Exploiting Application Parallelism for Processor-in-Memory Architecture

Slo-Li Chu
Department of Information and Computer Engineering
Chung Yuan Christian University
Chung-Li, TAIWAN
slchu@cycu.edu.tw

Tsung-Chuan Huang
Department of Electrical Engineering
National Sun Yat-sen University
Kaohsiung, TAIWAN
tch@mail.nsysu.edu.tw w

Abstract

Continuous improvements in semiconductor fabrication density are supporting new classes of System-on-a-Chip (SoC) architectures that combine extensive processing logic/processor with high-density memory. Such architectures are generally called Processor-in-Memory or Intelligent Memory and can support high-performance computing by reducing the performance gap between the processor and the memory. This architecture combines various processors in a single system. These processors are characterized by their computation and memory-access capabilities. Therefore, a novel strategy must be developed to identify their capabilities and dispatch the most appropriate jobs to them in order to exploit them fully. Accordingly, this study presents an automatic source-to-source parallelizing system, called SAGE, to exploit the advantages of Processor-in-Memory architectures. Unlike conventional iteration-based parallelizing systems, SAGE adopts statement-based analyzing approaches. This study addresses the one-host and one-memory processor configuration. The strategy of the SAGE system, in which the original program is decomposed into blocks and a feasible execution schedule is produced for the host and memory processors, is investigated as well. The experimental results for real benchmarks are also discussed.

Keywords: Processor-in-Memory, statement analysis, SAGE, SoC.

1. Introduction

In current high-performance computer architectures, the processors run many times faster than the computer's main memory. This performance gap is often referred to as the Memory Wall [25]. This gap can be reduced using the System-on-a-Chip strategy, which integrates the processor(s) and memory on a single chip. The rapid growth in silicon fabrication density has made this strategy accomplished. The Semiconductor Industry Association's Technology Roadmap predicts that a single high-end microprocessor die will contain approximately 84 million logic transistors by 2009. The density of dynamic random access memory (DRAM) is increasing even more rapidly. By 2009, a state-of-the-art DRAM chip is expected to have a capacity of 2Gbytes. Larger scale problems will be easily handled "on-chip" using these tremendous increases in memory and logic density [19]. Accordingly, many researchers have addressed integrating computing logic and high density DRAM on a single die [7, 13, 17, 21, 22] as so-called them Processor-in-Memory (PIM) or Intelligent RAM (IRAM).

Integrating DRAM and computing logic on a single integrated circuit (IC) die generates PIM architecture with several desirable characteristics. First, the physical size and weight of the overall design can be reduced. As more functions are integrated on each chip, fewer chips are required for a complete design. Second, very wide on-chip buses between the CPU and memory can be used, since DRAM is located with computing logic on a single die. Third, eliminating off-chip drivers reduces the power consumption and latency [19].

The design philosophy of PIM chips is to replace the main memory chips in a computer system rather than to rebuild an entirely new system. Accordingly, PIM chips may act as co-processors when the main processor spawns them. This approach is employed by Active Page [21], DIVA [7] and FlexRAM [11, 13, 14], among others.

This class of architectures constitutes a hierarchical hybrid multiprocessor environment by the host (main) processor and the memory processors. The host processor is more powerful but have a deep cache hierarchy and higher latency when

accessing memory. In contrast, memory processors are normally less powerful but have a lower latency in memory access. The main problems addressed here concern the method for dispatching suitable tasks to these different processors according to their characteristics to reduce execution times, and the method for partitioning the original program to execute simultaneously on these heterogeneous processor combinations.

Previous studies of programming for PIM architectures [7, 11, 13, 17, 22] have concentrated on spawning as many processors as possible to increase speedup, rather than the capability difference between the host and memory processors. However, such an approach does not exploit the real advantages of PIM architectures. This study refines our earlier SAGE (Statement-Analysis-Grouping-Evaluation) system [4, 9, 10] that integrates statement splitting, weight evaluation and scheduling mechanism. The original scheduling mechanism is improved to generate a superior execution schedule with a reduced time complexity, using our new *seesaw dispatching* mechanism. A weight evaluation mechanism is established to obtain more precise expected execution time, called *weight*. The SAGE system can automatically analyze the source program, generate a Weight Partition Dependence Graph (WPG), determine the weight of each block, and then dispatch the most suitable blocks for execution on the host and memory processors.

The rest of this paper is organized as follows: Section 2 introduces the architectures of PIM. Section 3 reviews other related work. Section 4 describes our SAGE system and the algorithms. Section 5 illustrates an example. Section 6 presents experimental results. Conclusions are finally drawn in Section 7.

2. Processor-in-Memory Architecture

2.1 Architecture Description

Figure 1 depicts the organization of the PIM architecture evaluated in this study. It contains an off-the-shelf processor— P.Host, and a PIM chip. The PIM chip integrates one memory processor— P.Mem with 64 Mbytes of DRAM. This architecture is derived from FlexRAM [11, 13, 14]. The tiny memory processors (P.Array) in the original FlexRAM are omitted to reduce the complexity of analysis. The techniques presented in this paper involve one P.Host and one P.Mem configuration, and can be extended to support multiple P.Mems.

Table 1 lists the main architectural parameters of the PIM architecture. P.Host is a six-issue superscalar processor that allows out-of-order execution and runs at 800MHz, while P.Mem is a two-issue superscalar processor with in-order capability and runs at 400MHz. P.Mem has lower memory access latency than the P.Host since the former is integrated with DRAM. Thus, computation-bound codes are more suitable for running on the P.Host, while memory-bound codes are preferably running on the P.Mem to increase efficiency.

The PIM chip is designed to replace regular DRAMs in current computer systems, and must therefore meet a memory standard that involves additional power and ground signals to support on-chip processing. One such standard is Rambus [5], so the PIM chip is designed with a Rambus-compatible interface.

Table 1. Major parameters of the PIM architecture.

P.Host	P.Mem	Bus & Memory
Working Freq: 800 MHz	Working Freq: 400 MHz	Bus Freq: 100 MHz
Dynamic issue Width: 6	Static issue Width: 2	P.Host Mem RT: 262.5 ns
Integer unit num: 6	Integer unit num: 2	P.Mem Mem RT: 50.5 ns
Floating unit num: 4	Floating unit num: 2	Bus Width: 16 B
FLC_Type: WT	FLC_Type: WT	Mem_Data_Transfer: 16
FLC_Size: 32 KB	FLC_Size: 16 KB	Mem_Row_Width: 4K
FLC_Line: 64 B	FLC_Line: 32 B	
FLC_Replace policy: LRU	FLC_Replace policy: LRU	
SLC_Type: WB	SLC: N/A	
SLC_Size: 256 KB		
SLC_Line: 64 B		
Replace policy: LRU		
Branch penalty: 4	Branch penalty: 2	
P.Host_Mem_Delay: 88	P.Mem_Mem_Delay: 17	

** FLC stands for the first level cache, SLC for the second level cache, BR for branch, RT for round-trip latency from the processor to the memory, and RB for row buffer.*

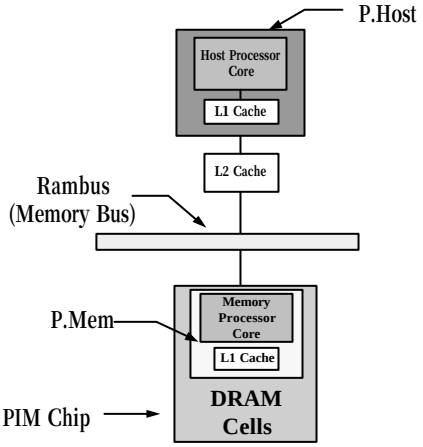

Fig. 1. The organization of PIM architecture.

2.2 Synchronization Mechanism

In FlexRAM architecture, a hardware synchronization mechanism is provided [13, 14] to synchronize P.Host and P.Mem when they enter the computation stage. No signals other than those required for memory access should be introduced to ensure compatibility with conventional processors and system buses. However, recent development in memory technology, such as Rambus, enables P.Host and P.Mem to exchange information without adding new signals for memory access.

In PIM architecture, P.Mem must be initiated by P.Host and cannot perform tasks autonomously. The P.Host starts a P.Mem by sending it an entry point (Program Counter value) and an enabling signal. Sending data from P.Host to P.Mem only requires to access memory since P.Mem resides in the memory system of P.Host. Hence, the memory location used to pass information from P.Host to P.Mem must have a fixed address that is independent of the P.Host context switch. The Rambus standard provides additional register space in the memory space. The registers in DRAM chips are for access mode control, packet packing and unpacking. The PIM architecture uses the registers in the register space as command registers from P.Host to P.Mem. Sending an entry point and an enabling signal can be regarded as writing to the register space of DRAMs.

A synchronization variable is used to implement a barrier between the P.Host and P.Mem. The synchronization variable can be a register in Rambus register space. By polling the synchronization variable, P.Host can synchronize with P.Mem. The PIM architecture also provides a barrier interface, similar to the general barrier function call, to enable the programmer to synchronize processors easily.

2.3 Cache Coherence Mechanism

The contents of the cache become incoherent during execution when some of the code is executed on P.Host and some on P.Mem simultaneously. The latest version of data must be obtained when a processor accesses a variable to avoid incorrect results. FlexRAM architecture provides two simple primitives for P.Host, *write-back* and *invalidation*, to manipulate the cache [14]:

write-back: Before P.Mem starts to execute, the "write-back" primitive forces P.Host to write back all the dirty lines in its caches to the main memory where P.Mem may read or modify. When a line is written back, the corresponding P.Mem's cache is also updated. This primitive ensures that P.Mem sees the latest version of data.

invalidation: After P.Mem finishes the execution, P.Host invalidates all lines in its cache that were updated by P.Mem. This primitive ensures P.Host sees the latest version of data.

The SAGE system will automatically insert the primitive of *barrier*, *write-back* or *invalidation* in appropriate locations if required.

3. Related Work

Some relevant studies on PIM architecture, with reference to the architecture and the compiler, are introduced.

3.1 Architecture Aspects

Several PIM architectures integrate processors and memory on a single chip.

(a) Vector Processors in DRAM

The VIRAM (Vector IRAM) [22] combines vector units, a scalar processor, and DRAM on a single chip. The VIRAM processor consists of an in-order dual-issue superscalar processor and a first-level cache, tightly integrated with a vector execution unit and multiple pipelines. The memory system includes 96MB of DRAM, connected to the scalar and vector units via a crossbar. If 16 add-multiply units are used and are clocked at 500MHz, VIRAM can run Linpack at 8 GFLOPS, five times faster than a Cray T-90. Other benchmarks, such as multimedia applications, are reported to run well on VIRAM architecture.

(b) Logic-in-Memory

Several investigations on integrating logic with memory have been undertaken. C•RAM (computation RAM) [6] uses special logic in DRAM as an MPEG encoder. SmartSIMM [15] uses integrated logics to overcome the I/O Bus bottleneck. Among these logic-in-memory architectures, Active Page [21] is the most general-purpose scheme. Logically, an Active Page consists of a page of data and a set of associated functions that operate on the data set. Physically, this kind of PIM chip, called RADRAM, carries several Active Pages. Each 512KB memory block is associated with an LE (Logic Element) which is built by FPGAs. The logic function of an Active Page can be programmed when the application is compiled by a

special compiler that can hardwire the FPGA logic at compile-time. The cost of fabricating a RADRAM chip is not high because its logic part is FPGA rather than complicated processors. Conceptually, FPGA can be regarded as a re-configurable processor and RADRAMs can be used for several applications if LEs are re-configured. In RADRAM, only simple functions can be implemented by an Active Page, since only 1K transistors are present on each LE and these LEs are in FPGA format. The implemented functions are simple set operations, such as array insert/delete, data search, basic arithmetic computation, table fill, index comparison and data gather/scatter. According to [21], these operations are accelerated up to 1000 times the speed of a simple scalar RISC running at 1GHz with a non-pipelined memory system.

(c) Processing-in-Memory

Some investigations on high-speed computation integrate multiple processor cores with memory. EXECUBE [8], unlike VIRAM that is memory-centric, is a processor-centric architecture. EXECUBE integrates sixteen 32KB memory banks and eight CPUs with a four-link DMA on the center of the chip. Like the Cray T series on a single chip, EXECUBE can perform highly parallel computations. However, the biggest challenge comes from implementation. The first, current technology only supports a small memory bank in the above architecture because several processors on a chip occupy too much area and the die area is not large enough to accommodate more memory. The second challenge is the wiring for connectivity, which might occupy a large area and limit the speed of the processor.

3.2 Compiler Aspect

Some investigations of compiler have been studied to utilize the benefits of PIM architectures. Yelick *et al.* [12] designed a VIRAM compiler for Vector-IRAM systems based on the vectorizing compiler of the Cray system. They modified the code generator of the original CRAY compiler to generate both MIPS scalar codes and vector codes to fit the architectural definition of VIRAM. Barua *et al.* [2] proposed a compiler, called Maps, to generate codes for the Raw chip that is composed of a set of tiles which integrate processor and memory. The main contributions of their work are the distribution of data across several tiles and disambiguating memory access to specific tiles. Using more tiles can increase more memory access parallelism. Disambiguating memory access can enables the compiler to manage the communication between tiles efficiently. In addition to the above systems, Gupta *et al.* [27] developed a compiler that allows different cache line sizes for various portions of the program. Moritz *et al.* [20] designed a framework, called FlexCache, to cache data at compile-time. This framework is based on a flexible software platform, with the possibility of adding hardware resources if required.

4. Methodology

Most current parallelizing compilers focus on the transformation of loops to execute all or some iterations concurrently, in a so-called *iteration-based* approach. This approach is suited to homogeneous and tightly coupled multi-processor systems. However, it has an obvious disadvantage for heterogeneous multi-processor platforms because iterations have similar behavior but the capabilities of heterogeneous processors are diverse. Therefore, a different approach is adopted here, using the statements in a loop as a basic analysis unit, called *statement-based* approach, to develop the SAGE system.

SAGE *(Statement-Analysis-Grouping-Evaluation)* is an automatic parallelizing compiler, that partitions and schedules an original program to exploit the specialties of the host and the memory processor. At first, the source program is split into blocks of statements according to dependence relations. Then, the *Weight Partition Dependence Graph (WPG)* is generated, and the weight of each block is evaluated. Finally, the blocks are dispatched to either the host or the memory processors, according to which processor is more suitable for executing the block. The major difference between SAGE and other parallelizing systems is that it uses *statement* rather than *iteration* as the basic unit of analysis. This approach can fully exploit the characteristics of statements in a program and dispatch the most suitable tasks to the host and the memory processors.

Table 2. A simple fully parallelizable program.

Program	*Weight for P.Host*	*Weight for P.Mem*
DO I = 1 to N		
S1: A= A mod B	3	6
S2: C= D[I] + E	5	1
S3: F= G[I] + H[I]	6	2
ENDDO		

Table 2 presents a simple example to demonstrate the advantages of statement-based parallelization. The program is fully parallelizable and can be partitioned into statements or iterations. The table lists the assumed statement weights for the P.Host and P.Mem. Table 3 shows five parallelization cases in Table 2 and their execution times. The first two involve executing the program solely on P.Host and P.Mem, respectively. Case 3 parallelizes the program using conventional parallelizing compilers, such as SUIF [8] or Polaris [3] to identify the parallelizable loops and dispatch them for execution on P.Host and P.Mem. This approach only achieves good speedup for processors with homogeneous capabilities (including memory access latency, computing power, and so on). In case 4, the iterations are dispatched to the processors according to the processors' capabilities, but the compiler does not consider the discrepancies among processors in executing statements. Case 5 uses the statement-based analysis approach (i.e., optimized by SAGE). This approach outperforms all the others since it dispatches *statements* to P.Host and P.Mem by accounting for the characteristics of statements and the capabilities of processors, motivating the development of the SAGE system for asymmetric multiprocessor environments. Figure 2 illustrates the organization of the SAGE system.

Table 3. Five parallelizing cases and their execution times.

Case	*Description*	*Execution Time*
1	Host processor operates solely	Latency = [PH(S1)+ PH(S2)+ PH(S3)]*# of iterations = (3+5+6)N = 14 N
2	Memory processor operates solely	Latency = [PM(S1)+ PM(S2)+ PM(S3)]* # of iterations = (6+1+2)N = 9 N
3	Host and memory processors cooperate in symmetric workload	Latency = max((3+5+6)* 0.5N, (6+2+1)* 0.5 N) = 7 N
4	Host and memory processors cooperate in asymmetric workload by parallelizing *iteration* space of the loop	Dispatch workload in proportion to the capability ratio of PH and PM obtained from Case 1 and Case 2: PH: PM = 9:14 Latency=14* (9/23)N= 5.48 N
5	Host and memory processors cooperate in asymmetric workload by SAGE	Latency = max (PH(S1) * N, PM(S2,S3)*N) = 3 N (Here S1 is more suitable for P.Host,

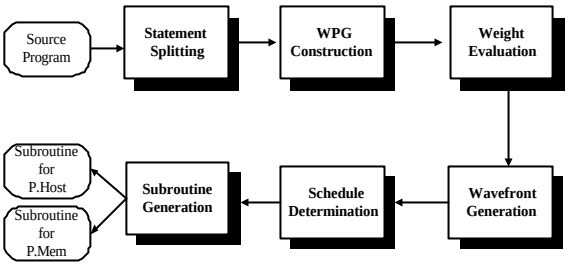

Fig. 2. The sequence of compiling stages in SAGE.

4.1 Statement Splitting and WPG Construction

Statement Splitting splits the dependence graph by Node Partition Π in [18]. WPG Construction constructs the Weighted Partition Dependence Graph (WPG), to be used in the subsequent stages of Weight Evaluation, Wavefront Generation and Schedule Determination.

The definitions relevant to Statement Splitting are introduced as below.

Definition 1 (Loop Denotation) [18]

A loop is denoted by $L = (I_1, I_2, \ldots I_n)(S_1, S_2, \ldots S_k)$, where I_j, $1 \le j \le n$, is a loop index, and S_d, $1 \le d \le k$, is a body statement which may be an assignment statement or another loop.

Definition 2 (Node Partition Π) [18]

For a given loop L on the dependence graph G, we define a node partition Π of $\{S_1, S_2, \ldots S_d\}$ in such a way that S_k and S_l, $1 \le k \le d$, $1 \le l \le d$, $k \neq l$, are in the same block (cell) of partition if and only if $S_k \Delta S_l$ and $S_l \Delta S_k$ where Δ is an indirect data dependence relation.

On the partition $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$, we define partial ordering relations α, $\alpha^{\wedge}$, and α^{o} as follows.

For $i \neq j$:

1) $\pi_i \alpha \pi_j$ iff there exist $S_k \in \pi_i$ and $S_l \in \pi_j$ such that $S_k \delta S_l$, where δ is the true dependence relation.

2) $\pi_i \alpha^{\wedge} \pi_j$ iff there exist $S_k \in \pi_i$ and $S_l \in \pi_j$ such that $S_k \delta^{\wedge} S_l$, where $\delta^{\wedge}$ is the anti dependence relation.

3) $\pi_i \alpha^{o} \pi_j$ iff there exist $S_k \in \pi_i$ and $S_l \in \pi_j$ such that $S_k \delta^{o} S_l$, where δ^{o} is the output dependence relation.

Based on the definition, the statements form a block (cell) of partition Π if and only if there exist a directed dependence cycle. Two blocks have true/anti/output dependence if and only if there exist a true/anti/output dependence between two statements, one for each block.

Definition 3 (Weighted Partition Dependence Graph)[9][10]

Given a node partition Π defined in Definition 2, we define a weighted partition dependence graph *WPG(B,E)* as follows. For each $\pi_i \in \Pi$, there is a corresponding node $b_i \langle I_i, S_i, W_i, O_i \rangle \in B$, where I_i denotes the loop index; S_i represents the body statements; W_i is the weight of node i in the form of W_i *(PH,PM)* with *PH* and *PM* be the weights of P.Host and P.Mem respectively; and O_i is the execution order of this node. There is an edge $e_{ij} \hat{I} E$ from b_i to b_j if b_i and b_j have dependence relations α, $\alpha^{\wedge}$, and α^{o} defined in Definition 1. These dependence relations are respectively denoted by $\longrightarrow$, $\xrightarrow{anti}$, and $\xrightarrow{O}$.

Based on these three definitions, we propose a *Statement Splitting* algorithm (Algorithm 1) to partition the loops:

Algorithm 1. (Statement Splitting Algorithm)

Given a loop L = (I_1, I_2, …. I_d) (S_1, S_2, …. S_d)

Step 1: Construct dependence Graph G by analyzing subscript expressions and index pattern.

Step 2: Establish a node partition Π on G as defined in Definition 2. If there are large blocks caused from control dependence relations, convert control dependence into data dependence first [16], and then partition the dependence graph.

Step 3: On the partition Π, establish a weighted partition dependence graph WPG(B,E) defined in Definition 3.

4.2 Weight Evaluation

Two approaches to evluating weight can be taken. One is to predict the execution time of programs by profiling the dominant parts [28]. The other considers the operations in a statement and estimates the program execution time by looking up a statistical operations table [24]. The former method may be more accurate, but the predicted result cannot be reused; the latter can determine statements for suitable processors but the estimated program execution time is not sufficiently accurate. Hence, the *Self-Patch Weight Evaluation* scheme was designed to combine the benefits of both approaches. For a detailed description of this scheme, please refer to [4]

4.3 Wavefront Generation and Schedule Determination

This section presents an algorithm for scheduling P.Host and P.Mem. In our previous work [9, 10], a method to obtain a load-balanced schedule for P.Host and P.Mem was proposed. However, the method has two weaknesses. First, it concentrates more on balancing the workload of processors but less on the capability difference between P.Host and P.Mem. Second, the dispatch mechanism requires large time complexity. In this paper, a new scheduling mechanism has been devised to solve these two problems. In solving the first problem, the new mechanism classifies the blocks into two sets, according to the weight difference between P.Host and P.Mem. Then, suitable blocks of each set are dispatched to P.Host and P.Mem, respectively. A *seesawing dispatch* mechanism is devised to reduce the time complexity. The weights of the blocks in partition Π are computed first, and the execution order of each block is then determined according to the dependence relations between the blocks. Blocks with the same execution order are assigned to a *wavefront*. Wavefronts are executed in sequence, but the blocks in the same wavefront will be executed simultaneously, scheduled to P.Host and P.Mem according to their weights.

Given a weighted partition dependence graph *WPG=(B,E)*, in which the weight and the order of the blocks has not been determined, the scheduling algorithm involves the following steps.

Step 1: Initialize the execution order of each block and determine the P.Host weight and P.Mem weight of each block by the weight evaluation mechanism mentioned earlier.

Step 2: Determine the execution order (wavefront) of each block by the following rule: A block's execution order equals the maximum execution order of all of its successors plus one.

Step 3: Set wavefront number j =1. Perform the following actions until j = maximum execution order:

3.1 Store the blocks whose execution order

equals *j* in the set *wf_tmp*.

3.2 Divide *wf_tmp* into two sets, *ph_tmp* and *pm_tmp*. The blocks with P.Host weight = P.Mem weight are stored in *ph_tmp*. The other blocks (*i.e.*, P.Mem weight < P.Host weight) are put into *pm_tmp*. Restated, the blocks in *ph_tmp* perform better if executed on P.Host.

3.3 Sort *ph_tmp* and *pm_tmp* in order of decreasing P.Host weight and P.Mem weight, respectively. Set *token* = P.Host.

3.4 Perform the *seesawing dispatch* mechanism:

3.4.1 If *token* = P.Host, perform Step 3.4.1.1, else perform Step 3.4.1.2

3.4.1.1 Put the first block (with largest P.Host weight) from *ph_tmp* into set *ph_sch*. If *ph_tmp* is empty, put the largest block from *pm_tmp* into *ph_sch*.

3.4.1.2 Put the first block (with largest P.Mem weight) from *pm_tmp* into set *pm_sch*. If *pm_tmp* is empty, put the largest block from *ph_tmp* into *pm_sch*.

3.4.2 If the total weight of P.Host in *ph_sch* = the total weight of P.Mem in *pm_sch*, set *token* = P.Mem. (This means P.Mem requires more blocks to achieve load-balance), else set *token* = P.Host.

3.4.3 If both *ph_tmp* and *pm_tmp* are empty, then generate wavefront $Wf_j = \{ph\_sch, pm\_sch\}$; insert a barrier to synchronize P.Host and P.Mem, and set $j = j + 1$; else return to Step 3.4.1.

4.4 Example

A simple synthetic program, shown in Fig. 3, illustrates the processes of SAGE. In the *Statement Splitting* stage, the loops are partitioned into seven separate loops (by Algorithm 1). Figure 4 presents the results and the blocks associated with these loops. Figure 5 shows the WPG graph obtained by applying the *WPG Construction*. The WPG graph consists of blocks (nodes) and edges. A block has four attributes: I (the set of loop indices), S (the set of statements in the loop), W (the weights of P.Host and P.Mem), and O (the execution order of the block). An edge specifies the dependence relation between two connected blocks, as defined in Definition 3. Using the *Weight Evaluation*, the P.Host weight and the P.Mem weight in each block can be determined, as shown in Fig. 6. Based on the P.Host weight, the P.Mem weight, and the dependence relations between the blocks, the execution schedule can be generated using *Wavefront Generation* and *Schedule Determination* in Algorithm 2. The seven blocks are scheduled into three wavefronts, as shown in Fig. 7. In the first wavefront, block *b1* is dispatched to P.Host and blocks *b2*, *b3*, and *b6* are dispatched to P.Mem. In the second wavefront, block *b7* is dispatched to P.Host and block *b4* is dispatched to P.Mem. In the third wavefront, only one block *b5* is present, and is dispatched to P.Host because its P.Host weight is less than P.Mem weight. Between two contiguous wavefronts, a synchronization barrier is required. Figure 8 presents those results.

```
// Loop 1

DO I = 1 TO N
   DO J = 1 TO M
S1:    A(I,J) = B(I,J)+C(I,J)
S2:    A(I,J) = A(I-1,J)+A(I+1,J)+C
S3:    X = A(I,J)+2
S4:    A(I,J+1)=X*7
S5:    D(I,J) = 2*D(I,J)+3
S6:    E(I,J) = 2*E(I,J)+2
   ENDDO
ENDDO

// Loop 2

DO J = 1 TO N
   DO I = 1 TO M
S7:    F(I,J) = E(I,J)*F(I,J)
S8:    F(I,J+1) = F(I,J)+5
S9:    G(I,J) = G(I-1,J)*G(I,J-1)
   ENDDO
ENDDO

// Loop 3

DO I = 1 TO N
   DO J = 1 TO M
S10:   Z = A(I,J)+A(I,J-1)
S11:   A(I,J) = Z*C
   ENDDO
ENDDO
```

Fig.3. A simple program with three loops.

```
// Block b1
DO I = 1 TO N
   DO J = 1 TO M
S1:    A(I,J) = B(I,J)+C(I,J)
S2:    A(I,J) = A(I-1,J)+A(I+1,J)+C
S3:    X = A(I,J)+2
S4:    A(I,J+1)=X*7
   ENDDO
ENDDO

// Block b2
DO I = 1 TO N
   DO J = 1 TO M
S5:    D(I,J) = 2*D(I,J)+3
   ENDDO
ENDDO

// Block b3
DO I = 1 TO N
   DO J = 1 TO M
S6:    E(I,J) = 2*E(I,J)+2
   ENDDO
ENDDO

// Block b4
DO J = 1 TO N
   DO I = 1 TO M
S7:    F(I,J) = E(I,J)*F(I,J)
   ENDDO
ENDDO

// Block b5
DO J = 1 TO N
   DO I = 1 TO M
S8:    F(I,J+1) = F(I,J)+5
   ENDDO
ENDDO

// Block b6
DO J = 1 TO N
   DO I = 1 TO M
S9:    G(I,J) = G(I-1,J)*G(I,J-1)
   ENDDO
ENDDO

// Block b7
DO I = 1 TO N
   DO J = 1 TO M
S10:   Z = A(I,J)+A(I,J-1)
S11:   A(I,J) = Z*C
   ENDDO
ENDDO
```

Fig. 4. Resulting program of Fig. 3 after *Statement Splitting*.

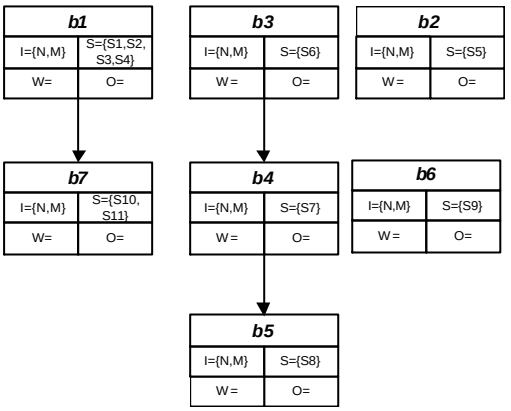

Fig. 5. WPG graph of the program in Fig. 3 after *WPG Construction.*

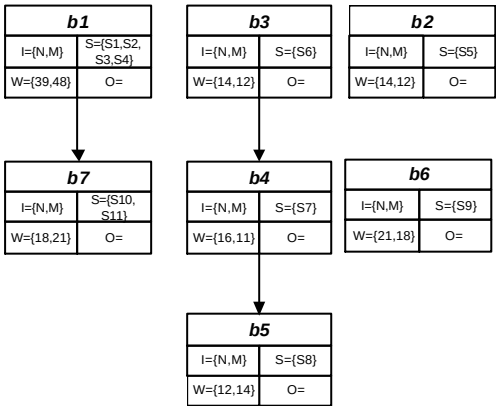

Fig. 6. Resulting WPG graph of Fig. 5 after *Weight Evaluation*.

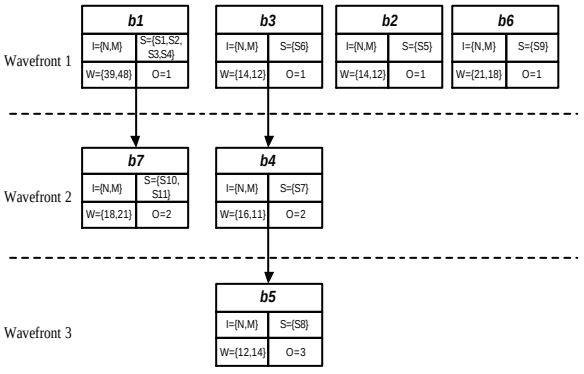

Fig. 7. The WPG graph in Fig. 6 after *Wavefront Generation.*

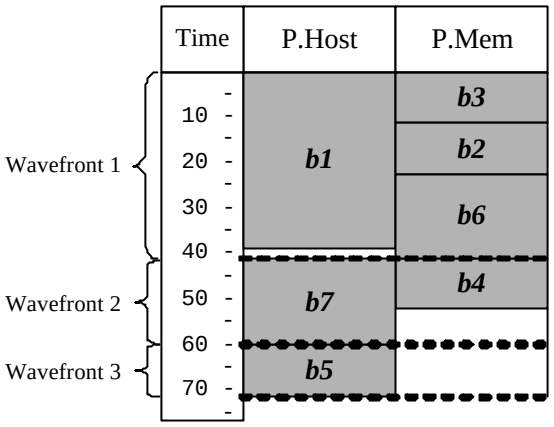

Fig. 8. The execution schedule for the P.Host and P.Mem of Fig. 3.

5. Experimental Results

The code generated by our SAGE system is targeted on the FlexRAM simulator developed by IA-COMA Lab. in UIUC [13]. Derived from MINT [26], this simulator models the environment of a dynamic superscalar multiprocessor and the detailed memory behavior cycle by cycle. Table 1 lists the major architectural parameters (Section 2.1). In this experiment, only one P.Mem processor is spawned to reflect the benefits of the memory processor. The applications evaluated include five benchmarks - *swim* and *tomcatv* from SPEC95, *strmm* from BLAS3, *ep* from the serial version of NAS and *fft* from [23]. This simulator is executed on an SGI Origin200, using a MIPSPro Fortran compiler with the optimizing option "-O2" to compile these five benchmarks. According to the claims made for the MIPSPro compiler, option "-O2" provides several instruction level optimizations, such as scalar replacement, tiling, constant propagation, dead code elimination, and others. Table 4 demonstrates the results of the experiment, in which "P.Host only" denotes that the applications are executed on P.Host alone; "P.Mem only" denotes that the applications are executed on P.Mem alone, and "SAGE opt" indicates that the applications are transformed by SAGE for execution on one P.Host and one P.Mem simultaneously. "Speedup" is obtained by dividing "P.Host only" by "SAGE opt".

Table 4. The execution cycles of five benchmarks.

Benchmark	P.Host only	P.Mem only	SAGE opt	Speed up
swim	228289321	355801581	116669760	1.96
strmm	233969505	356808711	204417723	1.14
tomcatv	380235321	455758516	375200330	1.01
fft	117998621	403954552	101841407	1.15
ep	103044816	250925512	86924945	1.19

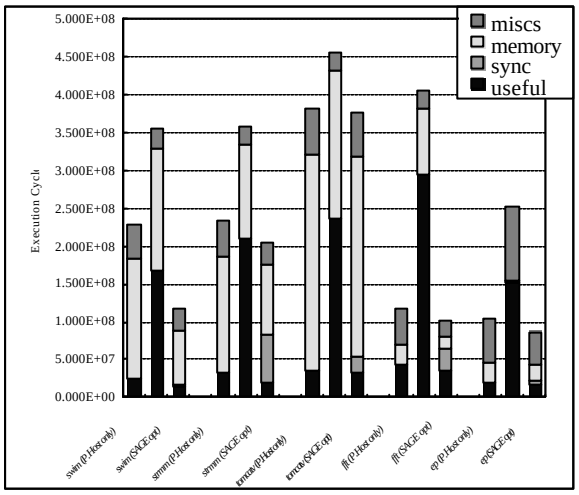

Fig. 9. The execution times of five benchmarks by P.Host only, P.Mem only, and optimized by SAGE.

In Table 4, *swim* has the best speedup because it can be partitioned into many blocks for scheduling to P.Host and P.Mem, according to the characteristics of blocks and processors. Restated, *swim* has more potential parallelism. In contrast, *strmm*, *tomcatv*, *fft*, and *ep* are intrinsically sequential. They can only be partitioned into several large blocks, preventing the generation of load-balance schedules. Therefore, even SAGE can not greatly improve the their performance.

Figure 9 depicts the experimental results with reference to the four major parts - useful (cycles for executing useful instructions), sync (cycles for synchronizing with memory processors), memory (cycles for memory access, including the time required for cache coherence) and misc (cycles for other hazards). Except *fft*, the "useful" parts of the benchmarks in the "P.Host only" mode are smaller than those in the "P.Mem only" mode and the "memory" parts in the "P.Host only" mode are larger than those in the "P.Mem only" mode. This observation explains the fact that the host processor computes more powerfully, while the memory processor has shorter memory access latency, as mentioned above. Additionally, *fft* cannot get benefit from memory access in "P.Mem only" mode because its data are too few, such that all data can be cached in the second level cache of P.Host. On the other hand, the "useful" and "memory" parts in the "SAGE opt" mode are significantly reduced since *swim* can be effectively partitioned and scheduled by SAGE. However, *strmm*, *tomcatv* and *fft* cannot be partitioned into many blocks; hence, synchronization time is required and their execution times are not greatly reduced by SAGE. Notably, synchronization time is required when P.Host and P.Mem execute the program simultaneously, but do not finish the execution at the same time. In the circumstances, one processor must wait for the other. The performance of the *ep* benchmark cannot be improved greatly even if P.Host and P.Mem cooperate to execute this benchmark in "SAGE opt" mode since the memory access time is rather small.

6. Conclusion

This study proposes an automatic source- to-source parallelization system, called SAGE, for a new class of high-performance SoC architecture, Processor-in-Memory, which consists of a host processor and a memory processor. The SAGE system partitions source codes into blocks by statement splitting; estimates the weight (execution time) of each block, and then schedules each block to the most suitable processor for execution. This study refined our earlier work by devising a new scheduling mechanism and integrating a new weight evaluation mechanism. Five real benchmarks, *swim*, *tomcatv*, *strmm*, *ep*, and *fft* were experimentally considered to evaluate the effects of SAGE system. The simulator used here is a PIM architecture that consists of one P.Host and one P.Mem, derived from FlexRAM. The obtained speedups are from 1.01 to 1.96, depending on the characteristics of applications and their potential parallelism. The techniques proposed here can be extended to run on DIVA, EXECUBE and FlexRAM, with several memory chips, each of which has several memory processors.

7. References

[1] J. R. Allen, D. Callahan, K. Kennedy, Automatic decomposition of scientific programs for parallel execution, in: Proc. ACM Symposium on the Principles of Programming Languages (Munich, Germany, Jan. 1987).

[2] R. Barua, W. Lee, S. Amarasinghe, A. Agarwal, Maps: A Compiler-managed Memory System for Raw Machines, in: Proc. 26th Annual International Symposium on Computer Architecture, (May 1999) 4-15.

[3] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, S. Weatherford, Effective Automatic Parallelization with Polaris, International Journal of Parallel Programming (May 1995).

[4] S. L. Chu, T. C. Huang, L. C. Lee, Improving Workload Balance and Code Optimization on Processor-in-Memory Systems , to appear in Journal of Systems and Software (2003).

[5] R. Crisp, Direct Rambus Technology: the New Main Memory Standard, IEEE Micro, (Nov. 1997) 18-28.

[6] D. Elliott, M. Stumm, M. Snelgrove, Computational RAM: The Case for SIMD Computing in Memory, in: Proc. ISCA Workshop on Mixing Logic and DRAM (1997).

[7] M. Hall , P. Kogge , J. Koller , P. Diniz , J. Chame , J. Draper , J. LaCoss , J. Granacki , J. Brockman , A . Srivastava , W. Athas , V. Freeh , J. Shin , J. Park, Mapping Irregular Applications to DIVA , a PIM-Based Data-Intensive Architecture, in: Proc. 1999 Conference on Supercomputing (Jan. 1999).

[8] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, M. Lam, Maximizing Multiprocessor Performance with the SUIF Compiler, IEEE Computer (Dec. 1996).

[9] T. C. Huang and S. L. Chu, SAGE: A New Analysis and Optimization System for FlexRAM Architecture, in: Proc. IMS 2000, Lecture Notes in Computer Science, Vol. 2107, (Springer-Verlag, Berlin, 2001) 160-168.

[10] T. C. Huang and S. L. Chu, A New Analysis Approach for Intelligent Memory Systems, in: Proc. ISCA 16th International Conference on Computers and Their Applications (Seattle, WA, Mar. 2001) 452-457.

[11] W. Huang, Exploiting Application Parallelism Using Advanced Intelligent Memory – The FlexRAM approach, MS Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

[12] D. Judd and K. Yelick, Exploiting On-Chip Memory Bandwidth in the VIRAM Compiler, in: Proc. 2nd Workshop on Intelligent Memory Systems (Cambridge, MA, Nov. 2000).

[13] Y. Kang, W. Huang, S. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, J. Torrellas, FlexRAM: Toward an Advanced Intelligent Memory System, in: Proc. International Conference on Computer Design (Austin, Texas, Oct. 1999).

[14] Y. Kang, An Intelligent Memory for Data-Parallel Applications, Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

[15] K. Keeton, R Arpaci-Dusseau, D.A.Patterson, IRAM and SmartSIMM: Overcoming the I/O Bus Bottleneck, in: Proc. ISCA Workshop on Mixing Logic and DRAM (1997).

[16] K. Kennedy and K. S. McKinley, Loop Distribution with Arbitrary Control Flow, in: Proc. Supercomputing '90 (New York, Nov. 1990).

[17] P. Kogge, The EXECUBE Approach to Massively Parallel Processing, in: Proc. International Conference on Parallel Processing (August 1994).

[18] D. J. Kuck, A Survey of Parallel Machine Organization and Programming, ACM Comput. Surv. Vol 9, 1 (Mar. 1977) 29-59.

[19] D. Landis, L. Roth, P. Hulina, L. Coraor, S. Deno, Evaluation of Computing in Memory Architectures for Digital Image Processing Applications, in: Proc. International Conference on Computer Design (1999) 146-151.

[20] C.A. Moritz, M. Frank, S. Amarasinghe, FlexCache: A Framework for flexible Compiler Generated Data Caching, in: Proc. 2nd Workshop on Intelligent Memory Systems (Cambridge, MA, Nov. 2000).

[21] M. Oskin, F. Chong, T. Sherwood, Active Pages: A Computation Model for Intelligent Memory, in: Proc. 25th Annual International

Symposium on Computer Architecture (June 1998) 192-203.

[22] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Tomas, K. Yelick, A Case for Intelligent DRAM, IEEE Micro (Mar./Apr. 1997) 33-44.

[23] W. H. Press, S.A. Teukolsky, W. T. Vetterling, B. P. Flannery, Numerical Recipes in Fortran 77 (Cambridge University Press, 1992).

[24] B. Reistad and D. K. Gifford, Static Dependent Costs for Estimating Execution Time, in: Proc. ACM Conference on LISP and Functional Programming (1994) 65-78.

[25] K. Snip, D.G. Elliott, M. Margala, N. G. Durdle, Using Computational RAM for Volume Rendering, in: Proc. 13th Annual IEEE International Conference on ASIC/SOC (2000) 253 –257

[26] J. Veenstra and R. Fowler, MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors, in: Proc. MAS-COTS' 94 (Jan. 1994) 201-207.

[27] V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, X. Ji, A Adapting Cache Line Size to Application Behavior, in: Proc. Supercomputing' 99 (Jun, 1999).

[28] K.Y. Wang, Precise Compile-Cime Performance Prediction for Superscalar-Based Computers, in: Proc. ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (1994) 73 – 84.