Toward to Utilize the Heterogeneous Multiple Processors of the Chip Multiprocessor Architecture

Slo-Li Chu

Department of Information and Computer Engineering,
Chung Yuan Christian University, Chung-Li, Taiwan, R.O.C.
slchu@cycu.edu.tw

Abstract. Continuous improvements in semiconductor fabrication density are supporting new classes of Chip Multiprocessor (CMP) architectures that combine extensive processing logic/processor with high-density memory in a single chip. One of the architecture, called Processor-in-Memory (PIM) can support high-performance computing by combining various processors in a single system. Therefore, a new strategy is developed to identify their capabilities and dispatch the most appropriate jobs to them in order to exploit them fully. This paper presents a novel scheduling mechanism, called Swing Scheduling to fully utilize all of the heterogeneous processors in the PIM architecture. Integrated with our Octans system, this mechanism can decompose the original program into blocks and can produce a feasible execution schedule for the host and memory processors, even for other CMP architectures. The experimental results for real benchmarks are also proposed.

Keywords: Chip Multiprocessor (CMP), Processor-in-Memory, Swing Scheduling, Octans.

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 or Chip Multiprocessor [13] strategies, which integrates the processors and memory on a single chip. The rapid growth in silicon fabrication density has made this strategy possible. Accordingly, many researchers have addressed integrating computing logic/processing units and high density DRAM on a single die [5][7][8][9][10][12][13]. Such architectures are also called 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 [12].

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 has 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 [4][6] have concentrated on spawning as many processors as possible to increase speedup, rather than on the capability difference between the host and memory processors. However, such an approach does not exploit the real advantages of PIM architectures. This study integrates our Octans system that integrates statement splitting, weight evaluation and a scheduling mechanism. The original scheduling [2] mechanism is improved to generate a superior execution schedule to fully utilize all heterogeneous processors in the PIM architecture, using our new Swing Scheduling mechanism. A weight evaluation mechanism is established to obtain a more precise estimate of execution time, called weight. The Octans system can automatically analyze the source program, generate a Weighted 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 PIM architectures. Section 3 describes our Octans system and the Swing Scheduling algorithms. Section 4 presents experimental results. Conclusions are finally drawn in Section 5.

2 The Processor-in-Memory Architecture

Fig. 1 depicts the organization of the PIM architecture evaluated in this study. It contains an off-the-shelf processor, P.Host, and four PIM chips. The PIM chip integrates one memory processor, P.Mem, with 64 Mbytes of DRAM. The techniques presented in this paper is suitable for the configuration of one P.Host and multiple P.Mems, and can be extended to support multiple P.Hosts.

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. There is a two-level cache in P.Host and a one-level cache in P.Mem. P.Mem has lower memory access latency than 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 conform to 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. The private interconnection network of the PIM chips is also provided.

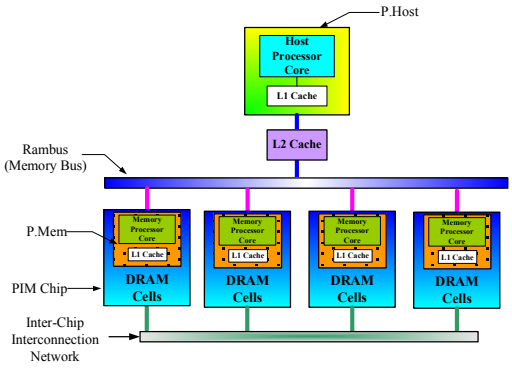

Fig. 1. Organization of the PIM architecture.

Table 1. 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
		P.Mem Mem RT: 50. 5 ns
Integer unit num: 6	Integer unit num: 2	Bus Width: 16 B
Floating unit num: 4	Floating unit num: 2	Mem_Data_Transfer: 16
FLC_Type: WT	FLC_Type: WT	Mem_Row_Width: 4K
FLC_Size: 32 KB	FLC_Size: 16 KB	
FLC_Line: 64 B	FLC_Line: 32 B	
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.*

3 The Octans System

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 Octans system.

Octans 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 Weighted 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 Octans and other parallelizing systems is that it uses a statement rather than an 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. Fig. 2 illustrates the organization of the Octans system.

3.1 Statement Splitting and WPG Construction

Statement Splitting splits the dependence graph of the given program by the extended node partition mechanism as introduced in [2]. It divides the original program into several small loops within the minimal statements. The detailed mechanisms can be

found in literature [2]. Then WPG Construction constructs the Weighted Partition Dependence Graph (WPG), to be used in the subsequent stages of Weight Evaluation, Wavefront Generation and Schedule Determination.

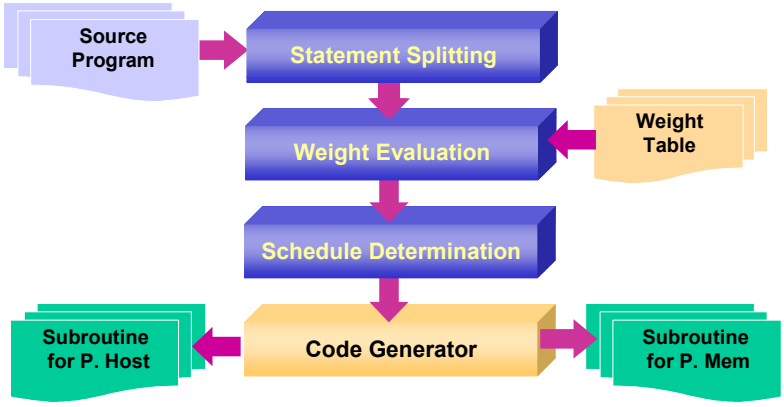

Fig. 2. The sequence of compiling stages in Octans.

3.2 Weight Evaluation

Two approaches to evaluating weight can be taken. One is to predict the execution time of programs by profiling the dominant parts. The other considers the operations in a statement and estimates the program execution time by looking up an operation weight table. The former method called code profiling may be more accurate, but the predicted result cannot be reused; the latter called code analysis 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. It integrates these two approaches together by analyzing code and searching weight table first to estimate the weight of a block. If the block contains unknown operations, the patch (profiling) mechanism is then activated to evaluate the weights of unknown operations. The obtained operation weights are added into the weight table for next look-up. For a detailed description of this scheme, please refer to [2].

3.3 The Swing Scheduling Mechanism

Here we propose the Swing Scheduling mechanism to achieve a good schedule for utilizing all of the memory processors in PIM architecture. At first, the redundancy and synchronization between processors are critical factors that affect the performance of job scheduling for multiprocessor platforms. A critical path mechanism is used to minimize the frequency of synchronization. Then the WPG is then partitioned into several Sections according to the nodes on the critical path and the dependence relations between these nodes. In a Section, the blocks are partitioned into several Inner Wavefronts in the following stages. Finally, the execution schedule

for all P.Host and P.Mems is obtained. If the number of occupied memory processors exceeds the maximum number of processors in the PIM configuration, then the execution schedule will be modified accordingly. Algorithm 1 presents the main steps of this scheduling mechanism.

Algorithm 1. (Swing Scheduling)

[Input]

WPG=(P,E): original weighted partition dependence graph after weight is determined.

[Output]

An critical path execution order schedule CPS, where $CPS = \{CPS_1, CPS_2, ..., CPS_i\}$. $CPS_i = \{CP_i, IWF_i\}$ where CP_i = {Processor(b_a)} where processor is PH or PM . $IWF_i = \{PH(b_a), PM_1(b_b), PM_2(b_c), ...\}$ means that in *Inner Wavefront i*, $PH(b_a)$ means that block b_a will be assigned to P.Host, $PM_1(b_b)$ means that blocks b_b will be assigned to $P.Mem_1$, $PM_2(b_c)$ means that blocks b_c will be assigned to $P.Mem_2$.

[Intermediate]

W: a working set of nodes ready to be visited.
EO_temp: a working set for execution order scheduling.
iwf_temp: a working set for *Inner Wavefront* scheduling.
max_EO: the maximum number of execution order.
min_pred_O(b_i): the minimum execution order for all b_i's predecessor blocks.
max_pred_O(b_i):the maximum execution order for all b_i's predecessor blocks.
min_succ_RO(b_i):the minimum execution order for all b_i's successor blocks.
max_succ_RO(b_i):the maximum execution order for all b_i's successor blocks.
PHW(b_i): the weight of b_i for P.Host.
PMW(b_i): the weight of b_i for P.Mem.
$Rank_u(b_i)$: the trace up value of b_i used for finding CP
$Rank_d(b_i)$: the trace down value of b_i used for finding CP

[Method]

Step 1: Call "*Initialization()"* to initialize the algorithm and determine the weights of each block.
Step 2: Call "*$Rank_d$_Exec_Order_Det()*" to determine the $Rank_d$ and Execution order of each block.
Step 3: Call "$Rank_u$_Det()" to determine the $Rank_u$ of each block.
Step 4: Call "*Critical_Path_Det()*" to determine the blocks which is belong to the Critical Path.
Step 5: Call "*Critical_Path_Block_Sch()*" to find out the *Section* and schedule the Critical Path Block in each *Section* to the a suitable processor.
Step 6: Call "*Inner_Wavefront_Sch()*" to partition the blocks which is belong to the same *Section* into several *Inner Wavefront* and schedule the blocks in the same *Inner Wavefront* to the suitable processors.
Step 7: Call "*Generate_Schedule()*" to generate the execution schedule, CPS.
Step 8: If the occupied processor number is larger than the maximum processor number, call "Modify_schedule()" to modify the original execution schedule to fit the processor number, else Stop the algorithm.

The algorithm includes eight major steps. In Step 1, the algorithm calls *"Initialization()"* to initiate the necessary variables and determine the P.Host and P.Mem weights of each blocks determined by the weight evaluation mechanism.

Swing algorithm adopts the critical path method to partition WPG into *Sections*. Therefore, the critical path and the blocks on the critical path must be determined. Then the attributes, $rand_u$ and $rank_d$, of block b_i in WPG are defined by the following equations.

$$rank_u(b_i) = PMW(b_i) + \max_{b_j \in succ(b_i)}(rank_u(b_j))$$

$$rank_d(b_i) = \max_{b_j \in pred(b_i)}\{rank_d(b_j) + PMW(b_j)\}$$

Here, $succ(b_i)$ and $pred(b_i)$ represent all of the successors and predecessors of b_i, respectively. The *critical path* is defined as the following equation.

A block b_i is on the *critical path*, if and only if $rank_u(b_i) + rank_d(b_i) = rank_u(b_s)$, where b_s is the start block of the WPG, and b_i is called the *critical path block.*

According to the above definitions, the *critical path* and the *critical path block* can be determined from Step 2 to Step 4. Step 2 calls *"Rank$_d$_Exec_Order_Det()"* to determine $rank_d$ and the execution order of each block. Step 3 calls *"Rank$_u$_Det()"* to determine $rank_u$ of each block. Then, the algorithm calls "*Critical_Path_Det()*" to determine which blocks are *critical path blocks* in Step 4.

```
Subroutine: Critical_Path_Det()
    CP = (rank_d(b_s)),where b_s is the start block of WPG
    CP_num_sec=0
    for i=1 to max_EO do
        store all of b_i whose O_i =i in EO_temp
        for each block b_i ∈ EO_temp do
            if (rank_d(b_i)+rank_u(b_i))=CP then
                CP_num_sec=CP_num_sec+1
                CP_O(CP_num_sec )= O(b_i)
                CP_temp(CP_num_sec )= b_i
        end for
end for
```

```
Subroutine: Rank_u_Det()
    W=P-{b_e},where b_e is the end block of the WPG
    RO(b_e)=1
    done = False
        while done = False AND W≠φ do
            done=True
            for each b_i ∈W do
                if min_succ_RO(b_i)=0  then
                    done=False
                else
                    rank_u(b_i) = PMW(b_i) + max_{b_j∈succ(b_i)}(rank_u(b_j))
                    RO_i= max_succ_RO(b_i)+1
                    W=W-{ b_i }
                end if
            end for
end while
```

```
Subroutine: Critical_Path_Det()
    CP = (rank_d(b_s)),where b_s is the start block of WPG
```

```
    CP_num_sec=0
    for i=1 to max_EO do
    store all of b_i whose O_i=i in EO_temp
    for each block b_i ∈ EO_temp do
       if (rank_d(b_i)+rank_u(b_i))=CP then
          CP_num_sec=CP_num_sec+1
          CP_O(CP_num_sec )= O(b_i)
          CP_temp(CP_num_sec )= b_i
    end for
end for
```

Fig. 3 illustrates the WPG of the synthetic program, which is processing in stages stated above. In this WPG, the shadow blocks are on the critical path. When the critical path is determined in Step 5, "*Critical_Path_Block_Sch()*" is called to partition all blocks in the WPG into several *Sections*. Fig. 4 illustrates the result of the given WPG, which is partitioned into five *Section*s, *Section*1:{b1}, *Section* 2: {b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14}, *Section* 3:{b15}, *Section* 4: {b16, b17, b18, b19, b20, b21, b22, b23, b24, b25, b26, b27, b28} and *Section* 5:{b29}. The execution order of *Sections* is governed by their dependence relations. After the critical path block is identified, the remaining blocks are partitioned into several *Inner Wavefronts* according to the order of execution and the dependence relations. In Fig. 4, *Section* 2 of the WPG is used to explain how blocks are scheduled in a *Section*. Since b2 is the block on critical path in *Section* 2, "*Critical_Path_Block_Sch()*" is firstly used to schedule b2 to reduce the waiting and synchronization frequencies. The remaining blocks are partitioned into three wavefronts according to the O_i of each block, by calling *"Inner_Wavefront_Sch()"* in Step 6. Finally, iw1={b3, b4, b5, b6}, iw2={b7, b8, b9}, iw3={b10, b11, b12, b13} are determined.

```
Subroutine: Critical_Path_Block_Sch()
    i=1, k=0
    while k ≤CP_num_sec do
       k=CP_O(i)
       if PHW(CP_temp( i ))- PMW(CP_temp( i ))< 0 then
          CP_k={PH(CP_temp(i))}
          PH_Used=true
          PM1_Used=false
       else
          CP_k={PM_1(CP_temp(i))}
          PH_Used=false
          PM1_Used=true
       end if
       i=i+1
    end while
```

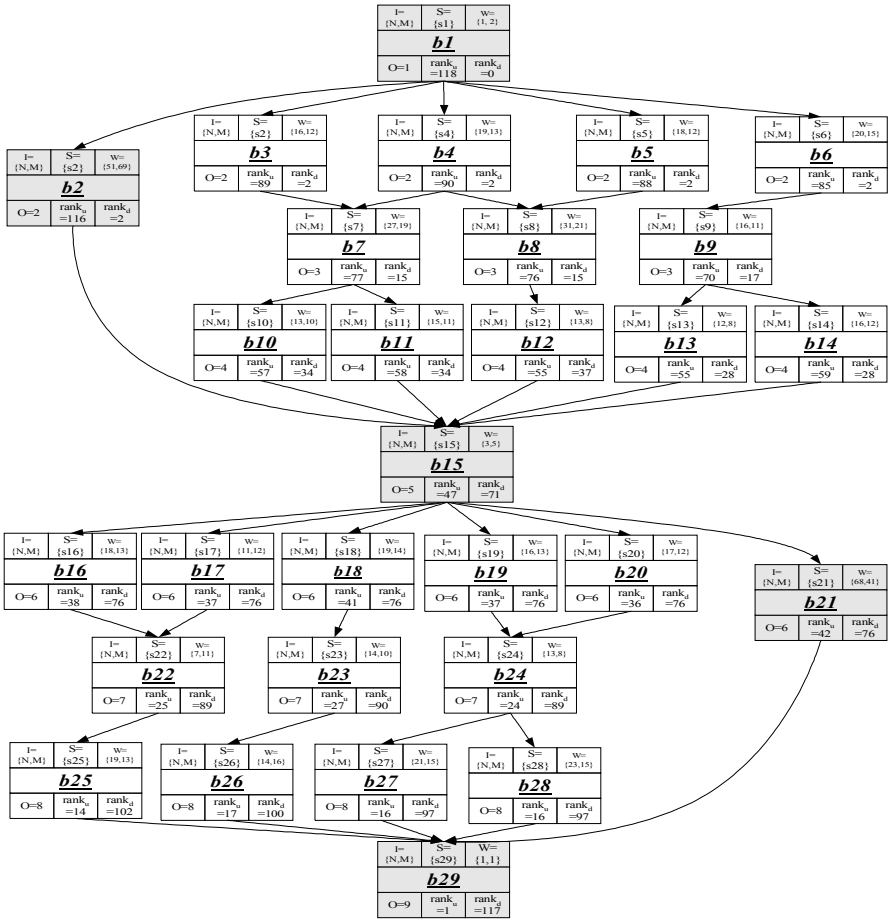

Fig. 3. WPG of a synthetic example.

Section 2={b2,b3,b4,b5,b6,b7,b8,b9,,b10,b11,b12,b13}
Critical path ={b2}

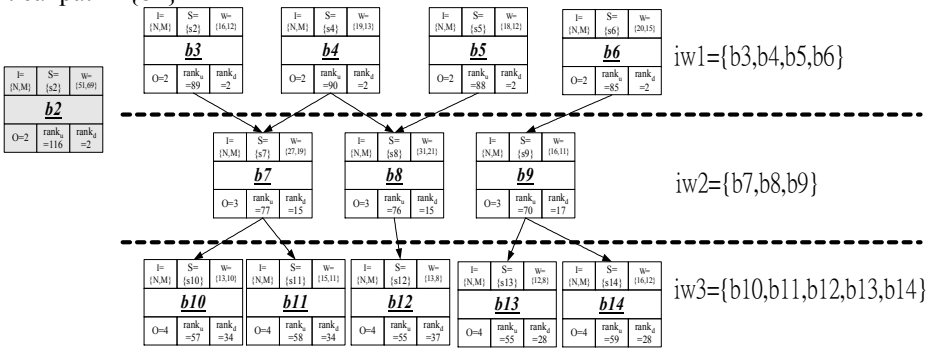

Fig. 4. Scheduled WPG of *Section* 2.

$CPS = \{CPS1 , CPS2 , CPS3 , CPS4 , CPS5\}$
$=\{\{CP1 , IWF1\}, \{CP2 , IWF2\}, \{CP3 , IWF3\}, \{CP4 , IWF4\}, \{CP5 , IWF5\}\}$

CPS_1 : **/*Section 1*/**
$CP_1=\{PH(b1)\}$,
$IWF_1=\{\phi\}$

CPS_2 : **/*Section 2*/**
$CP_2=\{PH(b2)\}$,
$IWF_2=\{iwf1, iwf2, iwf3\} =\{\{PM_1(b3), PM_2(b4), PM_3(b5), PM_4(b6)\}, \{PM_1(b7), PM_2(b8), PM_3(b9)\}, \{PM_1(b10), PM_2(b11), PM_3(b12), PM_4(b13), PM_5(b14)\}\}$

CPS_3 : **/*Section 3*/**
$CP_3=\{PH(b15)\}$,
$IWF_3=\{\phi\}$

CPS_4 : **/*Section 4*/**
$CP_4=\{PM_1(b21)\}$,
$IWF_2=\{iwf1, iwf2, iwf3\} =\{\{PH(b16), PM_1(b17), PM_2(b18), PM_3(b19), PM_4(b20)\}, \{PH(b22), PM_1(b23), PM_2(b24)\}, \{PH(b25), PM_1(b26), PM_2(b27), PM_3(b28)\}\}$

CPS_5 : **/*Section 5*/**
$CP_5=\{b29\}, IWF_5=\{\phi\}$

Fig. 5. Output of the Swing scheduling algorithm.

In Step 7, the execution schedule is generated by "*Generate_Schedule()*", as shown in Fig. 5. and Fig. 6 shows the graphical execution schedule. The shaded blocks the Fig. 5 represent the execution latency. The blank blocks indicate that the processor is waiting for other processors to synchronize. The bold and dotted lines determine the point of synchronization of *Section* and *Inner Wavefront* respectively.

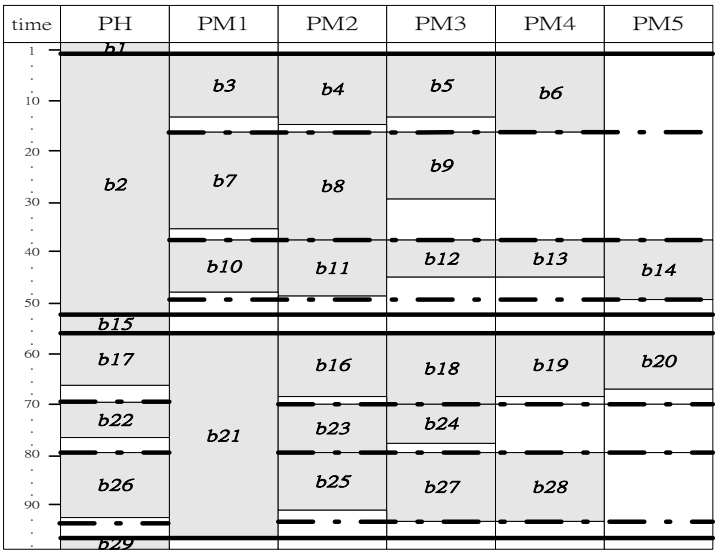

Fig. 6. Graphical execution schedule of the given example.

Sometimes, the execution schedule may occupy more processors than are present in the architectural configuration. Therefore, Step 8 calls *"Modify_schedule()"* to modify the execution schedule as necessary.

5 Experimental Results

The code generated by our Octans system is targeted on our PIM simulator that is derived from the FlexRAM simulator developed by the IA-COMA Lab. at UIUC [13]. Table 1 lists the major architectural parameters. In this experiment, the configuration of one P.Host with many P.Mem processors is modeled to reflect the benefits of the multiple memory processors.

This experiment utilizes multiple P.Mem processors in the PIM architecture to improve performance. The evaluated applications include five benchmarks: *cg* is from the serial version of NAS; *swim* is from SPEC95; *strsm* is from BLAS3; *TISI* is from Perfect Benchmark, and *fft* is from [45].

Table 2 and Fig. 7 summarize the experimental results. “Standard” denotes that the application is executed in P.Host alone. This experiment concerns a general situation of a uniprocessor system, and is used to compare speedup. "1-P.Mem” implies that the application is transformed and scheduled by the simplified Swing Scheduling for the one-P.Host and one-P.Mem configuration of the PIM architecture. “n-P.Mem” implies that the application is transformed and scheduled by Swing Scheduling mechanism for the one P.Host and multiple P.Mem configuration of the PIM architecture.

Table 2 and Fig. 7 indicate that *swim* and *cg* have quite a good speedup when the Swing Scheduling mechanism is employed because these programs contain many memory references and few dependence relations. Therefore, the parallelism and memory access performance can be improved by using more memory processors. Applying the 1-P.Mem scheduling mechanism can also yield improvements. *strsm* exhibits an extremely high parallelism but a rather few memory access, so the Swing Scheduling mechanism is more suitably adopted than the 1-P.Mem scheduling mechanism. *TISI* cannot generate speedup when the 1-P.Mem scheduling mechanism is applied, since it is a typical CPU bounded program, and involves many dependencies. The Swing Scheduling mechanism can be suitably used to increase speedup. Finally, in *fft*, the program is somewhat computation-intensive and sequential, and therefore only a little speedup can be improved after the 1-P.Mem scheduling mechanism is applied. However, an additional overhead is generated when the Swing Scheduling mechanism is applied. Accordingly, 1-P.Mem and Swing scheduling mechanisms are suitable for different situations. Choosing the 1-P.Mem or Swing scheduling mechanism more heuristically in the scheduling stage of the Octans system will improve performance.

Table 2. Execution cycles of five benchmarks.

Bench-mark	Standard	1-P.Mem Scheduling	n-P.Mem Scheduling	Speedup		
				1-P.Mem Scheduling	n-P.Mem Scheduling	n(Occupied P.Mem)
swim	228289321	116669760	52168027	1.96	4.38	6
cg	91111840	51230772	32124287	1.78	2.84	4
strsm	703966766	489967053	187989176	1.44	3.74	5
TISI	133644087	173503404	91098174	0.77	1.47	2
fft	117998621	101841407	110399171	1.16	1.07	2

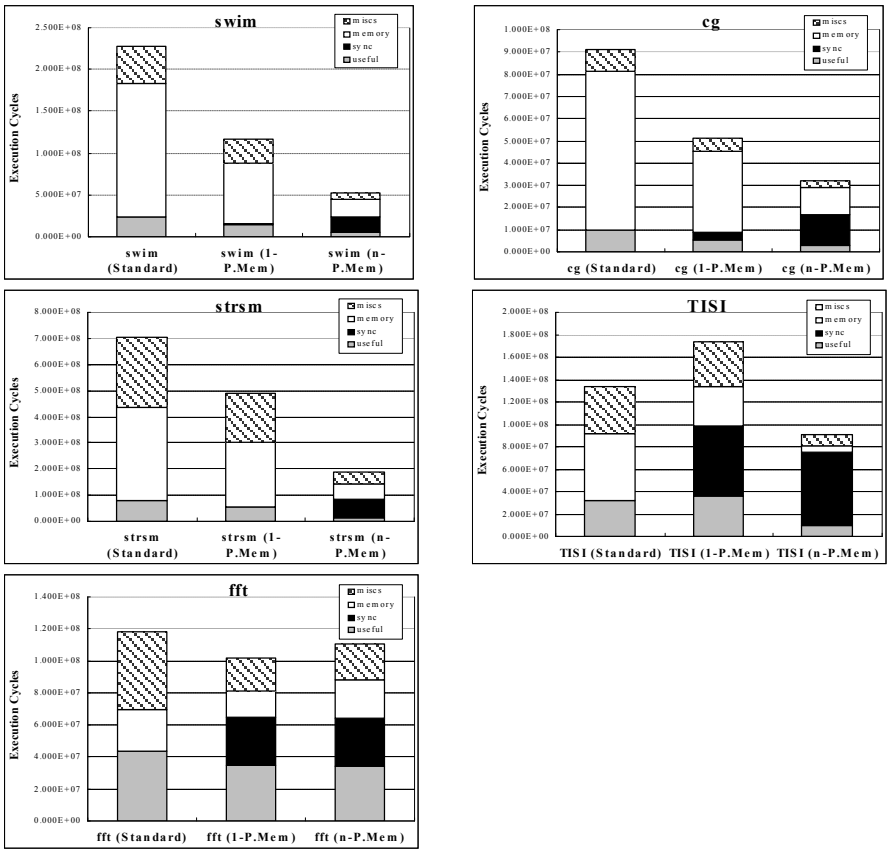

Fig. 7. Execution times of five benchmarks obtained by Standard, 1-P.Mem and n-P.Mem settings.

6 Conclusions

This study proposes a new scheduling mechanism, called Swing Scheduling, with Octans system for a new class of high-performance chip multiprocessor architectures, Processor-in-Memory, which consists of a host processor and many memory processors. The Octans system partitions source code 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. Five real benchmarks, swim, TISI, strsm, cg, and fft were experimentally considered to evaluate the effects of the Swing Scheduling. In the experiment, the performance was improved by a factor of up to 4.38 while using up to six P.Mems and one P.Host. The authors believe that the techniques proposed here can be extended to run on DIVA, EXECUBE, FlexRAM, and other high-performance chip multiprocessor architectures by slightly modifying the code generator of the Octans system.

Acknowledgement

This work is supported in part by the National Science Council of Republic of China, Taiwan under Grant NSC 96-2221-E-033 -019-

References

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

[2] Chu, S. L.: PSS: a novel statement scheduling mechanism for a high-performance SoC architecture. In Proceedings of Tenth International Conference on Parallel and Distributed Systems, Jul. (2004). pp. 690-697.

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

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

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

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

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

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

[9] Oskin, M., Chong, F. T., and Sherwood, T.: Active Page: A Computation Model for Intelligent Memory. Computer Architecture. In Proceedings of the 25th Annual International Symposium on Computer Architecture, (1998), pp. 192-203.

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

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

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

[13] Swanson, S., Michelson, K., Schwerin, A. and Oskin, M.: WaveScalar. MICRO-36, Dec. (2003).

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

[15] Wang, K. Y.: Precise Compile-Time Performance Prediction for Superscalar-Based Computers, In Proceedings of ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, (1994), pp. 73 – 84