The Journal of VLSI Signal Processing—Systems for Signal, Image, and Video Technology, 20, 183–206 (1998)

Multimedia Signal Processors: An Architectural Platform with Algorithmic Compilation

YEN-KUANG CHEN AND S.Y. KUNG
Department of Electrical Engineering, Princeton University, Princeton, NJ 08544

Abstract. Novel algorithmic features of multimedia applications and advances in VLSI technologies are driving forces behind the new multimedia signal processors. We propose an architecture platform which could provide high performance and flexibility, and would require less external I/O and memory access. It is comprised of *array processors* to be used as the hardware accelerator and *RISC cores* to be used as the basis of the programmable processor. It is a hierarchical and scalable architecture style which facilitates the hardware-software codesign of multimedia signal processing circuits and systems. While some control-intensive functions can be implemented using programmable CPUs, other computation-intensive functions can rely on hardware accelerators.

To compile multimedia algorithms, we also present an operation placement and scheduling scheme suitable for the proposed architectural platform. Our scheme addresses data reusability and exploits local communication in order to avoid the memory/communication bandwidth bottleneck, which leads to faster program execution. Our method shows a promising performance: a linear speed-up of 16 times can be achieved for the block-matching motion estimation algorithm and the true motion tracking algorithm, which have formed many multimedia applications (e.g., MPEG-2 and MPEG-4).

1. Introduction

Multimedia signal processing involves the joint processing of digital information in various representations. It covers a very broad spectrum of applications [1, 2, 3, 4]:

- **Audio & speech processing:** audio compression (G.711, G.722, G.728), surround sound processing, AC-3, etc.
- **Image & video processing:** resolution conversion, image enhancement, image restoration, image compression (JBIG, JPEG), video compression (MPEG), etc.
- **Content-based indexing & retrieval:** feature extraction (fillet coordination, moment, histogram), pattern recognition, face detection/recognition, fusion of multi-modality, etc.
- **2D, 3D, & 4D graphics:** volume rendering, modeling transformation, texture mapping, shading, shadowing, ray-tracing, computer-assisted animation, virtual reality, etc.

In general, the following important design issues emerge for multimedia signal processing:

- High performance and high flexibility
- Low cost, low power, and efficient memory usage
- The ease of system integration or single-chip solution
- Fast design turn-around

These objectives may be best achieved by implementing multimedia signal processing systems in an application specific paradigm supplied by a comprehensive design methodology.

Algorithm and Architecture Codesign. For complex multimedia signal processing systems,

the inherent interaction of various design parameters comprising hardware and software issues must be taken into account. These issues are highly dependent on each other [5].

1. *We should bear the architectural style in mind when we design or choose the algorithm for a specific task.*

For a multimedia application, there are tons of different algorithms which can achieve roughly the same task. However, these algorithms have different characteristics in performance, computation requirement, and hardware implementation. Some of them take more computational time but perform better while some take less time but perform not so well. Furthermore, some of the algorithms are more efficiently implemented in the array processor while some are very easy to be implemented in commercial microprocessors. Choosing the right one depends on users' needs.

In other words, let algorithm $i = 1, \cdots, N$ be the solutions for a multimedia signal processing task and the execution time $T_i = T_{is} + \frac{T_{ip}}{P}$ where T_{is} is the non-parallelizable execution time, T_{ip} is the parallelizable execution time, and P the number of parallel execution units. In a uniprocessor system where $P = 1$, algorithm i is better if $T_{is} + T_{ip}$ is minimal. For a multiprocessor system, algorithm j is better if $T_{js} + \frac{T_{jp}}{P}$ is minimal. As P become larger and larger in the near future, the design/choice of algorithms should be changed.

Using motion estimation as a design example, we found that the full-search block-matching algorithm (BMA) is more efficiently implemented in systolic array than the hierarchical-search BMA although the full-search BMA needs more operations.

2. *We must also bear in mind the characteristics of multimedia signal processing algorithms when we design or choose the architecture.*

Different architecture can support different sets of algorithms. Systolic array can provide very high computing power for very regular and computationally intensive tasks while a programmable RISC core can deliver very complex logic tasks.

One of the main themes of this work is that, in Section 3, based on the common characteristics of multimedia signal processing algorithms and several architectural trends in multimedia signal processor, we present an architectural style for high-throughput multimedia signal processing [6, 7]. It is comprised of array processors and RISC cores. The processor arrays are built as the hardware accelerator of the platform so as to provide high performance. The RISC cores are built as the basis of the programmable processor so as to provide high flexibility.

Note that the key to success in a fixed-scheduling media processor (such as VLIW, SIMD) hinges on the success of the compiler. Similarly, the key components of the proposed implementation are the platform itself and a compiler to map applications efficiently on the platform (especially, on the array processors).

Another main theme of this work is that, in Section 4, based on a systematic systolic design methodology—multiprojection [8], we present an operation placement and scheduling scheme for the array processors [9]. The key advantages are twofold: (1) This multiprojection method, which deals with multidimensional parallelism systematically, can alleviate the burden of the programmer in coding and data partitioning. (2) It puts a lot of emphasis on cache localities and local communication in order to avoid the memory/communication bandwidth bottleneck, and can lead to faster program execution.

In addition, the whole design process is to map an application onto a pre-defined architecture rather than to design full-custom hardware. The verification process focuses on the functionality and performance of the application running on the target platform. As a result, the efforts in hardware/software codesign and co-verification are less than the efforts in conventional design processes because the platform has been pre-defined.

2. Fundamentals of Multimedia Signal Processors

The overall architectural design can be divided into *internal* and *external* design spaces. The internal design focuses on *core processor* upgrade,

Table 1. List of some announced programmable multimedia processors. (1) All of them use massive parallelism (SIMD, split-ALU, MIMD, or VLIW) and pipelines. (2) In general, the size of the operands for the ALUs or functional units is less than 32 bits. Some of the ALUs are the split-ALUs which can operate on multiple sets of operands in one instruction. (3) They have high-speed and high-bandwidth internal communication channels (400 MB/s to 18 GB/s). (4) They all have high-speed on-chip data memory (register file, cache, RAM). (5) They can provide high computing power (1 BOPS to 6 BOPS). (6) Their external memory bandwidths are very high (400 MB/s to 1200 MB/s).

Processor	Architecture	Size of datapath (bits)	Internal communication	Internal data memory (KB)	Peak performance (BOPS)	External memory bandwidth (MB/s)	Ref.
Chromatic Mpact 2	VLIW/SIMD (6 ALUs)	72	792-bit cross-bar (18 GB/s)	4	6.0	1200	[10]
NEC V830R/AV	RISC core + a SIMD (split-ALU) coprocessor	32/ 64	64-bit bus (1.6 GB/s)	16	2.0	600	[19]
Philips TriMedia	VLIW (27 FUs)	32	32-bit bus (400 MB/s)	16	4.0	400	[36]
TI 'C62x	VLIW (6 ALUs + 2 multipliers)	32/ 16	32-bit bus (800 MB/s)	64	1.6	800	[17]
TI 'C8x	4 split-ALU DSPs + RISC core (MIMD)	32/ 32	crossbar (2.4 GB/s)	36	2.1	480	[15]

while the external design focuses on *accelerators* that off-load tasks from the main core.

Several alternatives exist to exploit the parallelization potential of multimedia signal processing algorithms for programmable architectures (cf. Table 1):

1. Single Instruction Stream, Multiple Data Streams (external SIMD):
   Aiming at data parallelism, SIMD (Single Instruction Stream, Multiple Data Streams) architectures are characterized by several data paths executing the same operation on different data entities in parallel. While thus a high degree of parallelism can be achieved with little control overhead, data path utilization rapidly decreases for scalar program parts. In general, pure SIMD architectures are not an efficient solution for complex multimedia applications; they are best suited for algorithms with highly regular computation patterns. For example, Chromatic's Mpact 2, which can deliver 6 BOPS[1], is a mixture of SIMD and VLIW [10] instead of pure SIMD.
2. Split-ALU (internal core-processor SIMD):
   Architectures featuring a split-ALU are based on a principle similar to SIMD: a number of lower-precision data items are processed in parallel on the same ALU. Figure 1 shows a possible implementation of the split-ALU concept. The advantage of this approach is its small incremental hardware cost provided that a wide ALU is already available. Recent multimedia extensions of general-purpose processors are typically based on this principle, e.g., MAX-2 for HP's PA-RISC [11], VIS for SUN's UltraSparc [12], MMX for Intel's x86 [13].

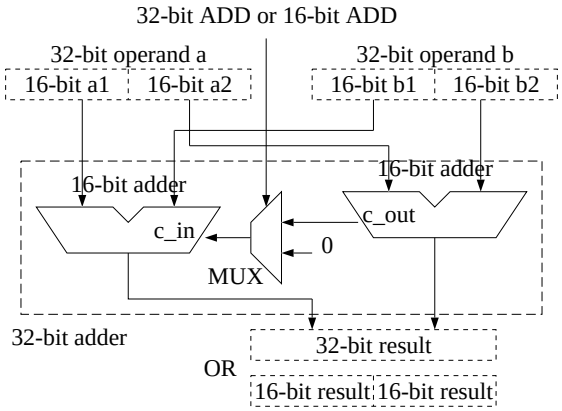

Fig. 1. An example of the split-ALU implementation. A 32-bit adder can work as two 16-bit adders, which add two pairs of 16-bit operands. The only difference between the two functionalities of this adder is the carry propagation from the lower 16-bit adder to the upper 16-bit adder. Splitting this 32-bit adder into two 16-bit adders allows one single instruction to process multiple data. This data parallelism (also called subword parallelism) is very similar to the SIMD architecture.

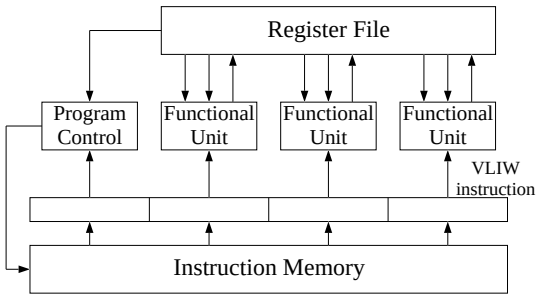

Fig. 2. A generic VLIW architecture. A very-long-instruction-word architecture consists of multiple functional units (FUs). An issue of the VLIW instruction can activate multiple FUs to operate independently on multiple sets of operands.

3. Multiple Instruction Streams, Multiple Data Streams (external MIMD):
   Task level as well as data level parallelism can be exploited by MIMD (Multiple Instruction Streams, Multiple Data Streams) architectures, which are characterized by a number of parallel data paths featuring individual control units [14]. Thus, MIMD processors offer the highest flexibility for algorithms to be executed in parallel. For example, TI's TMS320C80 [15], which can deliver 2 BOPS, is a MIMD processor. However, MIMD processors incur a high hardware cost for multiple control units as well as for a memory system delivering the sufficient bandwidth to supply all required instruction streams. Furthermore, synchronization difficulties and poor programmability have prevented MIMD processors from widespread use in multimedia applications so far.
4. Very Long Instruction Word (internal coreprocessor MIMD):
   Instruction level parallelism is targeted by VLIW (Very Long Instruction Word) architectures, which specify several operations within a single long instruction word to be executed concurrently on multiple functional units (Fig. 2) [16]. In contrast to superscalar architectures, VLIW processors must rely on static instruction scheduling performed at the compilation time. The advantage is a simplified design since no hardware support for dynamic code reordering is required. For example, TMS320C6201 [17], a general-purpose programmable fixed-point DSP adopting a Very Long Instruction Word (VLIW) implementation, can deliver 1600 MIPS[2].

```
; Assume ``r0'' holds 0
;        ``r5'' holds 255
      cmp   r1,r5     ; If r1 >= 255,
      bge   _max      ;    go to _max.
      mov   r5,r1     ; Clip to 255.
      br    _next
_max: cmp   r0,r1     ; If r1 <= 0,
      bge   _next     ;    go to _next.
      mov   r0,r1     ; Clip to 0.
_next:
```

(a)

```
      min3  r1,r5,r1  ; If r1 > 255, r1 = 255.
      max3  r0,r1,r1  ; If r1 < 0, r1 = 0.
```

(b)

Fig. 3. Specialized instructions replace sequences of standard instructions: for example, the instruction stream for minimum maximum operations on the V810 (a) compared to the V830 (b). By introducing a single new instruction comprising a frequently executed sequence of standard instructions, the instruction count of multimedia code can be reduced significantly [18].

Simultaneously, another widely employed way of adapting programmable processors to special multimedia signal processing algorithm characteristics is to introduce specialized instructions for frequently recurring operations of higher complexity, e.g., a multiply-accumulate operation with saturation [18]. By replacing longer sequences of standard instructions, the use of specialized instructions may significantly reduce the instruction count, resulting in faster program execution (Fig. 3). The design complexity required for implementing specialized instructions can usually be kept at modest levels; the decision about which instructions to implement depends on the probability of their use.

3. Architectural Platform for High-Throughput Multimedia Signal Processing

Multimedia signal processor design should be driven by algorithmic features in multimedia applications. From algorithmic perspectives, impor-

tant characteristics of the multimedia signal processing algorithms can be summarized as following:

1. *Intensive computation for highly regular operations*
   Computation-intensive applications usually depend on a loop of instructions. There is a huge amount of computations for highly regular operations. There is a great deal of parallelism on common operations, such as addition, subtraction, and multiplication. Therefore, parallels and pipelines should be exploited in the multimedia architecture, as shown in Section 2.
2. *Intensive I/O or memory access*
   There is a huge amount of I/O or memory access in multimedia applications. Hence, a multimedia signal processor should be able to support a high memory bandwidth (cf. Table 1). Because multimedia data operands have very frequent and very regular reusability, a good architecture should make good use of the data reusability.
3. *Frequent execution of small integer operands*
   In MPEG and other pixel-oriented algorithms, the data being operated on are small integers (such as, 8-bit or 16-bit), narrower than the existing integer data paths of microprocessors. Small processing elements or subword parallelism must be exploited for higher efficiency, e.g., HP's PA-RISC [11], Intel's x86 [13], NEC's V830R/AV [19], SUN's UltraSparc [12], TI's C80 [15].
4. *High control complexity in less computationally intensive tasks*
   There are also some high control complexity tasks which are less time-consuming. It may be more efficient and economical to resort to software solutions for such tasks. Therefore, flexible RISC cores (master processors) are preferred, e.g., NEC's V830R/AV [19] and TI's C80 [15].

Multimedia signal processor design should also be driven by available VLSI technologies. There are two important features in VLSI technologies:

1. *External memory is slow*
   There is a huge gap between memory speed and processor speed. Therefore, a high-speed on-chip data memory (register file, cache, RAM) is necessary to bridge the gap. For example, most of the announced programmable media processors (as listed in Table 1) use 16 KB to 64 KB on-chip data memory.
2. *Long-distance communication is slow*
   Because the feature size of the processing technology is getting smaller and smaller, more and more of the signal delay is on the wire than the transistor [20]. Long-distance and global (one to many) communication takes longer and is more expensive than local communication. Hence, for a sound design, it is important to make use of local communication channels and it is necessary to support local communication efficiently.

Conventional standard processors do not correspond well to those characteristics of multimedia signal processing algorithms. Therefore, special architectural approaches are necessary for multimedia processors to deliver the required high processing power with efficient use of hardware resources.

It is generally agreed that some multimedia signal processing functions can be implemented using programmable CPUs (software solutions) while others must rely on hardware accelerators (hardware solutions) [21]. A sound multimedia signal processing architecture style should base on this principle. We propose an architecture style for high-performance multimedia signal processing as shown in Fig. 4 which is built upon some earlier platforms proposed by [14, 22, 23, 24]. *It consists of array processors used as the hardware accelerator and RISC cores as the basis of the programmable processor.* The programmable processor provides software solutions which mean high flexibility while the accelerator provides hardware solutions for high performance.

The processing array in our architecture platform has three unique features. (1) Every processing unit (PU) is very small, 8-bit or at most 16-bit. (2) Every PU has its own local data memory/cache. The local caches have an external control protocol. For example, the program can ask the caches not to cache some part of the data [25]. (3) There is a local bus between two consecutive PUs. Hence, the PUs can talk to each other in two

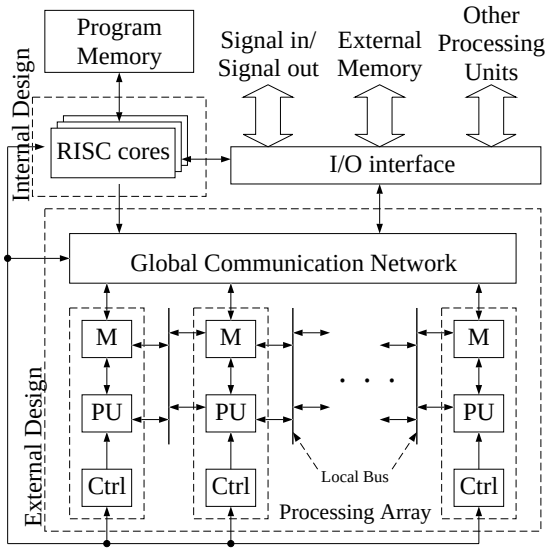

Fig. 4. Proposed architectural style for high performance multimedia signal processing. There are two main components: (1) *processor arrays* to be used as the hardware accelerator for computationally intensive and regular components in an algorithm, and (2) *RISC cores* to be used as the basis of the programmable processor for complex but less computationally intensive components. M stands for the local memory. PU stands for the processing unit. Ctrl stands for the control unit.

ways: (a) via the local bus between them, and (b) via the global communication channel, which may be a bus or a crossbar network.

These unique features provide four advantages. (1) A high percentage of operands are 8-bit or 16-bit integers in MPEG and other pixel-oriented algorithms. Without careful design, more than half of the data path is wasted in the current 64-bit microprocessors. Therefore, the data path on our PU is designed to be 8-bit or 16-bit. In addition, since the PU is simpler, the area is smaller. We can pack more small PUs into a single chip than large PUs. (2) The circuit delay of the small PU is smaller than the delay of the large PU. The clock cycle time of the small PU is smaller than the cycle time of the large PU. (3) The local data memory can provide very high data throughput. (4) The local communication can provide very high communication bandwidth between two consecutive PUs at a very low cost (in terms of area, power, and delay).

It is critical to note that *multimedia signal processor designs should be supported by algorithmic partitioning of multimedia applications.* In order to have an effective execution, given a specific application, the algorithm is first manually or semi-automatically divided into two parts (cf. Fig. 5):

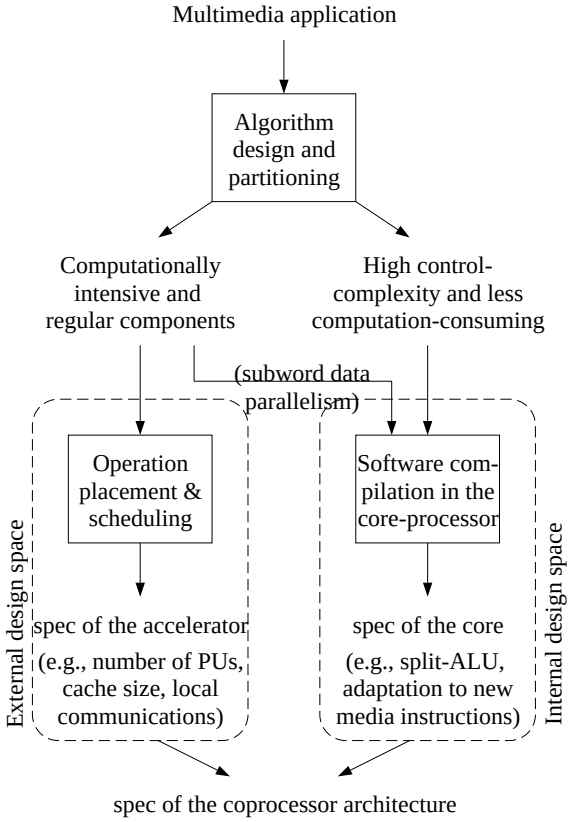

Fig. 5. The proposed algorithm and architecture codesign approach for multimedia applications. In order to have an effective execution, given a specific application, the algorithm is first manually or semi-automatically divided into two parts: (1) computationally intensive and regular components, for which a hardware solution is preferred (e.g., motion estimation, DCT, IDCT), and (2) complex but less computationally intensive components, for which a software solution is preferred (e.g., VLC, VLD, rate control). From the results of the automatic operation placement and scheduling scheme, we can determine the spec of the accelerators, such as the number of PUs, the size of the datapath, the size of the local data memory. Combining the spec of the accelerators and the results of the core-processor adaptation, we can determine the final spec of the architecture.

1. *Computationally intensive and regular components*, for which a hardware solution is preferred.
2. *Complex but less computationally intensive components*, for which a software solution is preferred.

Computationally Intensive and Regular Components. A systematic multimedia signal processing mapping method can facilitate design of processor arrays for computationally intensive and regular components. Since massive parallel and pipelined computing engines can provide very high computational power for regular, intensive operations, various formal systematic procedures for systolic designs of many classes of algorithms have been proposed [26]. These transfer the computationally intensive, regular operations into simple processing elements, each with a fixed, data-independent function, along with a one- or two-dimensional nearest-neighbor communication pattern. These are the basic components of our design methodology for the multimedia signal processing system.

One major design objective is to make sure that the speed of the external memory keeps up with the speed of the processing engine. As shown in Section 4, the proposed approach is to fully exploit the very frequent read-after-read data dependence (i.e., transmittent data) [8, 9]. By exploiting the locality, our allocation and scheduling reduces the communication-to-computation ratio, and hence reduces the amount of the external memory access/communication. The performance is enhanced, since the contention problem on the global communication network can be substantially alleviated.

In short, this architecture adopts systolic-type communication to speed up the computation since localized communication is faster. Moreover, this architecture reduces power consumption because it (1) segments global communication in local buses, (2) provides local, dedicated connection links, and (3) distributes control logics to individual PUs.

Complex but Less Computationally Intensive Components. The complex but less computationally intensive components (e.g., controlling, data-dependent tasks) are supported by the software solution on RISC cores. Minor modification to improved multimedia processing algorithms can be achieved by software updates. For example, different video coding standards can be implemented using the same hardware.

In addition, the processor arrays may also be efficiently utilized for some special functions. All functions, no matter how complicated, can be expressed as a combination of Boolean and arithmetic operations. In our design, the PUs are (1) simple, but at the same time (2) flexible (i.e., reconfigurable/programmable), as compared to conventional fix-function systolic PUs.

The built-in functions in PUs include integer adder/subtracter/comparator, multiplier, shifter, logic AND, logic OR, etc. Similar to microcoded architectures or custom computing machines, we can adopt pre-compiled macro-instruction sequences to implement special functions (e.g., mean filter [27]). In order to handle 64-bit or larger integer or floating-point operations, coordination among multiple PUs can be used. (Hence, the communication among these PUs becomes a burden of the design.)

Our design philosophy of this *merged-ALU* is similar to the philosophy of split-ALUs. The philosophies are similar because both our design and the split-ALU design can process multiple 8-bit operands, 16-bit operands, 32-bit operands, and even 64-bit operands by orchestrating a set of small ALUs.

On the other hand, the philosophies are different in determining the clock cycle time. (1) It takes the same time for the split-ALU design to process multiple 8-bit operands as it takes to process 32-bit operands. Therefore, the longest carry chain (say, 32 bits) of the split-ALU determines the clock cycle time. (2) It takes one cycle for our design to process 8-bit operands but it takes 4 pipeline cycles for our design to process 32-bit operands. Therefore, the 8-bit (shortest) carry chain determines the clock cycle time.

4. Systematic Operation Placement and Scheduling Method

In Section 3, we have presented an architecture platform that can be configured to perform a variety of application-specific functionalities. *The success of the proposed architectural platform depends on the efficient mapping of an application onto the target platform.*

Over the years, a variety of formal systematic procedures for systolic designs of many classes of algorithms have been proposed [26]. The procedures transfer the computationally intensive, reg-

ular operations into simple processing elements, each with a fixed, data-independent function, along with a one- or two-dimensional nearest-neighbor communication pattern. Another main theme of this work is that *we present a systematic operation placement and scheduling method (similar to systolic design approaches) for the execution of the computationally intensive and regular components in the proposed processing arrays* [9].

There are 3 stages in common systolic design methodology: the first one is dependence graph (DG) design[3], the second one is mapping the DG to a signal flow graph (SFG)[4], and the third one is design array processor based on SFG. 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 easily can transform an SFG to a systolic array, wavefront array, SIMD, or MIMD. Therefore, most research nowadays is focused 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, for example, might be to minimize the amount of communication—exchanges of data—between processors.)
2. **Scheduling:** In what order should the operations be assigned to a processor? (A criterion might be to minimize total computing time.)

Therefore, 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 very general; however, in order to derive a regular systolic array, linear assignments and scheduling attract more attention.

Similar to systolic design approaches, in this section, we present a systematic multimedia signal processing mapping method that can facilitate the design of processor arrays for computationally intensive and regular components. To ensure an effective program, the cache locality is important because of the large speed gap between microprocessors and memory systems. It is also important to make use of local communication whenever possible, since it is cheaper, faster, and less power hungry than global communication.

Different data placement and operation scheduling would need different cache size requirement and global/local communication. We observe that although input dependence imposes no ordering constraints, input dependence does reveal the critical information on the data localities. To maximize the hit ratio of the caches, such information should be utilized for better data placement and operation scheduling by the parallel compilers. The proposed systematic code scheduling method has the following features:

1. Our multiprojection method deals with high-dimensional parallelism systematically. It can alleviate the burden of the programmer in coding and data partitioning.
2. It generates a fine grain parallelism code which has low latency.
3. It exploits good temporary localities so that the utilization rate of caches is high.
4. It also exploits good spatial localities which are good for new parallel architectures where localized communication is cheaper than global communication (cf. Fig. 4).

4.1. Algebraic Formulation of Multiprojection

The process of multiprojection can be written as a number of single projections using the same algebraic formulation as introduced in [8, 26, 28].

1. Let the n-dimensional SFG be defined as the n-dimensional DG. In other words, $\underline{n}_n(\underline{c}_x) = \underline{c}_x$ and the $\vec{m}_n(\vec{e}_i) = \vec{e}_i$ where $\underline{c}$ represents a node in the DG, $\vec{e}$ represents a data dependence in the DG, $\underline{n}$ represents a node in the SFG, and $\vec{m}$ represents an edge in the SFG.
2. We project the l-dimensional SFG into $(l-1)$-dimensional SFG by the 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 $\vec{s}_l^T \vec{d}_l > 0$ and $\mathbf{P}_l \vec{d}_l = 0$.
   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

$$\underline{n}_{l-1}(\underline{c}_i) = \mathbf{P}_l \underline{n}_l(\underline{c}_i) \quad (1)$$
$$\vec{m}_{l-1}(\vec{e}_i) = \mathbf{P}_l \vec{m}_l(\vec{e}_i) \quad (2)$$

3. After $(n-k)$ projections, the results can be combined as the following. 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

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

where $M_l \geq 1 + (N_l - 1)\vec{s}_{l-1}^T \vec{d}_{l-1}$ where N_l is the maximum number of nodes along the $\vec{d}_{l-1}$ direction in 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:

$$\begin{bmatrix} D_k(\vec{e}_i) \\ \vec{m}_k(\vec{e}_i) \end{bmatrix} = \begin{bmatrix} \mathbf{S}^T \\ \mathbf{A} \end{bmatrix} \vec{e}_i \quad (6)$$

where T represents the execution time of the node and D represents the delay of the edge.

4.2. Data and Processor Availability

Every dependent datum comes from previous computation. To ensure data availability, every edge must have at least one delay unit if the edge is not for broadcasting data, i.e.,

$$\mathbf{S}^T \vec{e}_i > 0 \quad \forall \vec{e}_i \quad (7)$$

Two computation nodes that are mapped into a same processor could not be executed at the same time. To ensure processor availability,

$$\mathbf{S}^T \underline{c}_i \neq \mathbf{S}^T \underline{c}_j \quad \forall \underline{c}_i \neq \underline{c}_j, \mathbf{A}\underline{c}_i = \mathbf{A}\underline{c}_j \quad (8)$$

4.3. Optimization in Multiprojection

In this operation placement and scheduling scheme, the first step is to find an allocation $\mathbf{A}$ so that both of the following are satisfied. (1) A node in the SFG corresponds to one unique processor, i.e.,

$$\underline{n}_k(\underline{c}_i) \Rightarrow p_i \quad \forall \underline{n}_k(\underline{c}_i) \in SFG$$

(2) The amount of the global communication is minimized, i.e.,

$$\min_{\mathbf{A}}(\max_{\vec{e}_i}\{\mathbf{A}(\vec{e}_i)\}) \quad \forall \vec{e}_i \in DG$$

After projection directions are fixed, the structure of the array is determined. The remaining part of the design is to find a scheduling that (1) can complete the computation in minimal time and (2) can use a minimal-size cache under processor and data availability constraint, i.e.,

$$\begin{cases} \min\limits_{\mathbf{S}} \left(\max\limits_{\underline{c}_x, \underline{c}_y} \{\mathbf{S}^T(\underline{c}_x - \underline{c}_y)\} \right) & \forall \underline{c}_x, \underline{c}_y \in DG \\ \min\limits_{\mathbf{S}} \left\{ \sum\limits_{\vec{e}_i} \mathbf{S}^T \vec{e}_i \right\} & \forall \vec{e}_i \in DG \end{cases}$$

4.4. Graph Transformation Rules

Besides the above multiprojection optimization, we also provide some graph transformation rules for better design which can help us to reduce the amount of communication between processors, the size of buffer, or the power consumption. Table 2 is a brief summary [8].

5. Implementation of Block-Matching Motion Estimation Algorithm

Video compression plays an important role in many applications, such as, video-conferencing, video-phone, etc. The key to achieve compression is to remove temporal and spatial redundancies in video sequences. Block-matching algorithms (BMAs) have been widely exploited in various international video compression standards to remove temporal redundancy.

As shown in Fig. 6, the basic idea of the BMA is to locate a displaced candidate block that is most similar to the current block, within the search area

Table 2. Equivalent graph transformation rules for design optimization [8]. The transmittent data [26], which are used repeatedly by many computation nodes in the DG, are extremely critical in these rules.

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

in the previous frame. Various similarity measurement criteria have been presented for block matching. The most popular one is the sum of the absolute differences (SAD) as

$$\mathrm{SAD}[u,v] = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} \mid s[i+u, j+v] - r[i,j] \mid \\ -p \le u < p,\ -p \le v < p \qquad (9)$$

where n is the block width and height, p is the absolute values of the maximum possible vertical and horizontal motion, $r[i,j]$ is the luminance value (pixel intensity) in the current block at coordinates (i,j), $s[i+u, j+v]$ is the luminance value in the search area in the previous frame, and (u,v) represents the candidate displacement vector. The motion vector is determined by the least $\mathrm{SAD}[u,v]$ for all possible displacements (u,v) within a search area, as shown here:

$$\text{Motion Vector} = \arg\min_{[u,v]}\{\mathrm{SAD}[u,v]\} \qquad (10)$$

The operations of a BMA are very simple—additions and subtractions. However, BMAs are known to be the main bottleneck in real-time encoding applications. For example, 6.2×10^9 additions per second and 3.1 GB/sec external memory access would be required for a real-time MPEG-1

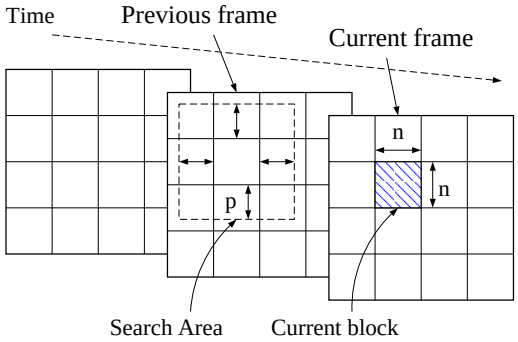

Fig. 6. In the process of the block-matching motion estimation, the current frame is divided into a number of non-overlapping current blocks, which are (n pixels) × (n pixels). Each of them will be compared with $2p \times 2p$ different displaced blocks in the search area of the previous frame.

```
for (u = -p; u < p; u++)
    for (v = -p; v < p; v++)
    {
        SAD[u, v] = 0;
        for (i = 0; i < n; i++)
            for (j = 0; 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 (SADmin > SAD[u, v])
        {
            SADmin = SAD[u, v];
            MV = [u, v];
        }
```

Fig. 7. The pseudo code of the BMA for a single current block. In the process of the block-matching motion estimation, the current frame is divided into a number of non-overlapping current blocks, which are (n pixels) × (n pixels). Each of them will be compared with $2p \times 2p$ different displaced blocks in the search area of the previous frame. SAD is the sum of the absolute differences between the current block and the displaced block in the search area. The motion vector is the displacement which carries the minimal SAD.

video coding, when there are 30 frames per second with frame size 352 pixels × 288 pixels and search range $\{-16, \cdots, +15\} \times \{-16, \cdots, +15\}$. The search for an effective implementation has been a challenging problem for years.

Figure 7 shows the pseudo code of the BMA of a single current block. We first concentrate on the first half, calculating the SADs (cf. Eq. (9)). Instead of viewing the BMA with only two-dimensional read-after-write data dependence, we consider that the BMA has four-dimensional read-after-read input dependence. Figure 8 shows the 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 four-dimensional space i, j, u, v. The following is the core in the 4D DG of the BMA:

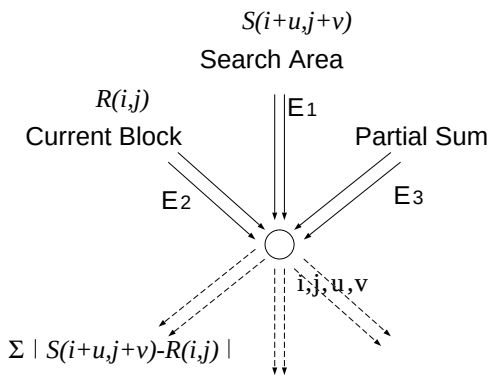

Fig. 8. A core in the 4D DG of the BMA. There are $n \times n \times 2p \times 2p$ nodes in the DG. The node i, j, u, v represents the computation, `SAD[u, v] = SAD[u, v] + | s[i + u, j + v] - r[i, j] |`. $\vec{E}_1$ denotes the *read-after-read* data dependence of the search window. The $s[i+u, j+v]$ will be used repeatedly for (1) different i, j, (2) same $i+u$, and (3) same $j+v$. $\vec{E}_1$ is a two-dimensional reformable mesh. One possible choice is $[1, 0, -1, 0]^T$ and $[0, 1, 0, -1]^T$. The $r[i, j]$ will be used repeatedly for different u, v. Hence, $\vec{E}_2$, the data dependence of the current block, could be $[0, 0, 1, 0]^T$ and $[0, 0, 0, 1]^T$. The summation can be done in i-first order or j-first order. $\vec{E}_3$, which accumulates the difference, could be $[1, 0, 0, 0]^T$ and $[0, 1, 0, 0]^T$. The representation of the DG is not unique; most of the dependence edges can be redirected because of data transmittance. Although read-after-read data dependence is not "real" data dependence (does not affect the execution order of the operations), the read-after-read data dependence can identify memory and communication localities.

Search window ($\vec{E}_1$)	$[1, 0, -1, 0]^T$	$D_4 = 0$
	$[0, 1, 0, -1]^T$	$D_4 = 0$
Current blocks ($\vec{E}_2$)	$[0, 0, 1, 0]^T$	$D_4 = 0$
	$[0, 0, 0, 1]^T$	$D_4 = 0$
Partial sum of SAD ($\vec{E}_3$)	$[1, 0, 0, 0]^T$	$D_4 = 0$
	$[0, 1, 0, 0]^T$	$D_4 = 0$

The index i, j $(0 \le i, j < n)$ are the indices of the pixels in a current block. The u and v $(-p \le u, v < p)$ are the indices of the potential displacement vector. The actual DG would be a four-dimensional repeat of the same core.

5.1. *Multiprojecting the 4D DG of the BMA to a 1D SFG*

From this 4D DG, we design a 1D SFG that can easily be implemented in the 1D processing array shown in Fig. 2. First, we project the 4D DG along v, u, and j direction, using

$\vec{d}_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$	$\vec{s}_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$	$P_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
$\vec{d}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$	$\vec{s}_3 = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$	$P_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$
$\vec{d}_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$	$\vec{s}_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$	$P_2 = \begin{bmatrix} 1 & 0 \end{bmatrix}$

To ensure processor availability, $M_4 = 2p$ and $M_3 = n$. Therefore,

$$\mathbf{A} = \mathbf{P_2 P_3 P_4} = [1, 0, 0, 0] \quad (11)$$

$$\mathbf{S}^T = \vec{s}_2^T \mathbf{P_3 P_4} + M_3 \vec{s}_3^T \mathbf{P_4} + M_3 M_4 \vec{s}_4^T = [n+1, 1, n, 2pn] \quad (12)$$

Therefore, we have

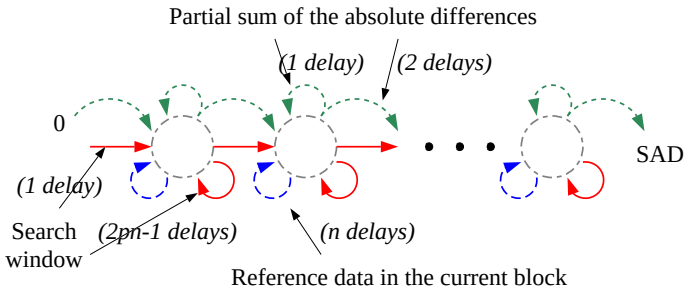

Fig. 9. The SFG from multiprojecting the 4D DG of the BMA.

Search window ($\vec{E}_1$)	Current blocks ($\vec{E}_2$)	Partial sum of SAD ($\vec{E}_3$)
1 ($D_1 = 1$)	0 ($D_1 = n$)	1 ($D_1 = 1 + n$)
0 ($D_1 = 1 - 2pn$)	0 ($D_1 = 2pn$)	0 ($D_1 = 1$)

Because the edge $\vec{E}_1$, $[0, 1, 0, -1]^T$, has negative delay, we apply the redirection rule to it. Therefore, the new delay will be $(2pn - 1)$. Because the edge $\vec{E}_3$, $[1, 0, 0, 0]^T$, has too many units of delay, we apply the reformation rule to it so that the new delay would be 2 units. Note that the edge $\vec{E}_2$, $[0, 0, 0, 1]^T$, and the edge $\vec{E}_2$, $[0, 0, 1, 0]^T$, has the same transmittent direction. In addition, the former is a multiple of the latter. Hence, the former can be eliminated. The final SFG becomes the following:

Search window ($\vec{E}_1$)	Current blocks ($\vec{E}_2$)	Partial sum of SAD ($\vec{E}_3$)
1 ($D_1 = 1$)	0 ($D_1 = n$)	1 ($D_1 = 2$)
0 ($D_1 = 2pn - 1$)		0 ($D_1 = 1$)

which can be visually seen in Fig. 9.

5.2. Interpretation of the SFG

The search window data dependence, passed from PU_i to PU_{i+1}, has 1 unit of delay. As a result, we do not need a global broadcasting of the search window. Using local communication is faster and less power demanding.

In the mean time, the search window data dependence, passed from PU_i to itself, has $(2pn - 1)$ units of delay. Consequently, we can expect that the same data will be reused after $(2pn - 1)$ operations. Using a cache with size $(2pn - 1)$ bytes is enough for this scheduling. For example, the cache size is 0.5 K-Bytes for $n = 16$ and $p = 16$. (Note that it is independent on the frame size.)

The reference data of the current block always stay in the same PU. There are 16 bytes for each PU. They can be put into either the cache or registers.

The summation of SAD, which is read-after-write data dependence, has two directions. The one which is inside PU_i itself has one unit of delay. That is, it is used immediately one after one to collect all of SAD in terms of loop j. The other one, which is passed from PU_i to PU_{i+1}, collects all the partial sum of SAD in terms of loop i. Because it has two units of delay, the data passing is not synchronous. (The systolic implementation of the SFG is shown in Fig. 10.)

The program can be divided into 4 parts:

1. First initialization loop, where there are no reference data of the current block and search window data in the PU, as shown in Fig. 11.
2. Initialization loops with cold caches, where there are reference data of the current block in the PU but there are no search window data in the PU, as shown in Fig. 12.
3. Full-speed pipeline with few cache misses, where reference data and most of the search window data are in the cache. However, the very last search window datum is new (cf. Fig. 13).

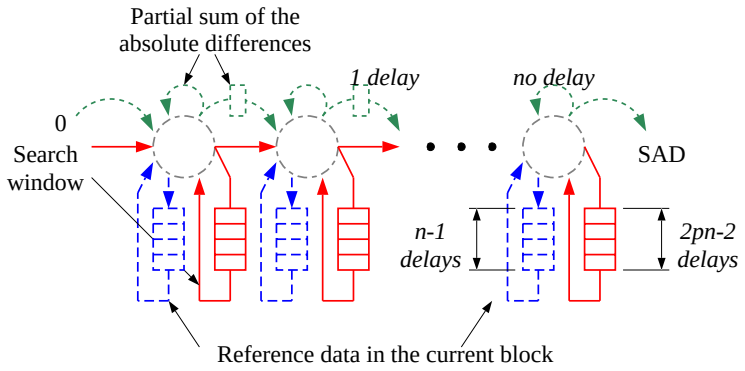

Fig. 10. The systolic implementation of the SFG from multiprojecting the 4D DG of the BMA (cf. Fig. 9).

PU_0	PU_1	PU_2	$\cdots$	PU_{n-1}
v=-p	- (idle)	- (idle)	-	- (idle)
u=-p	-	-	-	-
j=0	-	-	-	-
SAD[u,v]=0	-	-	-	-
get(s[i+u,j+v])	-	-	-	-
get(r[i,j])	-	-	-	-
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	-	-	-	-
j=1	-	-	-	-
get(s[i+u,j+v])	-	-	-	-
get(r[i,j])	-	-	-	-
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	-	-	-	-
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Fig. 11. A "source-level" representation of the code assigned to the processor i ($0 \leq i < n$) during the initialization loops. Note that there is a `get(r[i,j])` from $j = 0$ to $j = n - 1$ when $u = -p$. When there is a mark like get(r[i,j]), it denotes an external memory operation.

4. Full-speed pipeline with cache filled-up, where reference data and search window data are already in the cache (cf. Fig. 14).

PU_i is one operation ahead of the PU_{i+1} in terms of j and one loop ahead in terms of u.

5.3. Implementation

For cache and communication localities, it is important to maximize the exploitation of read-after-read input dependence. Therefore, our multi-dimensional projection method for operation placement and scheduling is introduced.

Table 3 shows the comparison among several placement and scheduling results using 16 PUs. Our placement and scheduling result (obtained by multiprojecting the 4D DG of the BMA) reduces the amount of the external memory access by 95 times. With an 8-KB cache and 33 MB/sec local communication, the required bandwidth of the external memory access is more practical although this placement and scheduling takes more cycles (1.7%).

PU_0	PU_1	PU_2	$\cdots$	PU_{n-1}
send(SAD[u,v])	- (idle)	- (idle)	-	- (idle)
v=-p	- (idle)	-	-	-
u=-p+1	v=-p	-	-	-
j=0	u=-p	-	-	-
SAD[u,v]=0	j=0	-	-	-
get(s[i+u,j+v])	get(SAD[u,v])	-	-	-
send(s[i+u,j+v])	get(s[i+u,j+v])	-	-	-
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	get(r[i,j])	-	-	-
j=1	SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	-	-	-
get(s[i+u,j+v])	j=1	-	-	-
send(s[i+u,j+v])	get(s[i+u,j+v])	-	-	-
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	get(r[i,j])	-	-	-
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
v=-p	send(SAD[u,v])	-	-	-
u=-p+2	v=-p	-	-	-
j=0	u=-p+1	v=-p	-	-
SAD[u,v]=0	j=0	u=-p	-	-
get(s[i+u,j+v])	get(SAD[u,v])	j=0	-	-
send(s[i+u,j+v])	get(s[i+u,j+v])	get(SAD[u,v])	-	-
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	send(s[i+u,j+v])	get(s[i+u,j+v])	-	-
j=1	SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	get(r[i,j])	-	-
get(s[i+u,j+v])	j=1	SAD[u,v]+=$\cdots$	-	-
send(s[i+u,j+v])	get(s[i+u,j+v])	j=1	-	-
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	send(s[i+u,j+v])	get(s[i+u,j+v])	-	-
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Fig. 12. A "source-level" representation of the code assigned to the processor i ($0 \leq i < n$) during the initialization loops. After $u > -p$, then `r[i,j]` can be loaded from the local cache. Also, because the next processor would require the data to be passed, the instruction `get(r[i,j])` is replaced by `send(s[i+u,j+v])`. When there is a mark like send(s[i+u,j+v]), it denotes a local bus transaction. Since the local buses are effectively used, there are only two external memory operations in each j loop in total.

PU_0	PU_1	PU_2	$\cdots$	PU_{n-1}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
v=-p+1	send(SAD[u,v])	$\vdots$	$\cdots$	$\cdots$
u=-p	v=-p+1	send(SAD[u,v])	$\cdots$	$\cdots$
j=0	u=-p	v=-p+1	$\cdots$	$\cdots$
SAD[u,v]=0	j=0	u=-p	$\cdots$	$\cdots$
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	get(SAD[u,v])	j=0	$\cdots$	$\cdots$
j=1	SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	get(SAD[u,v])	$\cdots$	$\cdots$
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	j=1	SAD[u,v]+=$\cdots$	$\cdots$	$\cdots$
j=2	SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	j=1	$\cdots$	$\cdots$
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	j=2	SAD[u,v]+=$\cdots$	$\cdots$	$\cdots$
j=3	SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	j=2	$\cdots$	$\cdots$
$\vdots$	$\vdots$	SAD[u,v]+=$\cdots$	$\vdots$	$\vdots$
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	j=n-3	$\vdots$	$\cdots$	$\cdots$
j=n-2	SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	j=n-3	$\cdots$	$\cdots$
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	j=n-2	SAD[u,v]+=$\cdots$	$\cdots$	$\cdots$
j=n-1	SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	j=n-2	$\cdots$	$\cdots$
get(s[i+u,j+v])	j=n-1	SAD[u,v]+=$\cdots$	$\cdots$	$\cdots$
send(s[i+u,j+v])	get(s[i+u,j+v])	j=n-1	$\cdots$	$\cdots$
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	send(s[i+u,j+v])	get(s[i+u,j+v])	$\cdots$	$\cdots$
send(SAD[u,v])	SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	send(s[i+u,j+v])	$\cdots$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Fig. 13. A "source-level" representation of the code assigned to the processor i ($0 \leq i < n$) after the cache is full.

PU_0	PU_1	PU_2	$\cdots$	PU_{n-1}
v=-p	send(SAD[u,v])	send(s[i+u,j+v]	$\cdots$	$\cdots$
u=-p+n	v=-p	send(SAD[u,v])	$\cdots$	$\cdots$
j=0	u=-p+n	v=-p	$\cdots$	$\cdots$
SAD[u,v]=0	j=0	u=-p+n	$\cdots$	$\cdots$
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	get(SAD[u,v])	j=0	$\cdots$	$\cdots$
j=1	SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	get(SAD[u,v])	$\cdots$	$\cdots$
SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	j=1	SAD[u,v]+=$\cdots$	$\cdots$	$\cdots$
j=2	SAD[u,v]+=abs(s[i+u,j+v]-r[i,j])	j=1	$\cdots$	$\cdots$
$\vdots$	$\vdots$	SAD[u,v]+=$\cdots$	$\vdots$	$\vdots$

Fig. 14. A "source-level" representation of the code assigned to the processor i ($0 \leq i < n$) after the cache is filled up. Note that after $v > -p$, there is no need for passing `s[i+u,j+v]` (except the very last one) because the `s[i+u,j+v]` is already in the cache. Because the cache is effectively used, there is only one external memory operation per u loop in total and there are only two local bus transactions per u loop per processor.

Table 3. Comparison between the operation placement and scheduling by the brute force method and our method (with frame size is 352×288, $p = 16$, $n = 16$, and 16 PUs). The parallelism is fully realized in the brute force method, and the number of operation cycles is minimized. However, the operation placement and scheduling can only work when a very high external memory bandwidth or a huge cache is provided. If the design does not exploit local communication and local caches, each pixel in the previous frame and the current frame will be read repeatedly $(2p)^2$ times. Hence, a extremely high external memory bandwidth is required. In order to capture the data reusability in the brute force design, each PU can use the a local cache to store $(2p)$ lines of the previous frame and n pixels of current block. Consequently, the cache size is (352 (pixels/line) $\times$ 32 (lines/PU) + 16 (pixels/PU)) $\times$ 16 PUs = 180 KB, which is larger than the current state-of-the-art on-chip cache. Because the design does not use the local communication, each PU requests a copy of the previous frame independently, i.e., a pixel is read 16 times. Although the access to external memory is much less than that of the design without caches, it is still a huge amount. In our designs, besides using a small compiler-directed caches [25, 37] to exploit the data reusability, the PUs use few cycles to exchange information via the local communication channels. Therefore, while using more cycles (1.7%), our designs use a small external memory bandwidth.

Operation placement & scheduling	External memory access	Local communication per channel	Total cache size	Operations per block
Brute force without cache	3.1 GB/s	0	0	16384
Brute force with cache	64 MB/s	0	180 KB	16384
Our design from multiprojecting the 4D DG of the BMA	43 MB/s	33 MB/s	8 KB	16639
Our design from multiprojecting the 5D DG of the BMA	24 MB/s	27 MB/s	180 KB	16639

Moreover, when the data dependence between different reference blocks are taken into account, the dimension of the DG of the BMA is more than 4. Table 3 also shows that multiprojecting the 5D DG of the BMA reduces the amount of the external memory access by an additional 1.35 times.

The proposed implementation of this computationally intensive and regular component of the BMA can achieve a speed-up ratio of 15.9, compared to a single processor implementation. After that, we concentrate on the second half of the BMA, determining the motion vector by the least SAD (cf. Eq. (10) and the pseudo code of the BMA of a single current block in Fig. 7). Since this part is control intensive but less computation-consuming (around 12.2 MOPS[5]), it is easily supported by the software solution running on a RISC core.

6. Implementation of True Motion Tracking Algorithm

True motion tracking (tracking of features in the video) has many useful multimedia applications, such as:

- Video data compression: efficient coding, rate optimized motion vector coding (MPEG-2), subjective picture quality (less block effects), object-based video coding (MPEG-4), object-based global motion compensation, and so on.
- Video spatio-temporal interpolation: frame-rate conversion applications, interlaced-to-progressive scan conversion, enhancement of motion pictures, synthesis, and so forth.
- Computer/machine vision: object motion estimation (including recovering the camera motion relative to the scene), video object segmentation (MPEG-4 and MPEG-7), 3D video object reconstruction (monoscopic or stereoscopic), image analysis for security, transportation, and medical purposes, etc.

This case study demonstrates how to implement a true motion tracking algorithm [29] on the proposed architectural platform, which has a 16-PU processing array.

Because the true motion field is piecewise continuous, the motion of a feature block is determined by consulting all its neighboring blocks' directions. (Conventionally, the minimum SAD of a block of pixels is used to find the motion vector of the block in BMAs.) This allows a chance that a singular and erroneous motion vector may be corrected by its surrounding motion vectors (just like median filtering). Since the neighboring blocks may not have uniform motion vectors, a neighborhood relaxation formulation is used to allow some local variations of motion vectors among neighboring blocks:

$$\mathrm{Score}(B_{x,y}, \vec{v}) = \mathrm{SAD}(B_{x,y}, \vec{v}) + \sum_{B_{k,l} \in \mathcal{N}(B_{x,y})} W(B_{k,l}, B_{x,y}) \min_{\vec{\delta}} \{\mathrm{SAD}(B_{k,l}, \vec{v} + \vec{\delta})\} \quad (13)$$

where $B_{x,y}$ means a block of pixels of which we would like to determine the motion, $\mathcal{N}(B_{x,y})$ is the set of neighboring blocks of $B_{x,y}$, $W(B_{k,l}, B_{x,y})$ is the weighting factor for different neighbors. A small $\vec{\delta}$ is incorporated to allow some local variations of motion vectors among neighboring blocks. The motion vector is obtained as

$$\textit{motion of } B_{x,y} = \arg\min_{\vec{v}} \{\mathrm{Score}(B_{x,y}, \vec{v})\}$$

6.1. Algorithmic Partitioning of the True Motion Tracking Formulation

Although the formulation seems complicated, it can be divided into four steps as shown below. After that, each of them are regular for the hardware or software implementation.

Step 1. We calculate the basic SADs as shown below:

$$\mathrm{SAD}[x, y, u, v] = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} | s[nx + i + u, ny + j + v] - r[nx + i, ny + j] | \quad (14)$$

where n is the block width and height, p is the absolute values of the maximum possible vertical and horizontal motion, indices x and y indicate the block position, $r[nx + i, ny + j]$ is the luminance value (pixel intensity) in the current block at coordinates (i, j), $s[nx + i + u, ny + j + v]$ is the luminance value in the search area in the previous frame, and (u, v) represents the candidate displacement vector.

For a frame with 352 pixels × 288 pixels, there are 22 blocks × 18 blocks because the block size is 16 × 16 ($n = 16$). That is, $0 \le x < 22$ and $0 \le y < 18$. As the search range p is 16, there are $32 \times 32 \times 16 \times 16 \times 2 = 524 \times 10^3$ additions for a block. For a P-frame, therefore, there are 208×10^6 additions, which must be finished within 1/30 of a second in a real-time application.

This step takes considerable computation and memory access. (In fact, it is the most computationally intensive part of the true motion tracker.) Fortunately, it is very regular for parallel and pipeline processing. Section 5 shows an efficient implementation.

Step 2. We calculate the minimum SADs after the $\vec{\delta}$-vibration:

$$\mathrm{mSAD}[x, y, u, v] = \min_{-1 \le \delta_u, \delta_v \le 1} \mathrm{SAD}[x, y, u + \delta_u, v + \delta_v] \quad (15)$$

where the vibration of the motion vector is limited within $\{-1,1\} \times \{-1,1\}$.

Each $\text{mSAD}[x,y,u,v]$ takes 9 operations (1 assignment and 8 comparisons) in Eq. (15). There are $32 \times 32 \times 22 \times 18$ such $\text{mSAD}[x,y,u,v]$ in a frame (within 1/30 seconds). Therefore, this step needs 109×10^6 operations per second.

Although this step takes less computation than the first step, a conventional programmable processor still has difficulty to supply such a high computation demand. In Section 6.2, we will demonstrate how to implement this step on our processing array.

This computation requires a huge memory bandwidth. The second step reads the SAD array 109×10^6 times per second. There are also 97×10^6 read-after-write operations per second over the partial minimum. Without a good memory flow, the system design could be impractical (very expensive to supprot a high memory bandwidth). Section 6.2 will address the memory flow design.

Step 3. We calculate the Scores.

$$\begin{aligned}\text{Score}[x,y,u,v] = \text{SAD}[x,y,u,v] + w\{ & \\ \text{mSAD}[x-1,y,u,v] + \text{mSAD}[x+1,y,u,v] + & \\ \text{mSAD}[x,y-1,u,v] + \text{mSAD}[x,y+1,u,v]\}& \end{aligned} \quad (16)$$

where the neighborhood is the nearest four neighboring blocks. For simplicity, we make the $W(\cdot)$ depend on the distance between the central block and the neighboring block only [29]. Because these four neighbors are equi-distance from the central block, their weightings equal a constant w.

Each $\text{Score}[x,y,u,v]$ takes 5 operations in Eq. (16). There are $32 \times 32 \times 22 \times 18$ such $\text{Score}[x,y,u,v]$ in a frame (with 1/30 seconds). Therefore, this step needs 61×10^6 operations per second. Section 6.3 will demonstrate how to implement this step on our processing array.

Step 4. We determine the motion vector by the least Score (cf. Eq. (10)):

$$\textit{motion of } B_{x,y} = \arg\min_{[u,v]}\{\text{Score}(x,y,u,v)\} \quad (17)$$

It takes 32×32 comparisons for each block. There are 406×10^3 comparisons per frame (1/30 seconds). Hence, this part is less computation-consuming (around 12 MOPS); it is easily supported by the software solution running on a RISC core.

6.2. Implementation of Calculating the mSAD

The 2D DG of calculating the mSAD. As shown in Eq. (15), there are 6 independent indices $(x,y,u,v,\delta_u,\delta_v)$. Therefore, the DG of calculating the mSAD could be six-dimensional. However, there is no data dependence between different x and y. Therefore, the DG of calculating the mSAD is four-dimensional (u,v,δ_u,δ_v). In addition, because there is a high operation reusability in the Eq. (15), this task can be further divided into two sub-steps:

$$\begin{aligned}\text{pSAD}[x,y,u,v] &= \min_{\delta_u}\{\text{SAD}[x,y,u-\delta_u,v]\} \\ \text{mSAD}[x,y,u,v] &= \min_{\delta_v}\{\text{pSAD}[x,y,u,v-\delta_v]\}\end{aligned}$$

The DGs of the sub-steps are the same and two-dimensional. Figure 15 shows the DG, which is embedded in the 2D (u,δ_u) index space. Note that $-p \le u < p$ and $-1 \le \delta_u \le 1$.

There are two data-dependence edges in this DG. We use $\vec{E}_1$ to denote the read-after-read data dependence of the $\text{SAD}[x,y,u-\delta_u,v]$. The $\text{SAD}[x,y,u-\delta_u,v]$ will be used repeatedly for (1) different u, (2) same $u-\delta_u$. Therefore, one possible choice of the $\vec{E}_1$ is $[1,1]^T$. We use $\vec{E}_2$ to denote the read-after-write data dependence of the partial minimum. One possible choice is $[0,1]^T$. The algebraic representation of the 2D DG is shown below:

SAD ($\vec{E}_1$)	Partial min ($\vec{E}_2$)
$[1,1]^T$	$[0,1]^T$

Transform the 2D DG to a 3D DG. The size of the 2D DG is 32×3 (assuming $p = 16$). The target architecture is a linear processing array of 16 PUs. Therefore, it is necessary to partition the DG/SFG and execute the SFG in a parallel and pipeline manner. After careful evaluation, we decide to apply the locally sequential globally parallel (LSGP) scheme in this implementation [8].

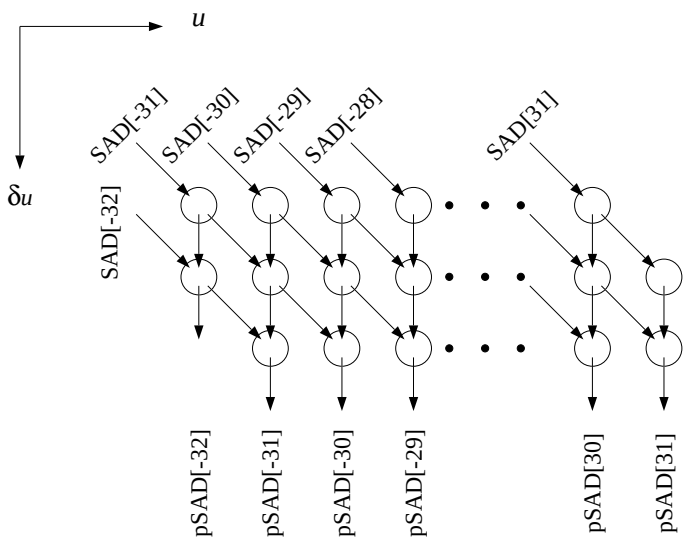

Fig. 15. The 2D DG of the second step of the true motion tracker. There are two data-dependence edges in this DG. $\vec{E}_1$ denotes the read-after-read data dependence of the SAD$[x, y, u + \delta_u, v]$. The SAD$[x, y, u + \delta_u, v]$ will be used repeatedly for (1) different u, δ_u, and (2) same $u + \delta_u$. Therefore, one possible choice of $\vec{E}_1$ is $[1, 1]^T$. $\vec{E}_2$, $[0, 1]^T$, denotes the partial minimum data dependence.

The first step in applying the LSGP is to transform the 2D DG to a 3D DG whose size is 2×16×3. Two new indices a and b are introduced. One unit of the u is one unit of the a when the dependence edge does not move across different packing segments. (A packing segment consists of all the computation nodes within two units of sequential u. That is, the packing boundary is when 2 divides u.) One unit of the u is 1 unit of the b and -1 unit of the a when the dependence edge crosses the packing boundary of the transformed DG one time. It is obvious that $u = a + 2b$. The 3D DG is shown below:

$\vec{d}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$	$\vec{s}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$	$P_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$
$\vec{d}_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$	$\vec{s}_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$	$P_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}$

To ensure processor availability, $M_3 = 2$. Therefore, we have the allocation matrix and scheduling vector as

$$\begin{aligned} \mathbf{A} &= \mathbf{P_2 P_3} = [0, 1, 0] \\ \mathbf{S}^T &= \vec{s}_2^T \mathbf{P_3} + M_3 \vec{s}_3^T \\ &= [1, 0, 2] \end{aligned}$$

and the 1D SFG as

SAD ($\vec{E}_1$)	Partial min ($\vec{E}_2$)
$[1, 0, 1]^T$	$[0, 0, 1]^T$
$[-1, 1, 1]^T$	

SAD ($\vec{E}_1$)		Partial min ($\vec{E}_2$)	
0	($D_1 = 3$)	0	($D_1 = 2$)
1	($D_1 = 1$)		

Multiprojecting the 3D DG into a 1D SFG. We multiproject the 3D DG into a 1D SFG using the following:

Using an extremely small buffer (3 Bytes) with the help of local communication, the SAD can be

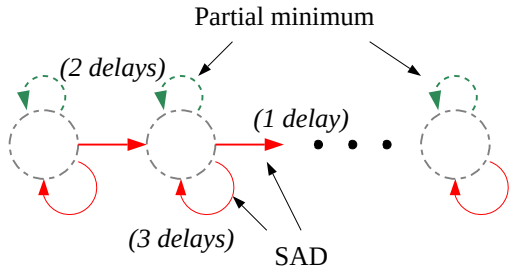

Fig. 16. The 1D SFG of the second step of the true motion tracker (from multiprojecting 3D DG).

used repeatedly without any extra external memory access. It is obvious that the partial minimum can be used in the same way.

Evaluation of the Implementation. The allocation matrix and the scheduling vector give us not only the SFG (cf. Fig. 16), but also some important features of the implementation:

1. **Execution cycles:** The computational time of a block is equal to the difference between the time of the first operation and the time of the last operation, i.e.,

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

where $\underline{c}_x$ and $\underline{c}_y$ are two computation nodes in the DG.
In this particular implementation, we have $-16 \le u \le 15$, $u = a + 2b$, and $-1 \le \delta_u \le 1$. Hence, we have

$$\begin{aligned} 0 \le\ & a & \le 1 \\ -8 \le\ & b & \le 7 \\ -1 \le\ & \delta_u & \le 1 \end{aligned}$$

In addition, we do not perform any useful computation if $u + \delta_u < -16$ or $u + \delta_u > 15$. It can be easily shown that this implementation takes 6 cycles by a simple integer linear programming.
Note that there are $32 \times 3 = 96$ computation nodes in the DG. If the parallelism are fully realized on the 16 processors, then the shortest execution time should be $96/16 = 6$ cycles. That is, our implementation achieves the highest efficiency in terms of computational time.
The computational time is 6 (cycles/line) $\times$ 32 (lines/block) = 196 cycles/block for the first sub-step. The computational time is also 196 cycles/block for the second sub-step. The total time is 384 (cycles/block) $\times$ 22 (blocks/slice) $\times$ 18 (slices/frame) $= 152 \times 10^3$ cycles/frame. This step adds 4.6 MOPS for each PU.
2. **Memory size:** Because we must store the SAD for 3 cycles and the partial minimum for 2 cycles, the total amount of memory size is 5 bytes.
3. **Internal communication per channel:** Two PUs exchange 3 bytes using the local bus per 6 clock cycles. The internal communication is 76 KB per frame (1/30 seconds). Therefore, this step adds 2.3 MB to the total internal communication per second.
4. **External memory access:** There are 32-byte memory reads and 32-byte memory writes in each sub-step for a fixed u (or a fixed v). There are 64 (operations/line) $\times$ 32 (lines/substep) $\times$ 2 (substeps/block) = 4096 external memory operations/block. Hence, this step adds 4096 (Bytes/block) $\times$ 22 (blocks/slice) $\times$ 18 (slices/frame) $\times$ 30 (frames/sec) = 48 MB/sec external memory access to the requirement of the external communication bandwidth.
As we mentioned before, without reusing the data, this step will take 206 MB/sec of external memory access. Because our design has special data flow from this operation placement and scheduling, it needs only 24% of that global communication bandwidth.

6.3. Implementation of Calculating the Score

The 4D DG of calculating the Score. Eq. (16) can be written as the following:

$$\begin{aligned} \text{Score}[x, y, u, v] = \text{SAD}[x, y, u, v] + w \sum_{\delta_x=0}^{1} \sum_{\delta_y=0}^{1} \\ \text{mSAD}[x - \delta_x - \delta_y + 1, y - \delta_x + \delta_y, u, v] \end{aligned} \quad (18)$$

Although there are 6 independent indices $(x, y, \delta_x, \delta_y, u, v)$, there is no data dependence between different u and v. Assuming that $\text{Score}[x, y, u, v] = \text{SAD}[x, y, u, v]$ initially, the DG of calculating the Score is four-dimensional $(x, y, \delta_x, \delta_y)$. Figure 17 shows the core of the

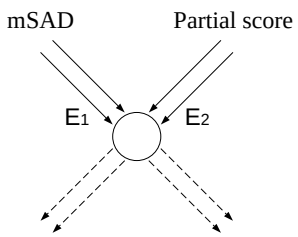

Fig. 17. The 4D DG of the third step of the true motion tracker. There are two data-dependence edges in this DG. $\vec{E}_1$ denotes the read-after-read data dependence of the mSAD$[x - \delta_x - \delta_y + 1, y - \delta_x + \delta_y, u, v]$. The mSAD$[x - \delta_x - \delta_y + 1, y - \delta_x + \delta_y, u, v]$ will be used repeatedly for (1) different x, y, (2) same $x - \delta_x - \delta_y + 1$, and (3) same $y - \delta_x + \delta_y$. $\vec{E}_1$ is a two-dimensional reformable mesh. One possible choice is $[1, 1, 1, 0]^T$ and $[1, -1, 0, 1]^T$. $\vec{E}_2$ denotes the read-after-write data dependence of the partial score. It is a two-dimensional mesh as well. One possible choice is $[0, 0, 1, 0]^T$ and $[0, 0, 0, 1]^T$.

DG, which is embedded in the 4D $(x, y, \delta_x, \delta_y)$ index space. Note that $0 \leq x < 22$, $0 \leq y < 18$, and $0 \leq \delta_x, \delta_y \leq 1$ (assuming the picture size is 352×288 and block size is 16×16).

There are two data-dependence edges in this DG. We use $\vec{E}_1$ to denote the read-after-read data dependence of the mSAD$[x - \delta_x - \delta_y + 1, y - \delta_x + \delta_y, u, v]$. The mSAD$[x - \delta_x - \delta_y + 1, y - \delta_x + \delta_y, u, v]$ will be used repeatedly for (1) different x, y, (2) same $x - \delta_x - \delta_y + 1$, and (3) same $y - \delta_x + \delta_y$. Therefore, $\vec{E}_1$ is a two-dimensional reformable mesh. One possible choice is $[1, 1, 1, 0]^T$ and $[1, -1, 0, 1]^T$. We use $\vec{E}_2$ to denote the read-after-write data dependence of the partial score. It is also a two-dimensional mesh. One possible choice is $[0, 0, 1, 0]^T$ and $[0, 0, 0, 1]^T$. The algebraic representation of the 4D DG is shown below:

SAD ($\vec{E}_1$)	Partial sum ($\vec{E}_2$)
$[1, 1, 1, 0]^T$	$[0, 0, 1, 0]^T$
$[1, -1, 0, 1]^T$	$[0, 0, 0, 1]^T$

Transform the 4D DG to a 5D DG. The size of the 4D DG is $22 \times 18 \times 2 \times 2$. The target architecture is a linear processing array of 16 PUs. Therefore, it is necessary to partition the DG/SFG and execute the SFG in a parallel and pipeline manner. We decide to implement this step on 11 PUs using the LSGP scheme (cf. Section 6.2).

The first step in applying the LSGP is to transform the 4D DG to a 5D DG whose size is $2 \times 11 \times 18 \times 2 \times 2$ by introducing two new indices a and b. One unit of the x is one unit of the a when the dependence edge does not move across different packing segments. (A packing segment consists of all the computation nodes within two units of sequential x. That is, the packing boundary is when 2 divides x.) One unit of the x is 1 unit of the b and -1 unit of the a when the dependence edge crosses the packing boundary of the transformed DG one time. Note that $x = a + 2b$. The 5D DG is shown below:

SAD ($\vec{E}_1$)	Partial sum ($\vec{E}_2$)
$[1, 0, 1, 1, 0]^T$	$[0, 0, 0, 1, 0]^T$
$[-1, 1, 1, 1, 0]^T$	$[0, 0, 0, 0, 1]^T$
$[1, 0, -1, 0, 1]^T$	
$[-1, 1, -1, 0, 1]^T$	

Multiprojecting the 5D DG into a 1D SFG. We multiproject the 5D DG into a 1D SFG using the following:

$$\vec{d}_5 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \vec{s}_5 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad P_5 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\vec{d}_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \vec{s}_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad P_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\vec{d}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad \vec{s}_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad P_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\vec{d}_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \vec{s}_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad P_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

To ensure processor availability, $M_5 = 2$, $M_4 = 2$, and $M_3 = 2$. The resulted allocation matrix

and scheduling vector will be:

$$\begin{aligned}\mathbf{A} &= \mathbf{P_2P_3P_4P_5} = [0,1,0,0,0]\\ \mathbf{S}^T &= \vec{s}_2^T\mathbf{P_3P_4P_5} + M_3\vec{s}_3^T\mathbf{P_4P_5}\\ &\quad + M_3M_4\vec{s}_4^T\mathbf{P_5} + M_3M_4M_5\vec{s}_5^T\\ &= [1,0,8,4,2]\end{aligned}$$

Therefore, we have

SAD ($\vec{E}_1$)		Partial sum ($\vec{E}_2$)	
0	($D_1 = 13$)	0	($D_1 = 4$)
1	($D_1 = 11$)	0	($D_1 = 2$)
0	($D_1 = -5$)		
1	($D_1 = -7$)		

Because $\vec{E}_2$ is a 2D summation mesh, we apply the summation rule to it (cf. Table 2 and [8]) so that all the delays of the $\vec{E}_2$ edges are equal to 2.

Because two of the $\vec{E}_1$ edges contain negative delays, we apply the redirection rule to them so as to have positive delays. Furthermore, because $\vec{E}_1$ is 2D reformable read-after-read data dependence, we apply the reformation rule to it (cf. Table 2 and [8]). We let $[-1,1,1,1,0]^T$ become $[-1,1,1,1,0]^T + [1,0,-1,0,1]^T = [0,1,0,1,1]^T$ so that the delay of the $\vec{E}_1$ edge becomes 6. The final SFG becomes the following:

SAD ($\vec{E}_1$)		Partial sum ($\vec{E}_2$)	
0	($D_1 = 13$)	0	($D_1 = 2$)
1	($D_1 = 6$)	0	($D_1 = 2$)
0	($D_1 = 5$)		
−1	($D_1 = 7$)		

Evaluation of the Implementation. The final SFG (cf. Fig. 18) give us some important features of the implementation:

1. **Execution cycles:** By a simple integer linear programming, the computational time of fixed u and v is equal to

$$T = \max_{\underline{c}_x, \underline{c}_y} \{\mathbf{S}^T(\underline{c}_x - \underline{c}_y)\} + 1 = 144$$

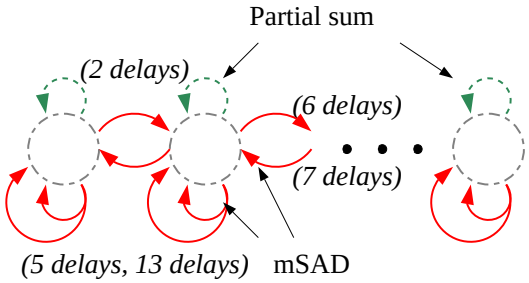

Fig. 18. The 1D SFG of the third step of the true motion tracker (from multiprojecting 3D DG).

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

Note that there are $22 \times 18 \times 4 = 1584$ computation nodes in the DG. If the parallelism are fully realized in the 16 PUs, then the shortest execution time should be $1584/16 = 99$ cycles. That is, our implementation is close to (but not is equal to) the lowest bound of the computational time because only 11 PUs are used in this design.

Since $-16 \le u, v \le 15$, the total number of cycles for a frame of picture is $144 \times 32 \times 32 = 147 \times 10^3$ cycles. This step adds 4.4 MOPS for each PU.

2. **Memory size:** Because we only need to store the mSAD for $13 + 5 = 18$ cycles and partial minimum for 2 cycles, the total amount of memory size is 20 bytes.
3. **Internal communication per channel:** Two PUs exchange 1 byte information using the local bus per 4 cycles. The internal communication is 37 KB per frame (1/30 seconds). Therefore, this step adds 1.1 MB internal communication to the total internal communication.
4. **External memory access:** Because the local data memory and the local bus are exploited for data reusability, each of the mSADs and the SADs will be read only once. There are $32 \times 32 \times 22 \times 18 \times 3 = 1.22$ MB external memory operations per frame (including the write-back of the Score). Hence, this step adds 36 MB to the total external communication.

   Without reusing the data, this step will read each of the mSADs four times in addition to read each SAD once and write each Score once. That is to say, there will be 73 MB/sec

Table 4. Implementation of the true motion tracking algorithm on the proposed architectural platform using our operation placement and scheduling scheme (with frame size is 352×288, $p = 16$, $n = 16$, and 16 PUs). The parallelism is almost fully realized. One of the most prominent features is that the design uses a small external memory bandwidth by exploiting small caches.

Step	Implementation on	MOPS per processor	Cache size	Internal Communication	External Communication
1 (Eq. (14))	Processing array	198	8 KB	33 MB/sec	43 MB/sec
2 (Eq. (15))	Processing array	5	80 B	2 MB/sec	48 MB/sec
3 (Eq. (16))	Processing array	4	320 B	1 MB/sec	36 MB/sec
4 (Eq. (17))	RISC core	12	-	-	12 MB/sec

of external memory access. Our design, with the new operation placement and scheduling, needs only 50% of that global communication bandwidth.

6.4. Summary of the Case Study

Table 4 shows a brief summary of the implementation of the true motion tracking algorithm. This case study demonstrates three important issues:

1. Architectural support for high performance parallel processing, low external and global communication, and flexibility
2. Compiler support between high performance parallel processing, low requirement of the memory bandwidth, and scalability
3. Algorithmic partitioning for higher coprocessor performance

We use the systolic-type compiler methodology to achieve high parallelism. Unlike conventional systolic arrays, our design is not targeted at a fixed functionality.

We program the processing array like a reconfigurable computing engine—e.g., FPGA. For different (sub-)tasks, the operations of the PUs and the memory data flow are different. However, unlike a conventional FPGA, the granularity of the reconfigurable part is higher (8-bit or 16-bit).

Conventionally, desirably high parallelism can be achieved only when a correspondingly high memory bandwidth is supported. Using our placement and scheduling scheme on the proposed architectural platform, such high parallelism can be achieved without a large cache or memory bandwidth overhead.

7. Conclusions

The rapid progress in VLSI technology will soon reach more than 100 million transistors in a chip, implying tremendous amount of computing power for many multimedia applications. The silicon area required for implementing a specific function will decrease considerably, and a higher functionality can be realized on a single chip, for example, single-chip MPEG-2 encoders from NEC Corporation or Philips Electronics [30]. This trend leads to the monolithic integration of programmable processor cores, function-specific modules, and various system interfaces in order to enable high multimedia functionality at decreased system design costs.

Because of the interaction of various design parameters comprising algorithm and architecture issues, the multimedia system design may be best accomplished via a *codesign of algorithm/architecture*: algorithms should be partitioned to facilitate dedicated processing modules; and architectures should be tailored to achieve higher efficiency for the special application domain.

In this work, we find that the future multimedia signal processing implementation hinges upon an optimal tradeoff between the two design spaces—*internal* (the core to be used for the software solution) and *external* (the accelerator to be used as the hardware solution), as shown in Fig. 5. Such a design approach naturally leads to a *coprocessor* architecture, as shown in Fig. 4, and a systematic operation placement and scheduling scheme as shown in Section 4.

In the codesign process, the degree of flexibility demanded by the less predictable algorithm components competes with the high efficiency of systematically derived dedicated approaches for regular, fine-granularity components. *The optimum partition has to be determined individually*

for the targeted application domain. The best architectural mix depends on the characteristics of the algorithms to be implemented and on the targeted application spectrum. For example, Philips Electronics presents a single-chip MPEG-2 video encoder, I.McIC, in [31]. Because its target application is digital recording at the latest consumer storage media, such as D-VHS and DVD, it allows higher bit-rate than broadcasting and other storage media. Therefore, the MPEG-2 ML@SP mode is chosen to be the main operation mode, which leads to this cost-effective single-chip solution easily.

8. Future Work—Algorithm, Architecture, and Programming Language Codesign

Today, most of the multimedia algorithms have great potential in parallelism. However, after being described in *C* or *C++*, they are less efficient implemented in the programmable hardware. One of the great differences is the following: conventional high-level language compilers translate an instruction into several low-level machine codes while new compilers must translate several instructions into one machine code in order to full utilize the parallelism supported by the hardware.

Besides the algorithm and architecture codesign, an important part of a sound solution for designing multimedia system should be new programming languages that can provide efficient description of multimedia algorithms and can be supported efficiently by the hardware. In the Electronic Eye project at Siemens AG, for example, a programming tool—Vision Programming Language (VPL)—is especially designed for the Vision Instruction Processor (VIP). The VPL language provides the formulation of algorithms for image processing in a quasi object-oriented notation and supports their implementation for the VIP's special architecture [32].

In the future, besides the interaction of various design parameters comprising algorithm and architecture issues, we should bear the characteristics of the programming languages in mind when we design or choose the algorithm/architecture [33, 34, 35], and vice versa.

Acknowledgements

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

Notes

1. BOPS: billion operations per second.
2. MIPS: million instructions per second.
3. A DG is a directed graph, $G =< V, E >$, which shows the dependence of the operations that occur in an algorithm. Each operation will be represented as one node, V, in the graph. The dependence relation among the operations will be shows as an arc, E, between the corresponding operations.
4. 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, $G =< V, E, D(E) >$ as the SFG expression consists of processing nodes, communicating edges, and delays. In general, a node, V, is representing an arithmetic or logic function performed with zero delay, such as, multiplication or addition. The directed edges E model the interconnections between the nodes. Each edge e of E connects an output port of a node to an input port of some node and is weighted with a delay count $D(e)$. The delay count is the number of delays along the connection. Often, input and output ports are refereed to as sources and sinks, respectively.
5. MOPS: million operations per second.

References

1. C.-L. Chen, B.-S. Liang, and C.-W. Jen, "Low-Cost Raster Engine for Video Game, Multimedia PC and Interactive TV," *IEEE Trans. on Consumer Electronics*, Vol. 41, No. 3, pp. 724–730, Aug. 1995.
2. S.-H. Lin, S. Y. Kung, and L. J. Lin, "Face Recognition/Detection by Probabilistic Decision-based Neural Networks," *IEEE Trans. on Neural Networks*, Vol. 8, No. 1, pp. 114–132, Jan. 1997.
3. T. Nishitani, P. H. Ang, and F. Catthoor, eds., *VLSI Video/Image Signal Processing*, (Norwell, MA), Kluwer Academic Publishers, 1993.
4. 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.
5. V. Bhaskaran, K. Konstantinides, R. B. Lee, and J. P. Beck, "Algorithmical and Architectural Enhancements for Real-Time MPEG-1 Decoding on a General Purpose RISC Workstation," *IEEE Trans. Circuits and Systems for Video Technology*, Vol. 5, No. 5, pp. 380–386, Oct. 1995.

6. S. Y. Kung and Y.-K. Chen, "On Architectural Styles for Multimedia Signal Processors," in *Proc. of IEEE Workshop on Multimedia Signal Processing*, pp. 427–432, June 1997.
7. P. Pirsch, H.-J. Stolberg, Y.-K. Chen, and S. Y. Kung, "Implementation of Media Processors," *IEEE Signal Processing Magazine*, Vol. 14, No. 4, pp. 48–51, July 1997.
8. Y.-K. Chen and S. Y. Kung, "A Systolic Design Design Methodology with Application to Full-Search Block-Matching Architectures," *Journal of VLSI Signal Processing Systems*, Vol. 19, No. 1, pp. 51–77, 1998.
9. 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.
10. Chromatic Research, "Mpact 2 Media Processor Data Sheet." `http://www.mpact.com/tech/mpact2.pdf`, Feb. 1998.
11. R. B. Lee, "Subword Parallelism with MAX-2," *IEEE Micro*, Vol. 16, No. 4, pp. 51–59, Aug. 1996.
12. M. O'Connor, "Extending Instructions for Multimedia," *Electronic Engineering Times*, No. 874, p. 82, Nov. 1995.
13. Intel, "Intel MMX Technology–Developer's Guide." `http://developer.intel.com/drg/mmx/manuals/dg/devguide.htm`, 1997.
14. K. Gaedke, H. Jeschke, and P. Pirsch, "A VLSI Based MIMD Architecture of a Multiprocessor System for Real-Time Video Processing Applications," *Journal of VLSI Signal Processing*, Vol. 5, No. 2-3, pp. 159–169, April 1993.
15. Texas Instruments, "TMS320C8x Product Information." `http://www.ti.com/sc/docs/dsps/products/c8x/`, 1998.
16. S. Dutta, A. Wolfe, W. Wolf, and K. J. O'Connor, "Design Issues for Very-Long-Instruction-Word VLSI," in *VLSI Signal Processing* (W. Burleson, K. Konstantinides, and T. Meng, eds.), Vol. IX, pp. 95–104, 1996.
17. Texas Instruments, "TMS320C6000 Product Information." `http://www.ti.com/sc/docs/dsps/products/c6000/`, 1998.
18. K. Nadehara, I. Kuroda, M. Daito, and T. Nakayama, "Low-Power Multimedia RISC," *IEEE Micro*, Vol. 15, No. 6, pp. 20–29, Dec. 1995.
19. K. Suzuki, T. Arai, K. Nadehara, and I. Kuroda, "V830R/AV: Embedded Multimedia Superscalar RISC Processor," *IEEE Micro*, Vol. 18, No. 2, pp. 35–47, March/April 1998.
20. D. Matzke, "Will Physical Scalability Sabotage Performance Gains?," *IEEE Computer*, Vol. 30, No. 9, pp. 37–39, Sept. 1997.
21. P. E. R. Lippens, V. Nagasamy, and W. Wolf, "CAD Challenges in Multimedia Computing," in *Proc. of Int'l Conf. on Computer-Aided Design*, pp. 502–508, 1995.
22. J. L. van Meerbergen, P. E. R. Lippens, B. McSweeney, W. F. J. Verhaegh, A. van der Werf, and A. van Zanten, "Architectural Strategies for High-Throughout Applications," *Journal of VLSI Signal Processing*, Vol. 5, No. 2-3, pp. 201–220, April 1993.
23. J. L. van Meerbergen, P. E. R. Lippens, W. F. J. Verhaegh, and A. van der Werf, "PHIDEO: High-Level Synthesis for High Throughput Applications," *Journal of VLSI Signal Processing*, Vol. 9, No. 1-2, pp. 89–104, Jan. 1995.
24. H. Yamauchi, Y. Tashiro, T. Minami, and Y. Suzuki, "Architecture and Implementation of a Highly Parallel Single-chip Video DSP," *IEEE Trans. on Circuits and Systems for Video Tech.*, Vol. 2, No. 2, pp. 207–220, June 1992.
25. L. Chol, H.-B. Lim, and P.-C. Yew, "Techniques for Compiler-Directed Cache Coherence," *IEEE Parallel and Distributed Technology*, Vol. 4, No. 4, pp. 23–34, Winter 1996.
26. S. Y. Kung, *VLSI Array Processors*, Prentice Hall, Englewood Cliffs, NJ, 1988.
27. M. J. Wirthlin and B. L. Hutchings, "A Dynamic Instruction Set Computer," in *Proc. of IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 99–107, April 1995.
28. 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.
29. Y.-K. Chen, Y.-T. Lin, and S. Y. Kung, "A Feature Tracking Algorithm Using Neighborhood Relaxation with Multi-Candidate Pre-Screening," in *Proc. of Int'l Conf. on Image Processing*, vol. II, pp. 513–516, Sept. 1996.
30. *Proc. of Int'l Solid-State Circuits Conference*, Feb. 1997.
31. R. P. Kleihorst, A. van der Werf, W. H. A. Brüls, W. F. J. Verhaegh, and E. Waterlander, "MPEG2 Video Encoding in Consumer Electronics," *Journal of VLSI Signal Processing Systems*, Vol. 17, No. 2/3, pp. 241–253, Nov. 1997.
32. F. Montrone, C. Niedermeier, and M. Richter, *VIP Vision Programming Language Manual*. Siemens AG, April 1997.
33. J. R. Hayes, M. E. Fraeman, R. L. Williams, and T. Zaremba, "An Architecture for the Direct Execution of the Forth Programming Language," in *Proc. of Second Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pp. 42–49, Oct. 1987.
34. L. Lopriore, "A Data Cache for Prolog Architectures," *Future Generation Computer Systems*, Vol. 9, No. 3, pp. 219–234, Sept. 1993.
35. E. Tick, *Memory Performance of Prolog Architectures*. Norwell, MA: Kluwer Academic Publishers, 1988.
36. Philips Electronics, "TRIMEDIA TM1000 Programmable Media Processor." `http://www-us2.semiconductors.philips.com/trimedia/products/tm1.stm`, 1997.
37. A. R. Hurson, K. M. Kavi, B. Shirazi, and B. Lee, "Cache Memories for Data Systems," *IEEE Parallel and Distributed Technology*, Vol. 4, No. 4, pp. 50–64, Winter 1996.