline{c}_y$, then the computation represented by node $\underline{c}_y$ must be executed after the computation represented by node $\underline{c}_x$ is completed. The feasibility of a schedule is determined by the partial ordering and the processor assignment scheme.

The necessary and sufficient conditions are stated below:

1. $\vec{s}^T\vec{e} \geq 0$, for any dependence arc $\vec{e}$. $\vec{s}^T\vec{e} \neq 0$, for non-broadcast data.
2. $\vec{s}^T\vec{d} > 0$.

The first condition stands for data availability and states that the precedent computation must be completed before the succeeding computation starts. Namely, if node $\underline{c}_y$ depends on node $\underline{c}_x$, then the time step assigned for $\underline{c}_y$ can not be less than the time step assigned for $\underline{c}_x$. The first condition means that the causality should be enforced in a permissible schedule. But, if a datum is used by many operations in the DG (read-after-read data dependencies), the causality constraint could be a little bit different. As popularly adopted, the same data value is broadcast to all the operation nodes. The data are called *broadcast data*. In this case, there is no delay required. Alternatively, the same data may be propagated step by step via local

arcs without being modified to all the nodes. This kind of data, which is propagated without being modified, is called *transmittent data*. There should be at least one delay for transmittent data.

The second condition stands for processor availability, i.e., 2 computation nodes cannot be executed in the same time if they are mapped into the same processor element. The second condition implies that nodes on an equi-temporal hyper-plane should not be projected to the same PE. In short, the schedule is permissible if and only if (1) all the dependency arcs flow in the same direction across the hyper-planes; and (2) the hyper-planes are not parallel with projection vector $\vec{d}$.

In general, the projection procedure involves the following steps:

1. For any projection direction, a processor space is orthogonal to the projection direction. A processor array may be obtained by projecting the index points to the processor space.
2. Replace the arcs in the DG with zero or nonzero delay edges between their corresponding processors. The delay on each edge is determined by the timing and is equal to the number of time steps needed for the corresponding arcs.
3. Since each node has been projected to a PE and each input (or output) data is connected to some nodes, it is now possible to attach the input and output data to their corresponding processors.

A.2. The Transformation of DG

Besides the direction of the projection and the schedule, the choice of a particular DG for an algorithm can greatly affect the performance of the resulting array. The following are the two most common transformations of the DG seen in the literature:

- **Reindexing:**
  A useful technique for modifying the DG is to apply a coordinate transformation to the index space (called *reindexing*). Examples for reindexing are plane-by-plane shifting or circular shifting in the index space. For instance, when there is no permissible linear schedule or systolic schedule for the original DG, it is often desirable to modify the DG so that such a desired schedule may be obtained. The effect of this method is equivalent to the re-timing method [13].
- **Localized dependence graph:**
  A locally recursive algorithm is an algorithm whose corresponding DG has only local dependencies—all variables are (directly) dependent upon the variables of neighboring nodes only. The length of each dependency arc is independent of the problem size.
  On the other hand, a non-localized recursive algorithm has global interconnections/dependencies. For example, a same datum will be used by many operations, i.e., the same data value will repeatedly appear in a set of index points in the recursive algorithm or DG. As popularly adopted, the operation nodes receive the datum by broadcasting. The data are called *broadcast data* and this set is termed a broadcast contour. Such a non-localized recursive algorithm, when mapped onto an array processor, is likely to result in an array with global interconnections.
  In general, global interconnections are more expensive than localized interconnections. In certain instances, such global arcs can be avoided by using a proper projection direction in the mapping schemes. To guarantee a locally interconnected array, a localized recursive algorithm would be derived (and, equivalently, a localized DG). In many cases, such broadcasting can be avoided and replaced by local communication. For example, in Fig. 23, the variable `s[u+i, u+j]` and `r[i, j]` in the inner three loops of the BMA (cf. Fig. 22(b)) are replaced by local variables `S[u,v,i,j]` and `R[u,v,i,j]` respectively.
  The key point is that instead of broadcasting the (public) data along a global arc, the same data may be propagated step by step via local arcs without being modified to all the nodes. This kind of data, which is propagated without being modified, is called *transmittent data*.

A.3. General Formulation of Optimization Problems

It takes more efforts to find an optimal and permissible linear scheduling than it does to find a permissible linear scheduling. In this section, we show how to derive an optimal design.

Optimization Criteria. Optimization plays an important role in implementing systems. In terms of parallel processing, there are many ways to evaluate of a de-

sign: one is to measure by the completion time (T), another one is to measure by the product of the VLSI chip area and the completion time ($A \times T$) [12]. In general, the optimization problems can be catagorized into:

1. To find a best scheduling that minimizes the execution time, for given constraints on the number of processing units [25].
2. To minimize the cost (area, power, etc.) under certain given timing constraints [19].

In either case, such tasks are proved to be NP-hard. *In this paper, we focus on how to find an optimal schedule given an array structure—the timing is an optimization goal, not a constraint.*

Basic Formula. First, we know that the computation time of a systolic array can be written as

$$T = \max_{\underline{c}_x, \underline{c}_y} \{\vec{s}^T(\underline{c}_x - \underline{c}_y)\} + 1$$

where $\underline{c}_x$ and $\underline{c}_y$ are two computation nodes in the DG.

The optimization problem becomes the following the min-max formulation:

$$\vec{s}_{op} = \arg\left[\min_{\vec{s}}\left[\max_{\underline{c}_x, \underline{c}_y}\{\vec{s}^T(\underline{c}_x - \underline{c}_y)\} + 1\right]\right]$$

under the following two constraints: $\vec{s}^T\vec{d} > 0$ and $\vec{s}^T\vec{e} > 0$, for any dependence arc $\vec{e}$.

The minimal computation time schedule $\vec{s}$ can be found by solving the proper integer liner programming [12, 21, 25] or quadratic programming [26].

A.4. Partitioning Methods

As DSP systems grow too complex to be contained in a single chip, partitioning is used to design a system into multi-chip architectures. In general, the mapping scheme (including both the node assignment and scheduling) will be much more complicated than the regular projection methods discussed in the previous sections because it must optimize chip area while meeting constraints on throughput, input/output timing and latency. The design takes into consideration I/O pins, inter-chip communication, control overheads, and tradeoff between external communication and local memory.

For a systematic mapping from the DG onto a systolic array, the DG is regularly partitioned into many blocks, each consisting of a cluster of nodes in the DG. As shown in Fig. 24, there are two methods for mapping the partitioned DG to an array: the locally sequential globally parallel (LSGP) method and the locally parallel globally sequential (LPGS) method [11].

For convenience of presentation, we adopt the following mathematical notations. Suppose that an n-dimensional DG is linear projected to an $(n-1)$-dimensional SFG array of size $L_1 \times L_2 \times \cdots \times L_{n-1}$. The SFG is partitioned into $M_1 \times M_2 \times \cdots \times M_{n-1}$ blocks, where each block is of size $Z_1 \times Z_2 \times \cdots \times Z_{n-1}$. $Z_i = L_i/M_i$ for $i \in \{1, 2, \cdots, n-1\}$,

Allocation.

1. In the LSGP scheme, one block is mapped to one PE. Each PE sequentially executes the nodes of the corresponding block. The number of blocks is equal to the number of PEs in the array, i.e., the array size equals to the product $M_1 \times M_2 \times \cdots \times M_{n-1}$.
2. In the LPGS scheme, the block size is chosen to match the array size, i.e., one block can be mapped to one array. All nodes within one block are processed concurrently, i.e., locally parallel. One block after another block of node data is loaded into the array and processed in a sequential manner, i.e., globally sequential.

Scheduling. In LSGP, after processor allocation, from the processor sharing perspective, there are $Z_1 \times Z_2 \times \cdots \times Z_{n-1}$ nodes in each block in the SFG, which share one PE. An acceptable (i.e., sufficiently slow) schedule is chosen so that at any instant there is at most one active PE in each block.

As to the scheduling scheme for the LPGS method, a general rule is to select a (global) scheduling that does not violate the data dependencies. Note that the LPGS design has the advantage that blocks can be executed one after another in a natural order. However, this simple ordering is valid only when there is no reverse data dependence for the chosen blocks.

Generalized Partitioning Method. A unified partitioning and scheduling scheme is proposed for LPGS and LSGP in [9]. The main contribution includes a unified partitioning model and a systematic two-level scheduling scheme. The unified partitioning model can support LPGS and LSGP design in the same manner.

The systematic two-level scheduling scheme can specify the intra-processor schedule and inter-processor schedule independently. Hence, more inter-processor parallelism can be effectively explored.

A general frame work for processing mapping is also proposed in [17, 18].

Optimization for Partitioning. The problem of finding an optimal (or reasonably small) schedule is a NP-hard problem. A systematic methodology for optimal partitioning is described in [23].

Acknowledgements

This work was supported in part by Sarnoff Research Center, Mitsubishi Electric, and the George Van Ness Lothrop Honorific Fellowship.

References

1. J. Baek, S. Nam, M. Lee, C. Oh, and K. Hwang, "A Fast Array Architecture for Block Matching Algorithm," *Proc. of IEEE Symposium on Circuits and Systems*, vol. 4, pp. 211–214, 1994.
2. S. Chang, J.-H. Hwang, and C.-W. Jen, "Scalable Array Architecture Design for Full Search Block Matching," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 5, no. 4, pp. 332–343, Aug. 1995.
3. Y.-K. Chen and S. Y. Kung, "An Operation Placement and Scheduling Scheme for Cache and Communication Localities in Fine-Grain Parallel Architectures," in *Proc. of Int'l Symposium on Parallel Architectures, Algorithms and Networks*, pp. 390–396, Dec. 1997.
4. L. De Vos, "VLSI-architectures for the Hierarchical Block-Matching Algorithm for HDTV Applications," *SPIE Visual Communications and Image Processing*, vol. 1360, pp. 398–409, 1990.
5. L. De Vos and M. Stegherr, "Parameterizable VLSI Architectures for Full-Search Block-Matching Algorithm," *IEEE Trans. on Circuits and Systems*, vol. 36, no. 10, pp. 1309–1316, Oct. 1989.
6. D. Le Gall, "MPEG: A Video Compression Standard for Multimedia Applications," *Communications of the ACM*, vol. 34, no. 4, Apr. 1991.
7. K. Guttag, R. J. Gove, , and J. R. V. Aken, "A Single-Chip Multiprocessor For Multimedia: The MVP," *IEEE Computer Graphics & Applications*, vol. 11, no. 6, pp. 53–64, Nov. 1992.
8. C.-H. Hsieh and T.-P. Lin, "VLSI Architecture for Block-Matching Motion Estimation Algorithm," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 2, no. 2, pp. 169–175, June 1992.
9. Y.-T. Hwang and Y.-H. Hu, "A Unified Partitioning and Scheduling Scheme for Mapping Multi-Stage Regular Iterative Algorithms onto Processor Arrays," *Journal of VLSI Signal Processing Applications*, vol. 11, pp. 133–150, Oct. 1995.
10. T. Komarek and P. Pirsch, "Array Architectures for Block Matching Algorithms," *IEEE Trans. on Circuits and Systems*, vol. 36, no. 10, pp. 1301–1308, Oct. 1989.
11. S. Y. Kung, *VLSI Array Processors*. Englewood Cliffs, NJ: Prentice Hall, 1988.
12. G.-J. Li and B. W. Wah, "The Design of Optimal Systolic Array," *IEEE Trans. on Computer*, vol. 34, no. 1, pp. 66–77, Jan. 1985.
13. N. L. Passos and E. H.-M. Sha, "Achieving Full Parallelism Using Multidimensional Retiming," *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, no. 11, pp. 1150–1163, Nov. 1996.
14. P. Pirsch, N. Demassieux, and W. Gehrke, "VLSI Architectures for Video Compression–A Survey," *Proceedings of the IEEE*, vol. 83, no. 2, pp. 220–246, Feb. 1995.
15. F. Sijstermans and J. van der Meer, "CD-1 Full-Motion Video Encoding on a Parallel Computer," *Communications of the ACM*, vol. 34, no. 4, pp. 81–91, Apr. 1991.
16. M.-T. Sun, "Algorithms and VLSI Architectures for Motion Estimation," *VLSI Implementations for Image Communications*, pp. 251–282, 1993.
17. J. Teich and L. Thiele, "Partitioning of Processor Arrays: a Piecewise Regular Approach," *INTEGRATION: The VLSI Journal*, vol. 14, no. 3, pp. 297–332, 1993.
18. J. Teich, L. Thiele, and L. Zhang, "Partitioning Processor Arrays under Resource Constraints," *Journal of VLSI Signal Processing*, vol. 17, no. 1, pp. 5–20, Sept. 1997.
19. W. F. Verhaegh, P. E. Lippens, E. H. Aarts, J. H. Korst, J. L. van Meerbergen, and A. van der Werf, "Improved Force-directed Scheduling in High-throughput Digital Signal Processing," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 8, pp. 945–960, Aug 1995.
20. B.-M. Wang, J.-C. Yen, , and S. Chang, "Zero Waiting-Cycle Hierarchical Block Matching Algorithm and its Array Architectures," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 4, no. 4, pp. 18–28, Feb. 1994.
21. Y. Wong and J.-M. Delosme, "Optimization of Computation Time for Systolic Array," *IEEE Trans. on Computer*, vol. 41, no. 2, pp. 159–177, Feb. 1992.
22. H. Yeo and Y.-H. Hu, "A Novel Modular Systolic Array Architecture for Full-Search Block Matching Motion Estimation," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 5, no. 5, pp. 407–416, Oct. 1995.
23. K.-H. Zimmermann, "A Unifying Lattice-Based Approach for the Partitioning of Systolic Arrays via LPGS and LSGP," *Journal of VLSI Signal Processing*, vol. 17, no. 1, pp. 21–47, Sept. 1997.
24. K.-H. Zimmermann, "Linear Mappings of n-Dimensional Uniform Recurrences onto k-Dimensional Systolic Array," *Journal of Signal Processing System for Signal, Image, and Video Technology*, vol. 12, no. 2, pp. 187–202, May 1996.
25. K.-H. Zimmermann and W. Achtziger, "Finding Space-Time Transformations for Uniform Recurrences via Branching Parametric Linear Programming," *Journal of VLSI Signal Processing*, vol. 15, no. 3, pp. 259–274, 1997.
26. K.-H. Zimmermann and W. Achtziger, "On Time Optimal Implementation of Uniform Recurrences onto Array Processors via Quadratic Programmin," *Journal of VLSI Signal Processing*, vol. 19, no. 1, pp. 19–38, 1998.