設 計 自 動 化 81－2
(1996. 10. 17)

プログラム可能なMSPA(Memory Sharing Processor Array)の設計法

李 冬菊　　國枝 博昭
東京工業大学
電気電子工学科
〒152 東京都目黒区大岡山 2-12-1
TEL: +81-3-5734-2574
FAX: +81-3-5734-2842
e-mail: dongju@ss.titech.ac.jp
e-mail: kunieda@ss.titech.ac.jp

あらまし　　筆者らは、規則構造のアルゴリズムを高速に実行するMSPA(Memory Sharing Processor Array)アーキテクチャを提案した。そこではシストリックアレイと異り、任意の数のプロセッサ要素を選択でき高い並列効率が達成できる点や設計過程が極めて簡単である点などのが特徴がある。今回はこれらの特徴を利用して、設計過程もハードウエアとして埋め込むことにより、応用範囲の広いプログラム可能なMSPAの設計法について述べる。得られたMSPAは、アルゴリズムサイズや先行制約関係などを入力パラメータに、それに適応したスケジュールやPE割り当てのためのパラメータを選択して、データパスやPE間接続などを変更する方式を採用しており、高速性を損なうことなく、プログラム性を実現している。

キーワード　　アレープロセッサ、並列処理、メモリアドレス生成、再構成、プログラム可能プロセッサ

Programmable Design for Memory Sharing Processor Array (MSPA)

Dongju Li and Hiroaki Kunieda
Dept. of EE Eng.
Tokyo Institute of Technology
Ookayama, Meguro-ku, Tokyo 152
TEL: +81-3-5734-2574
FAX: +81-3-5734-2842
e-mail: dongju@ss.titech.ac.jp
e-mail: kunieda@ss.titech.ac.jp

Abstract　　Memory sharing processor array (MSPA) architecture has been proposed with advantages of high efficient parallel processing, less data storage requirement, and high cost performance. MSPA design methodology has been developed with regularity structure and systematic procedure. In this paper, programmable MSPA is proposed to embed not only MSPA architecture, but design procedure into silicon chip so that various applications can be performed with excellently high speed. MSPA controller schedules operations corresponding to algorithm size, allocates resources altomaticaly, manipulates data flow among I/O ports, memory and processing elements (PEs). The introduction of this controller lead to a possibility to realize programmable and reconfigurable MSPA chip design.

Key words　　parallel processing, interconnection, memory address generation, array processor architecture.

1 Introduction

With the development of VLSI, large amount of high speed ASIC DSP are proposed in recent years. SIMD, MIMD, massively parallel, multiprocessor, array processor and so on are commonly adopted methodologies in order to realize high speed performance. [1] [2] [3] Memory sharing processor array (MSPA) is proposed as a new architecture to meet the high speed requirement. It works with successive executions without instruction fetch within the whole procedure of a job. [4][5]

In this paper, we extend the feasibility of MSPA to practical DSP application. The MSPA is used as a special work-mode in the developing DSP. Based on the regular structure and systematic design procedure of MSPA design methodology, it is possible to realize programmable and reconfigurable MSPA. A controller is introduced in order to fix a part of design procedure by hardware. It will be used to adjust operation scheduling, allocate resource automatically, manipulate data transfer between I/O interface and processing elements (PEs), convert data transfer rate, set reconfiguration parameters and control the reconfiguration of interconnection between PEs and memory address generation. This controller will lead to a programmable MSPA architecture.

As a co-processor, the programmable MSPA is able to work at a very high speed with successively operation cycles until it finishes a job. The proposed controller is extended to accelerate the operation scheduling corresponding to applications during the procedure of a job processing, which provides a feasible character to deal with semi-regular algorithms, such as DCT, DFT, FFT, etc.

In our DSP IC design, the programmable MSPA results in a possibility to transfer data in several kinds of schemes, not only adjacent local transfer such as systolic array, but also schedule-oriented reconfigurable path transfer. Therefore, the MSPA architecture gains another advantage of extending the range of data transfer. The controller leads to a programmable and reconfigurable MSPA which is more feasible to practical application. As an example, IC design result of the programmable and reconfigurable MSPA will be given to show its application in DSP IC. [4] [4] [4] [4] [4] [4] [4]

2 MSPA Principle

2.1 MSPA Architecture

Fig.1 describes the MSPA architecture. It consists of a processor array with direct links and buses associated with memory units (MU) or data format converter (DFC) and I/O ports. There is only one I/O port for the input of each variable. The data of a certain bandwidth comes in a word-serial scheme or a bit-serial scheme and is loaded directly to PEs or memory/DFC. The memory units store and load input data to processor array according to the control signal generated by the address generation unit (AGU).

The type of the processor array is restricted to the loosely coupled linear or 2D processor array, so that the

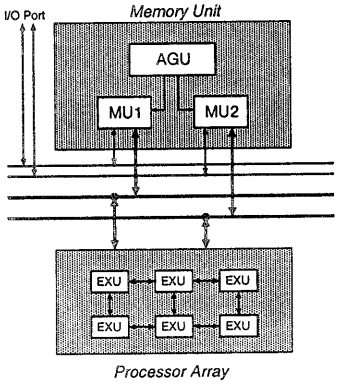

Fig. 1. MSPA Architecture

data transfers between processors are accomplished by local direct communications. Direct bus communications or bus communications via memory/DFC units are also available in this MSPA architecture.

For our hardware model, we assume that the processor cells operate synchronously at discrete time steps, and each processor cell completes its assigned operation within 1 step. Furthermore, we assume that it takes one step for both the data transfer between processor cells and memory unit/DFC through buses.

Tab.I shows the $U \times U$ matrix multiplier results of different exu, the number of PEs. It also includes the result of GPM systolic approach[6] for comparison. In Tab.I, the execution time T_{exe}, the number of PEs (exu), and the cost-performance of PEs ($T_{exe} \cdot exu$—the product of the two items above). are shown. Though the results of different algorithm size U are optimized by the number of PEs exu, any number of PEs can be selected in our approach.

Tab.II shows the design results for various applications. The first three applications perform the word-level operations[4], while the last two applications perform the bit-level operations[7]. Only the matrix-matrix multiplier belongs to a 3D algorithm, while the others belong to 2D algorithms. The functions of each processing element is also described. For our performance evaluation, we calculate the number of gates of PEs, memory units and DFC or control logic circuits as area description.

3 MSPA Design Flow

Firstly, we describe about design flow of ASIC MSPA. In order to keep the regularity of the algorithm and increase the resource utilization rate, we deal with the mapping problem in MSPA design by a modified linear transformation of the index set with an assumption that the number of processor elements is less than algorithm size. Fig.2 shows the ASIC design procedure on MSPA architecture.

TABLE I PE-TIME COMPARISON WITH GPM

U	GPM Systolic Array			Proposed MSPA		
	T_{exe}	exu	$T_{exu} \cdot exu$	T_{exe}	exu	$T_{exu} \cdot exu$
4	19	10	190	34	2	68
5	29	13	377	49	3	147
10	109	28	3052	208	5	1040
17	305	49	14925	577	9	5193
53	1717	261	448137	8425	18	151650
100	5644	496	2799424	20098	50	1004900
201	15001	1401	21016401	80801	101	8160901

TABLE II DESIGN RESULTS FOR VARIOUS APPLICATIONS

Application	Condition			T_{exe}	Area (Gates)		
	U	PE Func.	exu		$PE \times exu$	Mem.	DFC/Ctl
Matrix-vector	18	16 bits mult-add	6	69	2000 × 6	20736	2200
Matrix-matrix	16	16 bits mult-add	8	526	2000 × 8	16384	2000
Sorter	16	8 bits compare	2	47	100 × 2	0	830
Bit Serial Multiplier	16	1 bit adder	8 1.5ex1	46	48 × 8	0	306
Bit serial square rooter	14	1 bit signed digit adder	6	52	32 × 6	0	702

3.1 Number of PEs

We set up the appropriate number exu of processor elements as an initial value. It must be smaller than the algorithm size U. Among the possible exu, we find an optimal one, which achieves the minimum area-time product for the given algorithm. In other side, the number of PEs can also be fixed firstly, then, search for optimal scheduling and resource allocation vectors according to this value and algorithm size.

When U is a multiple of exu, the scheduling and resource vectors/matrices become very simple. So does the address generation unit. We consider the case that there is a integer t_2 satisfying

$$t_2 = \frac{U}{exu}. \tag{1}$$

If it is not the case, we can generate this situation by adding dummy operations in the uniform recurrence equations.

3.2 Scheduling and Resource Allocation

Under the fixed exu, we try to derive optimal mapping matrices for the operation scheduling and resource allocation respectively. The mapping of an algorithm onto an architecture is performed by setting up the time coordinate and the space coordinate in the index space. These linear transformations are expressed by both scheduling vector $T = (t_1, t_2, t_3)$ and resource allocation matrix $S = (o_1, o_2, o_3)$. Each node of an index vector $(i_1, i_2, i_3)^T$ is mapped to time t and PE number PE as

$$t = T(i_1, i_2, i_3)^T \tag{2}$$

$$PE = S(i_1, i_2, i_3)^T \tag{3}$$

The operation scheduling with T must satisfy the valid operation ordering so as not to violate the data dependence relations. In addition to that, it does not require more concurrent executing operations than the number exu of processor elements. On the other hand, the resource mapping with S must satisfy the resource conflict free condition so that the concurrent executing operations must not be allocated to the same processing element. Besides, it determines the physical distance of the data communications among the processing elements. We intend to realize the local communications among the processor array so as to reduce the area and time required for the interprocessor communications. Therefore, it is very difficult to get the optimal T and S theoretically.

Our purpose is to derive the practical solution for the array processing, even if the solution is not global optimal one. In this sense, we restrict our search area to get the practical feasible solutions. We have investigated optimal mapping vectors under the condition that the

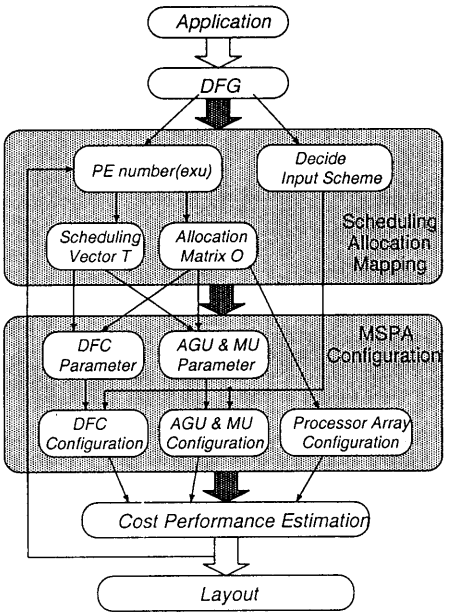

Fig. 2. MSPA Architecture Design Procedure

number exu of processor elements are less than the algorithm size U. As a result, we succeed to get a series of theorems for the optimal mapping vector and matrix. Since both the scheduling and allocation vector/matrix are derived in a closed form, the design become so simple for the fixed exu. [4]

Let (t_1, t_2) be the optimal scheduling vector for the 2D uniform recurrence algorithm. The optimal operation scheduling vector for the uniform recurrence algorithm with multiple U iteration loops is given by

$$T = (1, \frac{U}{exu}, \frac{U^2}{exu}) \tag{4}$$

The following resource allocation matrix can realize the conflict free allocation on $exu \times 1$ linear processor matrix.

$$S = (0, 1, U) \quad M = exu \tag{5}$$

We can also prove that the resource allocation matrix described by eq.(5) satisfies the local communication condition for all the precedence vectors in the first quadrant of its resource allocation coordinates.

Fig.4 and 5 shows the scheduling and the resource allocation of our example of matrix-vector multiplication. This example shows the case of $exu = U$ for 2D algorithm of the size $U = 6$. The scheduling vector T and S are given as

$$T = (1, 2) \tag{6}$$
$$S = (0, 1) \quad M = 3 \tag{7}$$

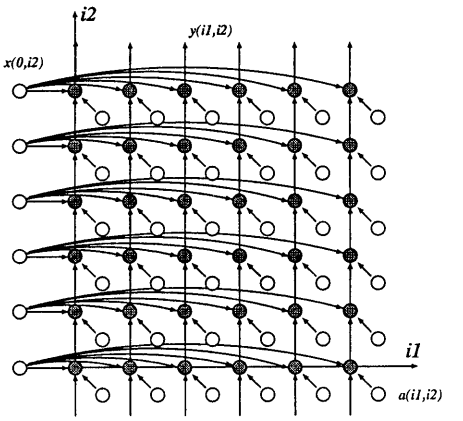

Fig. 3. Matrix-Vector Multiplication Example

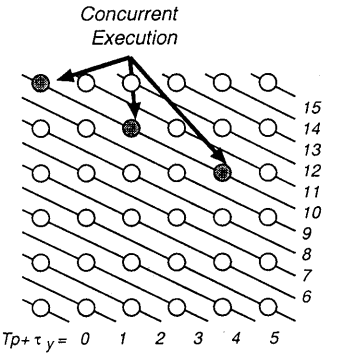

Fig. 4. Scheduling of Example

3.3 Data Transfer

Both the scheduling and resource allocation vector/matrix determine the data flow scheme to or from or among the processor array. In most cases, the data produced by PEs can be circulated within the processor array by the local communications. Let Δp be an precedence vector. Then, the result of PE is sent to another ΔPE farther PE than PE , after Δt cycles which are derived by

$$\Delta t = T\Delta p \tag{8}$$
$$\Delta PE = S\Delta p. \tag{9}$$

Fig.6 shows the data stream to 3 PEs. The result of each calculation of PEs for index (i_1, i_2) are consumed at index $(i_1, i_2 + 1)$. Therefore, the precedence vector is $(0, 1)^T$. Since the above equation become

$$\Delta t = (1, 2)(0, 1)^T = 2 \tag{10}$$

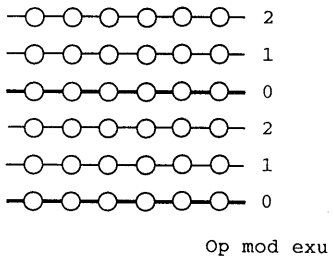

Fig. 5. Resource Allocation of Example

$$\Delta PE = (0,1)(0,1)^T = 1, \quad (11)$$

each PE transfers the result to next adjacent PE with 2 execution cycle delays.

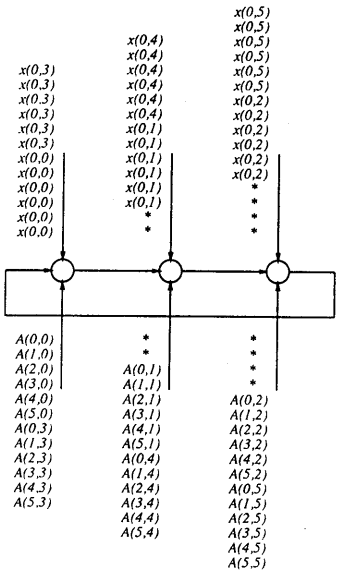

Fig. 6. Data Stream

3.4 AGU and MU

AGU can be constructed to manipulate the data transfer from MU(memory units). In case the data corresponding to the index node are stored in MU with index addresses, generation of address can be realized by generating invoking index points. The plural invoking indices $p = (i_1, i_2, i_3)^T$ are derived for the every time step t as

$$t = Tp = t_1 i_1 + t_2 i_2 + t_3 i_3. \quad (12)$$

By solving this equation for the time t, we get exu active indecies. However, we can get those $p = (i_1, i_2, i_3)^T$ elegantly by simple hardware.

Instead of MU and AGU, data format converter registers can be employed to convert the input data format from word-serial to word-parallel. Since their hardware models are decided, we produce their design parameters to derive the configuration of memory units with AGU and/or DFC.

3.5 Data I/O

The input data must be provided to the processor array. Since there are plural concurrently executing PEs, PEs require the multiple data flows from the input I/O. We try to restrict the number of I/O port so as to solve the I/O bottleneck problem.

For word-serial input, we classify the input scheme into two kinds of schemes. One is compile-time input, which must be stored in memory units before the execution starts. The stored data in memory units are loaded to the processor array, driven by the address generation units. The other is run-time input, which can be loaded in while the execution performs.

4 Programmable MSPA

4.1 Design Specification

A programmable MSPA architecture used in a developing dsp application is given as an example. The number of processing elements (PEs) denoted by exu is fixed to 4 in the MSPA. The function of the PEs is multiply-accumulation. The size of algorithms is denoted as U, In order to simplify the design, we assume that the U and exu keep the relation with integer times [4], and U is constrainted to be 4, 8, 12, 16. The outline of MSPA control unit is shown in Fig. 10.

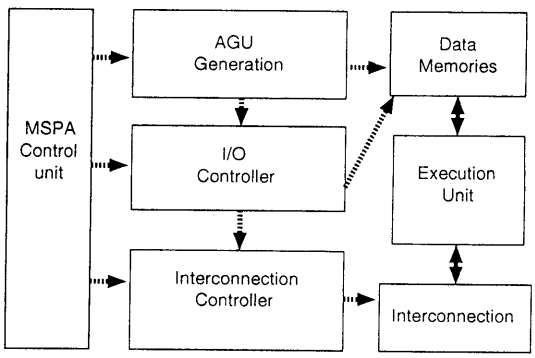

Fig. 7. Design Outline of Programmable MSPA

There are two data memories, one program memory in the chip. The MSPA works as a special work mode of the

general-purpose dsp. In MSPA-mode, data will be input from two data memories in the input-write cycle of as 4 times high as execution cycle. Memory read cycle will be synchronized by execution cycle. The data memories are one-write, one-read with free read and write function.

4.2 MSPA Datapath

The data-path in MSPA-mode is shown in Fig.8.

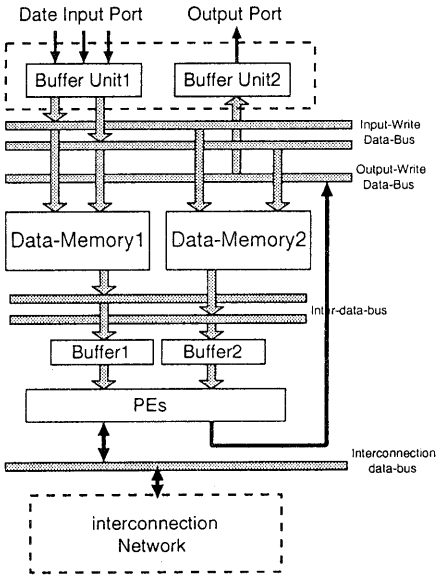

Fig. 8. Data-path in MSPA-mode

Therefore, there are at least four data registers for keeping input data from each data memory.

The pipeline schedule is shown in Fig.9. The start time of the 3 pipelines in Fig.9 will be decided by the MSPA start and enable instruction. When instruction "MSPA start" is decoded, the first input-write pipeline will be driven to start data load procedure. After one clock, the read pipeline will be driven by the write start signal. After the instruction "MSPA parameter-set", the MSPA finish reconfiguration procedure and fix the hardware mode. The execution stage will begin as soon as the instruction "MSPA enable" comes. The active time of output write pipeline will be determined by the *Out-write decode* logic which is reconfigurated by the *MSPA parameter-set* instruction.

4.3 Interconnection Design

Local communication between PEs can be held corresponding to different algorithm size or operation schedule. Then, the reconfigurable MSPA architecture is realized by this simple programmable way. We designed this

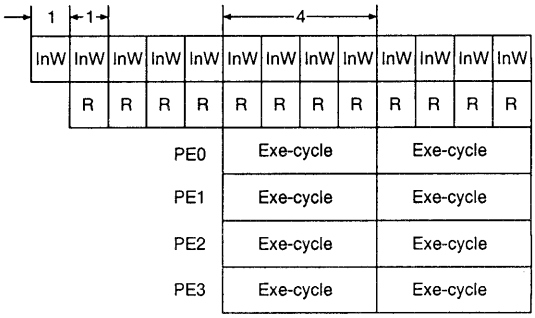

Fig. 9. Pipeline Schedule in MSPA-mode

part in schematic level by cadence in order to minimize the number of gates in this part to minimum.

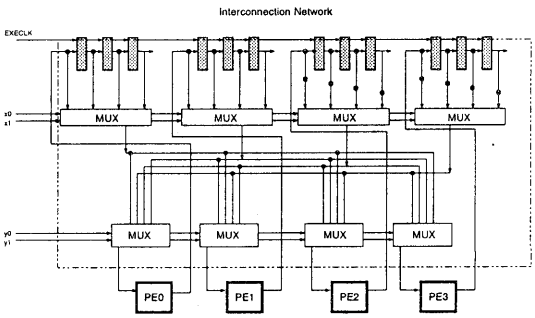

Fig. 10. Interconnection Network Architecture

5 MSPA Controller Design

5.1 Controller Configuration

The controller in MSPA-mode is a programmable and configurable logic unit. Controller consists of AGU(Address Generation Unit) controller, run-time input controller, interconnection controller and out-write controller.

The *AGU controller* control the data fetch to each PE. It generates of *exu* invoking indecies and produces the actual address of Memory units. At the same time, it produces the destination PEs for each read data from Memory units. Then, the data is controled to be transfer to appropriate PEs. The *Run-time input controller* start input-write (input from I/O port and write to Memory unit) pipeline based on *MSPA start* instruction and generate input-write timing, read-timing, and execution timing signals. The *Interconnection controller* is used to reconfigurate the local communication path between PEs. The *Out-write controller* is used to calculate the

start and interval time of output write pipeline of each PE.

5.2 AGU Controller

In order to generate the concurrent invoked indecies, we prepare the *exu* index point registers $R_u = (R_{u1}, R_{u2}, R_{u3})$ for u-th PE ($u = 0, 1, \cdots exu - 1$), each sub-register of which have a length of U. Each index point register R_u generates only one index point of the u-th, $u+exu$-th, $\cdots$ rows in the converted 2D index point space at a time, which are corresponding to the PE allocation. Therefore, each register generates active indecies for each PE.

The increment of time t usually makes an increment of i_1. In case R_{u1} reaches at U, R_{u1} is reset and since

$$t = t_1 U + t_2 i_2 + t_3 i_3 = t_1 0 + t_2(i_2 + exu) + t_3 i_3 \quad (13)$$

holds, a carry produced by R_{u1} must increment i_2 by *exu*.

In case R_{u2} and R_{u1} reaches at U, R_{u2} and R_{u1} are reset and since

$$t = t_1 U + t_2 U + t_3 i_3 = t_1 0 + t_2 0 + t_3(i_3 + 1) \quad (14)$$

holds, a carry produced by R_{u2} and R_{u1} must increment i_3 by 1.

The operation of the index point register is as follows.

1. R_{u1} increments +1 at every time steps and produces a carry when R_{u1} exceeds U.
2. R_{u2} increments $+exu$ by the carry of R_{u1}. and produces a carry when R_{u2} exceeds U.
3. R_{ui} for $i = 3, \cdots, n$ increments +1 by the carry of $R_{u(i-1)}$.

Run time input controller controller is also realized in similar fashion to AGU controller.

6 Execution Pipelining

MSPA and the systolic array utilize instruction-level pipeline processing for the regular structured algorithms. Their pipeline fashion is that each execution unit repeatedly performs a part of processing for the target algorithm one after another.

The further speed-up can be achieved by employing the pipeline execution units in order to exploit execution-level pipeline processing. However, the pipeline executions requires plural independent operations, among which there are no precedence relation or data transfers. Fortunately, MSPA design matches with this extension. MSPA scheduling and resource allocation based on the specified PE number exploit the parallelism of the same number of operations as PE number. It can simply be modified by design on PEs with pipeline execution units.

Let *exu* be the number of PEs and *stage* be the number of pipeline stages of all execution units in PEs. We perform the scheduling of MSPA operations with $exu \times stage$ execution units. If the algorithm size U is still less than $exu \times stage$, MSPA design schedule the $exu \times stage$ operations to be invoked concurrently. While, the resource allocation is performed in the same way with *exu*. Thus, the *stage* operations are assigned to each pipeline PEs.

Fig.11 shows $exu = 4$ and $stage = 4$ scheduling in MSPA design. Fig.12 shows its operation scheduling in each PE. As shown in Fig.11, each PE run *stage* times as fast as no pipeline execution units, but requests *stage* times as many data supply as no pipeline one. Thus, MSPA scheduling exploits independent operations so easily that it is easy to be extended to achieve not only instruction pipeline, but execution pipeline operations.

In practical design, the clock rate of $5[ns]$ is achieved both for memory access and 4 stage pipeline operation of a multiply-add. Therefore, our current design expects that a chip including such 4 pipeline stage PEs can achieve 0.8G operations per a second.

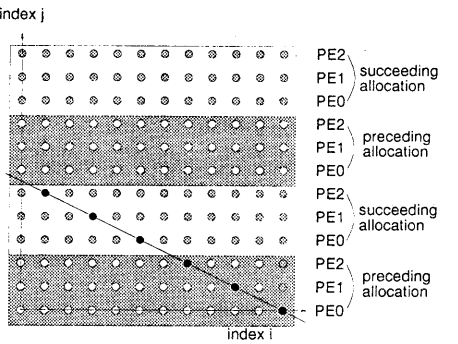

Fig. 11. MSPA with pipeline execution units

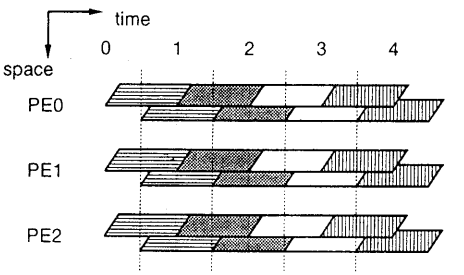

Fig. 12. MSPA with pipeline execution units

7 Conclusion

In this paper, we present the design for programmable MSPA. The simple design procedure of MSPA under

the fixed number of PE is excellently embedded into micro-architecture. Currently, we are developing programmable MSPA and further studying the further extension to general purpose array processor including programmable MSPA function.

ACKNOWLEDGMENTS

This work has been engaged as a project in CAD21 Research Body of Tokyo Institute of Technology. We wish to thank all the members of CAD21 for their suggestions and cooperations.

References

[1] K. Ganapathy and B.W. Wah, "Optimal synthesis of algorithm-specific lower-dimensional processor arrays" in *IEEE Trans. on Parallel and Distributed System, IEEE*, (accepted to appear) 1996

[2] R.H.Kuhn, "Optimization and interconnection complexity for parallel processors, single stage networks and decision trees " in Ph.D dissertation, Department of Computer Science, University of Illinois, Urbana, IL, Oct. 1980.

[3] D.I. Moldovan, "On the analysis and synthesis of VLSI algorithms" in *IEEE Transactions on Computers*, vol.C-31, pp.1121-1126, Nov. 1982

[4] Dongju Li and Hiroaki Kunieda, "Memory Sharing Processor Array(MSPA) Architecture" in *IEICE TRANS. Fundamentals*, Vol. E79-A, No. 12, (to appear) Dec. 1996.

[5] H. Kunieda and K. Hagiwara. "Effective processor array architecture with shared memory" in *IEEE, APCCAS'94*, pp.113-118,1994

[6] K.Ganapathy and B.W. Wah, "Optimazing general design objectives in processor-array Design" in *Pro. IEEE Int'l Parallel Processing Symposium*, April 1994, pp.295-302.

[7] H. Kunieda, Y.S. Liao, D.J. Li, K. Ito "Automatic design for Bit-serials MSPA architecture" in *Proceedings of ASP-DAC'95/CHDL'95/VLSI'95*, pp.27-32, 1995