Critical Block Scheduling: a Thread-Level Parallelizing Mechanism for a Heterogeneous 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. Processor-in-Memory (PIM) architectures are developed for high-performance computing by integrating processing units with memory blocks into a single chip to reduce the performance gap between the processor and the memory. The PIM architecture combines heterogeneous processors in a single system. These processors are characterized by their computation and memory-access capabilities. Therefore, a novel mechanism must be developed to identify their capabilities and dispatch the appropriate tasks to these heterogeneous processing elements. Accordingly, this paper presents a novel parallelizing mechanism, called Critical Block Scheduling to fully utilize all of the heterogeneous processors in the PIM architecture. Integrated with our thread-level parallelizing system, Octans, this mechanism decomposes the original program into blocks, produces corresponding dependence graph, creates a feasible execution schedule, and generates corresponding threads for the host and memory processors. The proposed Critical Block Scheduling not only can parallelize programs for PIM architectures but also can apply on other Multi-Processor System-on-Chip (MPSoC) and Chip Multiprocessor (CMP) architectures which consist of multiple heterogeneous processors. The experimental results of real benchmarks are also discussed.

Keywords: Chip Multiprocessor (CMP), Processor-in-Memory, Critical Block 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. This gap can be reduced using the System-on-a-Chip or Chip Multiprocessor [18] strategies, which integrates the various 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 [9][11][12][14][15][17][18]. Such architectures are also called Processor-in-Memory (PIM), or Intelligent RAM (IRAM).

Integrating DRAM and computing elements on a single chip generates PIM architecture with several desirable characteristics. First, the processors are heterogeneous for their purpose. Second, instead of traditional off-chip communication, the on-chip communication between processor-to-processor and processor-to-memory are very wide and fast. Third, eliminating off-chip drivers reduces the power consumption and latency [17].

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.

Since the mechanisms of partitioning and scheduling for heterogeneous multi-computers are classical NP-Hard problems, many researches propose their mechanisms for distributed-memory parallel computers. Opportunistic Load Balancing algorithm assigns each task, in arbitrary order, to the next available machine, regardless of the task's expected execution time on that machine [2]. Min-min algorithm minimizes the completion time for each task is computed for all machines. The newly mapped task is removed, and the process repeated until all tasks are mapped [2]. These methods are focus on how to reduce the communication cost of the parallel program. However, in PIM architecture, the communication cost is not the most significant factor of overall performance. Hence we veer to thread-level parallelizing mechanisms. Cintr et. al. [6] present a architectural support thread-level parallelization framework, which can obtain more potential parallelism by their speculative thread-level parallelizing mechanism with hardware support, especially for the modified CC-NUMA architecture. Arora et. al. [3], Zhou et al. [21], and Agrawal et. al. [1] propose their mechanisms to dynamically schedule the threads in the thread queue to reduce memory access cost and improve cache locality. These mechanisms improve the capabilities of thread scheduler of the targeted operating system, but can not apply on parallelizing compiler for static scheduling. Llosa, et. al. [13] propose a software pipelining mechanism, called Swing Modulo Scheduling (SMS), to partition iteration spaces of loops according to their dependence graph. This algorithm provides iteration-based mechanism that can improve the potential parallelism of the loops and reduce the usage of registers. It is also adopted in GNU Compiler Collection (GCC) Version 4.0. However SMS focuses on scheduling iterations of given loops but not restructure whole program. It isn't suitable for parallelizing program and generating corresponding threads for different heterogeneous processors. Therefore we have to consider other mid-grained approach instead of traditional fine-grained mechanism based on iteration analysis.

From the aspect of compilation for PIM architectures, previous approaches [8] [10] concentrate on instruction-level parallelization and loop vectorization to increase speedup, rather than on the figure out the capability difference between the host and memory processors. However, such approaches do not exploit the real advantages of PIM architectures. Accordingly we design a thread-level parallelization system,

Octans, which integrates statement splitting, weight evaluation and a scheduling mechanism. The original PSS scheduling [5] mechanism focuses on a simplified configuration of PIM architecture that only consists one-P.Host and one-P.Mem processors. Since PSS scheduling can not deal with multiple P.Mem processors and fully utilizes all heterogeneous computing resources, we design a new mechanism, Critical Block Scheduling, to generate a superior execution schedule to fully utilize all heterogeneous processors in the PIM architecture. A weight evaluation mechanism is established to collect characteristics of varied and estimate a precise execution time then generate a normalized value, called weight. The Octans system can automatically analyze the sequential program, partition program into several blocks, determine the weights of each block, produce a good executing schedule, and finally generate parallel threads for execution on the host and memory processors accordingly.

The rest of this paper is organized as follows: Section 2 introduces the PIM architecture. Section 3 describes our Octans system and the Critical Block 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 are 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 conform to a memory standard that involves additional power and ground signals to support on-chip processing. One such standard is Rambus [7], so the PIM chip is designed with a Rambus-compatible interface. The private interconnection network of the PIM chips is also provided.

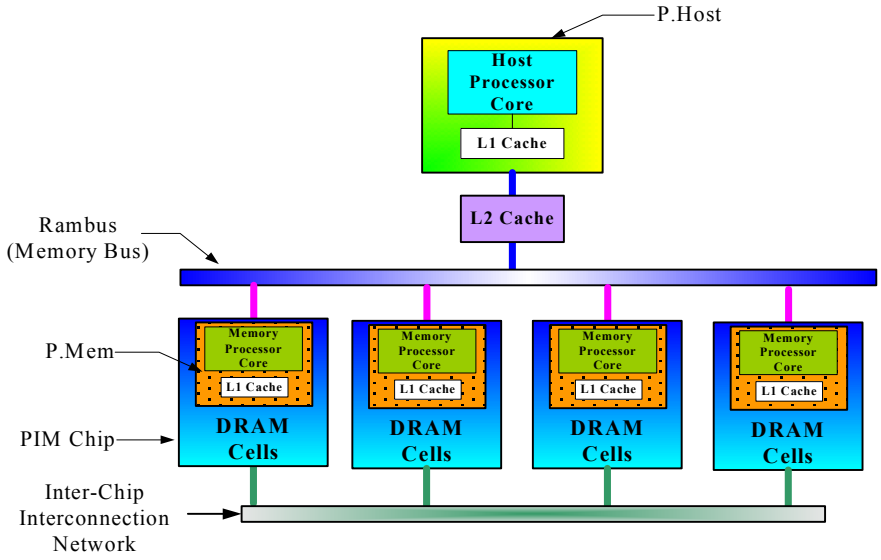

Fig. 1. Organization of the PIM architecture.

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
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	
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, which 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 [5]. 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.

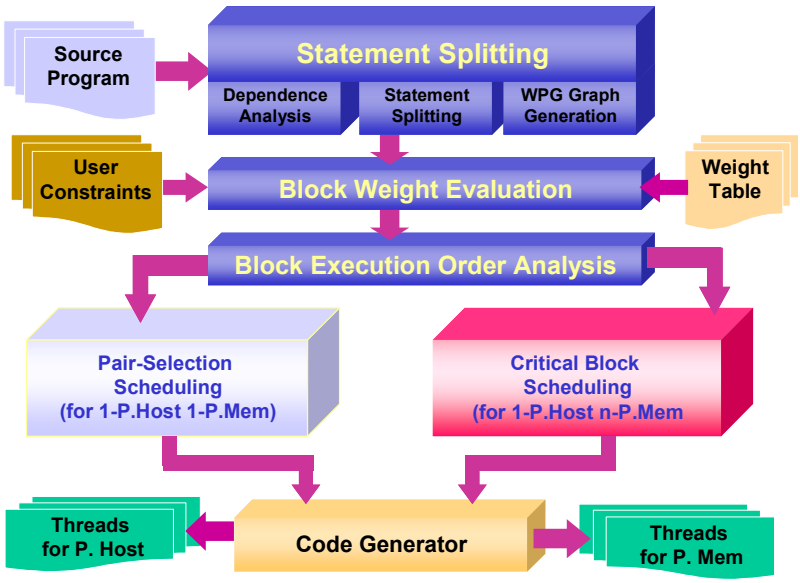

Fig. 2. The sequence of compiling stages in Octans.

3.1 Statement Splitting and WPG Construction

Statement Splitting splits the dependence graph by Node Partitioning as introduced in [5]. 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 Notation)

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_k, $1 \le k \le d$, is a body statement which may be an assignment statement or another loop. ■

Definition 2 (Node Partition $\prod$)

For a given loop L on the dependence graph G, we define a node partition Π for the statements set $\{ s_1 , s_2 , \ldots . s_d\}$ in such a way that s_k and s_l, $1 \le k \le d$, $1 \le l \le d$, $k \ne l$, are in the same block (cell) π_i of the partition Π 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 $i \ne j$:

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) π_i in the partition Π if and only if there is a directed dependence cycle among the statements. Two blocks have a true/anti/output dependence if and only if two statements, one in each block, exist a true/anti/output dependence.

Definition 3 (Weighted Partition Dependence Graph)

Given a node partition Π defined in Definition 2, we define a weighted partition dependence graph *WPG(B,E)* as follows with *B* denoting the set of nodes and *E* denoting the set of edges. For each $\pi_i \in \Pi$, there is a corresponding node b_i (I_i , S_i , W_i , O_i) $\in$ B, where I_i denotes the set of loop indices in block π_i; S_i represents the set of statements in block π_i; W_i is the weight of block π_i in the form of W_i *(PH,PM)* with *PH* and *PM* being the weights (i.e., the expected execution time) for the P.Host and P.Mem, respectively; and O_i is the execution order for block π_i. There is an edge $e_{ij} \in E$ 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_n) (s_1, s_2, …. s_d)
Step 1: Construct dependence Graph G by analyzing subscript expressions and index pattern by using Polaris [4].
Step 2: Establish a node partition Π on G as defined in Definition 2. If there are large blocks caused by control dependence relations, convert control dependence into data dependence first [5], 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.

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 [20]. 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 [5]

3.3 The Critical Block Scheduling Mechanism

Here we propose the Critical Block Scheduling mechanism to achieve an optimal 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 task scheduling for multiprocessor platforms. A critical block mechanism is used to minimize the frequency of synchronization. Then the WPG is then partitioned into several Sections according to the critical blocks 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 2 presents the main steps of this scheduling mechanism.

Algorithm 2. (Critical Block Scheduling Algorithm)

[Input]

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

[Output]

An critical block execution 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: For each block of the WPG, initializes the execution order, obtains the weights of P.Host and P.Mem by using the weight evaluation mechanism.

Step 2: Travel down all blocks of the WPG to determine its $rank_d$ which is the maximal $rank_d$ of the parent blocks, add itself P.Mem weight and increase its execution order according to the maximal execution order of its parent blocks.

Step 3: Travel up all block to determine the $rank_u$ by current block's P.Mem weight plus the max of children block's $rank_u$.

Step 4: Travel all block find out the critical block that $rank_u$ + $rank_d$ equal to the $rank_d$ of the starting block, and then append the block into CP_temp and its order into CP_O.

Step 5: In CP_temp, when a critical block's PHW is less than PMW, assign it to PH, otherwise assign it to PM_1. Append the block into CP_k, where k is CP_O of the block.

Step 6: Split all block to subset by CP_O, the subset doesn't include the critical block, and then perform each subset by follow step.

6.1 Split subset to new subset iwf_temp by order number.

6.2 Check the PH_Used and PM1_Used between CP_O for each iwf_temp.

6.3 Sort iwf_temp in decreasing order by the PMW.

6.4 If the PH_Used of iwf_temp is false then find the minimal PHW block to set PH tag.

6.5 Other block of iwf_temp set PM_k and append to IWF_i.

Step 7: Append CP_i and IWF_i to CPS_i set, and then append all CPS_i to CPS set to generate the execution schedule.

Step 8: Perform each IWF_i by follow steps to modify the execution schedule to fit the limitation of PM number.

8.1 Sort IWF in decreasing order by the block's weight.

8.2 If the PH_Used of IWF is false then find the minimal load of PH + PHW and set it to PH and add the PMW of block to PH load.

8.3 Find the PM with minimal load then reassign the block to it.

8.4 Repeat Step 8.3 until all blocks of IWF is done.

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

This algorithm figures out the critical nodes to partition WPG into *Sections*, so the critical blocks 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 block* is defined as the following equation.

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

According to the above definitions, the *critical block* can be determined by Step 2 to Step 4. Step 2 determines the $rank_d$ and the execution order of each block. In Fig.3 the $rand_u$ of b1 is zero and PWM(b1) is 2, that we can determine the $rank_d$ of b2…b6 are 2. The execution order O is the max execution order O increase. By this way we can determine the $rank_d$ and the execution order of each block. Step 3 determines the $rand_u$ of each block. The $rand_u$ determine by the max $rank_d$ of child block add the PWM of current block. Then, the algorithm determines which blocks are *critical blocks* in Step 4. In Fig.3 we can find the $rank_d$ +$rand_u$ of {b2,b15,b21,b29} equal to the $rand_u$ of b1, those block are the critical block. In order to split block set, we need to save the information of critical block for step 6.

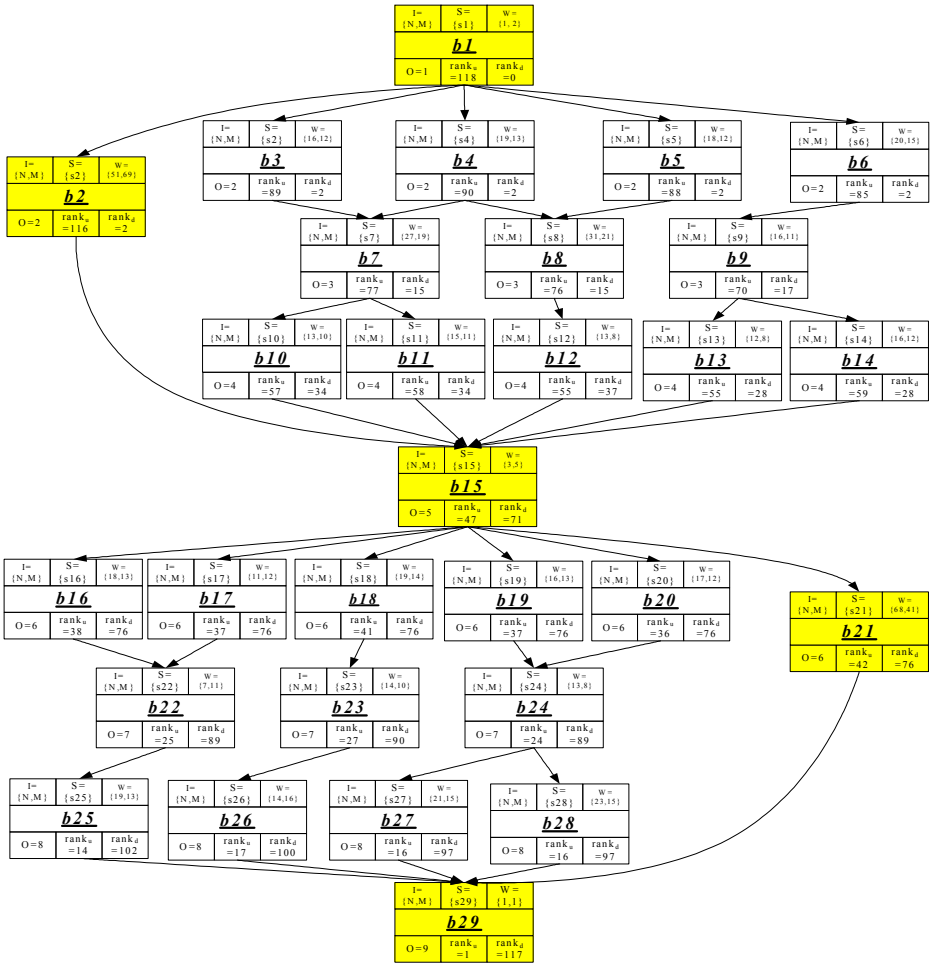

Fig. 3. WPG of a synthetic example.

Fig. 3 illustrates the WPG of the synthetic program, which is processing in stages stated above. In this WPG, the colored blocks are critical blocks.

When the critical blocks are determined in Step 5, it partition all blocks in the WPG into several Sections. Fig. 4 illustrates the result of the given WPG, which is partitioned into five Sections, Section1:{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 blocks are 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 critical block in Section 2, Step 5 is firstly used to schedule b2 to reduce the waiting and synchronization frequencies. The remaining blocks are partitioned into three wavefronts according to the Oi of each block, by calling Step 6. Finally, iw1={b3, b4, b5, b6}, iw2={b7, b8, b9}, iw3={b10, b11, b12, b13} are determined.

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

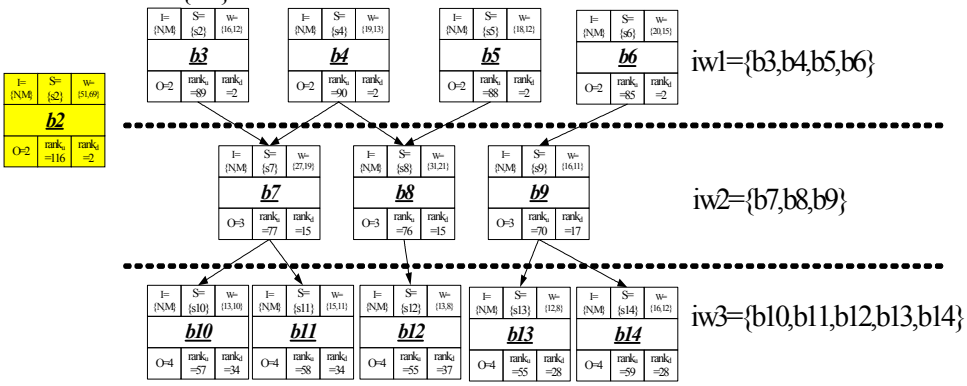

Fig. 4. Scheduled WPG of *Section* 2.

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={ϕ}

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={ϕ }

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={ϕ }

Fig. 3. Output of the Critical Block scheduling algorithm.

In Step 7, the execution schedule is generated as shown in Fig. 6. Fig. 5 shows the graph-mode of the execution schedule. The shaded blocks 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.

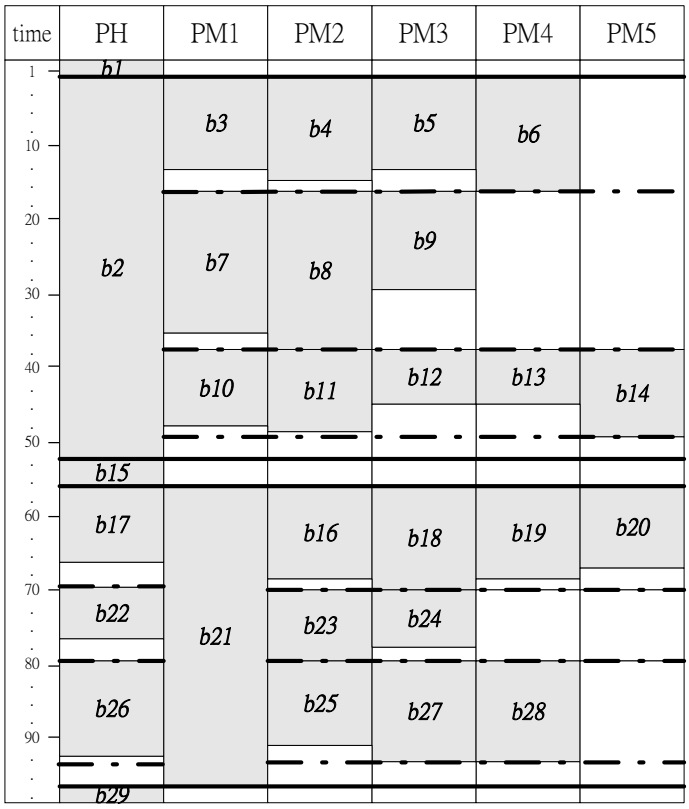

Fig. 4. Graphical execution schedule of the given example.

Sometimes, the execution schedule may occupy more processors than are present in the architectural configuration. Therefore, Step 8 modifies the execution schedule as necessary. The sub-step of Step 8 is finding the minimal load processor and place the comport block. If PH is idle, find the maximal PHW block to fill it. Then using a loop to find minimal load processor to fill it and plus the PMW of block to its load. Redo this loop until all block fit in processor.

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 [11] based on MINT simulator [14]. 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 [16].

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. "1H-1M” implies that the application is transformed and scheduled by our previous Pair-Selection Scheduling (PSS) [5] for the one-P.Host and one-P.Mem configuration of the PIM architecture. “1H-nM” implies that the application is transformed and scheduled by Critical Block Scheduling mechanism for the one P.Host and many P.Mem configuration of the PIM architecture.

Table 2 and Fig. 7 indicate that swim and cg have quite a good speedup when the Critical Block 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 1H-1M scheduling mechanism can also yield improvements. strsm exhibits an extremely high parallelism but a rather few memory access, so the Critical Block Scheduling mechanism is more suitably adopted than the 1H-1M scheduling mechanism. TISI cannot generate speedup when the 1H-1M scheduling mechanism is applied, since it is a typical CPU bounded program, and involves many dependencies. The Critical Block 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 1H-1M scheduling mechanism is applied. However, an additional overhead is generated when the Critical Block Scheduling mechanism is applied. Accordingly, 1H-1M and Critical Block Scheduling mechanisms are suitable for different situations. Choosing the 1H-1M or Critical Block 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	1H-1M	1H-nM	Speedup		
				1H-1M	1H-nM	n (Occupied P.Mem)
swim	228289321	116669760	52168027	1.96	4.38	6
cg	91111840	51230772	32124287	1.78	2.84	4
TISI	133644087	173503404	91098174	0.77	1.47	2
fft	117998621	101841407	110399171	1.16	1.07	2
strsm	201133647	139990872	53711479	1.44	3.74	5

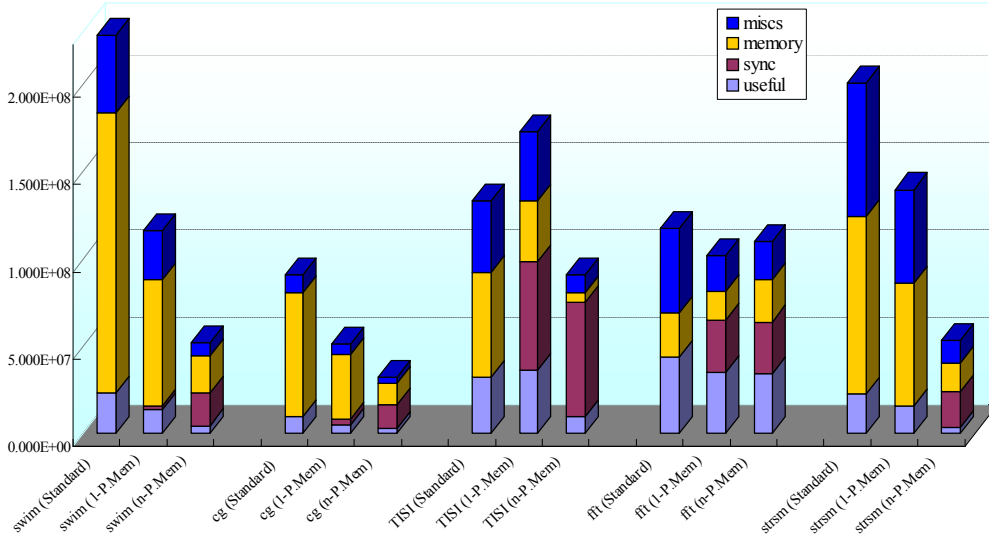

Fig. 7. Execution times of five benchmarks obtained by Standard, 1H-1M and 1H-nM.

6 Conclusions

This study proposes a new scheduling mechanism, called Critical Block Scheduling, with Octans system for a new class of high-performance SoC 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 Critical Block 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 MPSoC/CMP 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] Agrawal, K., He, Y., Hsu, W-J., Leiserson, C.: Shared Memory Parallelism: Adaptive Scheduling with Parallelism Feedback. In Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Mar. (2006).

[2] Armstrong, R., Hensgen, D., and Kidd, T.: The Relative Performance of Various, Mapping Algorithms is Independent of Sizable Variances in Run-Time Predictions. In Proceedings of 7th IEEE Heterogeneous Computing Workshop, Mar. (1998), 79-87.

[3] Arora, N., Blumofe, R., Plaxton, C.: Thread Scheduling for Multiprogrammed Multiprocessors. In Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, Jun. (1998).

[4] 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).

[5] 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.

[6] Cintra, M., Torrellas, J: Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors. In Proceedings of 2002 Eighth International Symposium on High-Performance Computer Architecture, Feb. (2002), 43-54.

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

[8] 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).

[9] 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).

[10] 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).

[11] 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).

[12] 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.
[13] Llosa, J.; Gonzalez, A.; Ayguade, E.; Valero, M.: Swing Module Scheduling: a Lifetime-Sensitive Approach. In Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, Oct. (1996), 80-86.
[14] 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.
[15] 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.
[16] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P.: Numerical Recipes in Fortran 77. Cambridge University Press, (1992).
[17] 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
[18] Swanson, S., Michelson, K., Schwerin, A. and Oskin, M.: WaveScalar. MICRO-36, Dec. (2003).
[19] 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
[20] 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
[21] Zhou, Y., Wang, L., Clark, D., Li, K.: Thread Scheduling for Out-of-Core Applications with Memory Server on Multicomputers; In Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems, May (1999).