COSPIM: A Program Optimization System for Tightly-Coupled Heterogeneous Environments

SLO-LI CHU
Department of Information and Computer Engineering
Chung Yuan Christian University
Chung-Li, TAIWAN

Abstract: Processor-in-memory is a new class of computer architecture designed for reducing the performance gap between the processor and the memory. This architecture provides a tightly-coupled heterogeneous environment by integrating different processors in a system. An efficient parallelization and optimization mechanism is necessary for this system to transform the existed applications to achieve better performance. In this paper, we propose a comprehensive framework, COSPIM, based on the statement viewpoint in our early SAGE system. It integrates program decomposition, ETC (expected time to compute) evaluation and scheduling mechanisms together. We describe how COSPIM splits statements and produces schedule to execute on the host processor and the coprocessor simultaneously. The experimental results of this approach are also discussed.

Key-words: COSPIM, Statement-based analysis, Processor-in-memory

1 Introduction

It is widely known that memory system has been one of the bottlenecks in high-performance computers because of the increasing gap between the processor speed and memory latency. For this reason, a new computer architecture, called Intelligent Memory (IRAM) or Processor-in-Memory (PIM) [2][5][6], integrates the processor and memory on a chip. This chip can be used to replace the original main memory chip, instead of rebuilding a whole new system. In this situation, PIM chips can act as coprocessors when main processor spawns them. This approach is taken by Active Page [5], DIVA [2] and FlexRAM [8], among others.

The architecture of integrating the host (main) processor and the memory processor provides a tightly-coupled heterogeneous (TCH) environment. The host processor is more powerful associated with deep cache hierarchies and higher latency to access memory. Contrarily, memory processors are usually less powerful but with lower latency in memory access. One way to exploit the TCH environment is to decompose an application task into portions; each portion is computationally well-suited for that processor. In conventional heterogeneous computing (HC) models, since the machines are loosely-coupled, they assumed that these portions (or subtasks) are independent coarse-grain subtasks. But in TCH environment, the host and memory processors are connected by the original memory bus such that they have the same communication latency in memory access. Therefore, we need new program decomposition and scheduling mechanisms to deal with the situation.

In this paper, we introduce a new comprehensive analysis and scheduling system called COSPIM (Code Optimization System for Processor-in-Memory). We call it statement-based analysis because the statements in a loop are used as the basic analyzing units, which is different from the iteration-based in the conventional parallelizing systems. The benefits of statement-based approach are described in [10]. Based on our original SAGE system [11], COSPIM operates in several phases. The first is to decompose the source application into several blocks which contain minimal number of statements in a loop. Then a corresponding Weight Partition Dependence Graph (WPG) is constructed according to their dependence relation. In the third phase, it adopts a code-analyzed mechanism to predict the ETC (expected time to compute) of each block. By the information of ETC and the dependence relation of WPG graph, COSPIM apply a static scheduling heuristics in the fourth phase to get the execution schedule for the host and memory processor.
Finally, if the workload is unbalanced, the fifth phase is used to refine the blocks by loop splitting mechanism and go back to the prior phase to reschedule them.

The organization of this paper is as follows. In section 2, the previous ETC prediction and scheduling algorithms are reviewed. Section 3 presents our COSPIM system and the whole phases in detail. Section 4 exhibits four TCH environments and their parameters. In section 5 we demonstrate the experimental results of four benchmarks running on these five TCH platforms and the speedup obtained when COSPIM is used. Finally, section 6 gives the conclusion and future works.

2 Related Works

General heterogeneous computing is a type of parallel computing, where a large, distributed network of heterogeneous machines is used as a single computational entity. Applications executing in this environment consist of a set of coarse-grained, precedence-constrained tasks. The performance is largely determined by how these tasks are assigned to the machines. The construction of such assignment is based on two factors: a precise scheme to predict the execution time of tasks and an effective task scheduling mechanism. In this section, we will review a set of ETC prediction schemes and scheduling algorithms.

In reference [16], three major solutions have been proposed to solve the execution time estimation problems: code analysis [14], analytic benchmarking/code profiling [17], and statistical prediction [13]. In code analysis, an estimated execution time is found through the analysis of source code of task. A given code analysis technique may be limited to a specific code type or a limited class of architectures. But for a specific environment, it may provide rather good predict result by its simple and efficient mechanism. The second solution, analytic benchmarking/code profiling, was firstly presented by Freund [17]. It defines a number of primitive code types. On each machine, benchmarks are used to evaluate the performance of the machine for each code type. Code profiling attempts to determine the composition of a task in terms of code types. The results of analytic benchmarking and code profiling are then combined to produce an ETC estimation. In the third solution, statistical prediction algorithms make predictions using the past observations. A set of past observations is recorded for each machine and then used to make new execution time predictions. These mechanisms have been presented by Iyer [13].

In the remainder of this section, we review a set of scheduling algorithms in the literatures [15]. The descriptions of these algorithms are sketched as follows.

OLB: Opportunistic Load Balancing (OLB) assigns the tasks, in arbitrary order, to the next available machine, regardless of task's expected execution time on that machine [12].

UDA: User-Directed Assignment (UDA) assigns each task, in arbitrary order, to the machine with the best expected execution time for the task [12].

Fast Greedy: Each task, in arbitrary order, is assigned to the machine with the minimum completion time for the task [12].

Min-min: The minimum completion time for each task is computed on all machines. The task with the overall minimum completion time is selected and assigned to the corresponding machine. The newly assigned task is removed, and the process repeats until all tasks are mapped [12].

Hybrid Remapper: Hybrid Remapper is based on a centralized policy which improves the statically obtained initial matching and scheduling by remapping to reduce the overall execution time. This algorithm operates in two phases. The first is executed prior to the application execution to partition the original application into subtasks, then generates the executing schedule. In the second phase, the hybrid remapper is executed during the application running time to collect the run-time results and reschedule these subtasks according to the practical situation.

3 COSPIM System Organization

In current parallelizing compilers for tightly-coupled multiprocessor environment, we always address on how to transform loops to make all or part of iterations executed simultaneously, i.e. the iteration-based viewpoint. This approach is suitable for homogeneous multi-processor systems but not for TCH platforms, because the behaviors of loop iterations are similar but the capabilities of heterogeneous processors are different. In addition, in the task level, coarse-grained parallelizing approaches of the conventional heterogeneous environments are difficult to apply in the TCH environments since one is loosely-coupled and the other is tightly-coupled. For these reasons, we take a different approach by using the statements in a
loop as our basic analyzing unit, called COSPIM. The organization of COSPIM is shown in Fig. 1. In this model, the source application is firstly decomposed into several blocks such that the number of statements in a block is as small as possible. Then a corresponding WPG graph is constructed according to the dependence relation of each block. After this, ETC (expected time to compute) evaluation stage will predict the ETC of each block running on different processors. Then these blocks are assigned to the most suitable processors according to their ETC and the execution order produced in the scheduling stage. Finally, if the execution schedule is unbalanced, the block refinement stage will further split blocks into several sub-blocks and rescheduled. In the following of this section, we will describe these mechanisms in detail.

**Fig. 1.** Whole sequence of analyzing stages for COSPIM system.

### 3.1 Program Decomposition and WPG Construction

In COSPIM, we first split the original dependence graph into Node Partition $\Pi$ [1] by loop distribution [1][3], then construct the Weighted Partition Dependence Graph (WPG), which will be used in the following stages. Here, we introduce some necessary definitions for partitioning loops at first.

**Definition 1 (Loop Denotation)** [1]

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

**Definition 2 (Node Partition $\Pi$)** [1]

On the dependence graph $G$, for a given loop $L$, we define a node partition $\Pi$ of $\{S_1, S_2, \ldots, S_d\}$ in such a way that $S_i$ and $S_j, 1 \leq k, l \leq d$ and $k \neq l$, are in the same subset if and only if $S_k \Delta S_l$ and $S_i \Delta S_j$ where $\Delta$ is an indirect data dependence relation. On the partition $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$, we define partial ordering relations $\alpha, \alpha^\wedge$, and $\alpha^\vee$ as follows.

For $1 \leq i, j \leq n$ and $i \neq j$:

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 $\delta$ 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. $\pi_i \alpha^\vee \pi_j$ iff there exist $S_k \in \pi_i$ and $S_l \in \pi_j$ such that $S_k \delta^\vee S_l$, where $\delta^\vee$ is the output dependence relation.

**Definition 3 (Weighted Partition Dependence Graph)**

For a given node partition $\Pi$ as in Definition 1, we define a weighted partition dependence graph $WPG(P, E)$. For each $\pi_i \in \Pi$, there is a node $b_{i} < I_i$, $S_i$, $W_i$, $O_i, r \in P$, where $I_i$ denotes the loop index and $S_i$ represents the body statements; $W_i$ is the weight of node $i$ in the form of $W_i(PH, PM)$ where $PH$ and $PM$ are the weights to P.Host and P.Mem respectively, and $O_i$ is the execution order of this node. There is an edge $e_{ij} \in E$ from $b_i$ to $b_j$ if $b_i$ and $b_j$ have dependence relations $\alpha, \alpha^\wedge$, and $\alpha^\vee$ (respective denoted by $\longrightarrow$, $\longleftarrow$, and $\Longrightarrow$) as in Definition 2.

According to these definitions, we present a formal procedure to partition loops in Algorithm 1.

**Algorithm 1.** (Program Decomposition Algorithm)

Given a loop $L = (I_1, I_2, \ldots, I_d)$ ($S_1, S_2, \ldots, S_d$)

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

**Step 2:** Establish a node partition $\Pi$ on $G$ as defined in Definition 2. If there are large blocks caused from the control dependence relations, convert the control
dependences into data dependences [4] first then partition the dependence graph.  

**Step 3:** On the partition \( \Pi \), establish a weighted partition dependence graph \( \text{WPG}(P,E) \) as in Definition 3.

### 3.2 ETC Evaluation

In TCH environment, each operation (e.g., branch, arithmetic, and memory operation) has different execution latency for different kind of processor. In section 2 we already described several classes of ETC prediction approaches. For the reason of fixed hardware configuration in TCHs, we proposed a static ETC evaluation mechanism to calculate the ETC of each block. This mechanism sums the ETCs of branch, integer, floating point, and memory operations for the host and coprocessors, respectively. The ETC of each operation on the target TCH platform is determined by the parameters of TCH architecture and our profiling results. Algorithm 2 describes the detailed mechanism.

**Algorithm 2. (ETC Evaluation)**

**[Input]**
- \( I \): the loop index set of a block.
- \( S \): the body statement set of a block.
- \( \text{TAB} \): the ETC table of target machine.

**[Output]**
- \( \text{W(PH, PM)} \): the ETC of a block for the host processor and coprocessor.

**[Intermediate]**
- \( \text{get_iter_num(I)} \): the total iteration number for the given loop index \( I \).
- \( \text{ph_weight} \): the ETC for host processor.
- \( \text{pm_weight} \): the ETC for coprocessor.
- \( \text{Branch(s, proc, table)} \): get ETC of branch operation from statement \( s \).
- \( \text{FP_OP(s, proc, table)} \): get ETC of floating point operation from statement \( s \).
- \( \text{INT_OP(s, proc, table)} \): get ETC of integer operation from statement \( s \).
- \( \text{MEM(s, proc, table)} \): get ETC of memory reference operation from statement \( s \).

**[Algorithm]**
- \( \text{ph_weight} = 0 \)
- \( \text{pm_weight} = 0 \)
- /* Compute the ETCs of statements in the block. */
- for each statement \( s_i \in S \) do
  - \( \text{ph_weight} = \text{ph_weight} + \text{INT_OP}(s_i, \text{PH, TAB}) + \text{FP_OP}(s_i, \text{PH, TAB}) + \text{Branch}(s_i, \text{PH, TAB}) + \text{MEM}(s_i, \text{PH, TAB}) \)
  - \( \text{pm_weight} = \text{pm_weight} + \text{INT_OP}(s_i, \text{PH, TAB}) + \text{FP_OP}(s_i, \text{PH, TAB}) + \text{Branch}(s_i, \text{PH, TAB}) + \text{MEM}(s_i, \text{PH, TAB}) \)
- end for
- \( \text{ph_weight} = \text{get_iter_num}(I) \times \text{ph_weight} \)
- \( \text{pm_weight} = \text{get_iter_num}(I) \times \text{pm_weight} \)
- return \( (\text{ph_weight}, \text{pm_weight}) \)

### 3.3 Scheduling

In the previous section, we have listed several scheduling mechanisms. But for the diversity of processors in TCH environment, we propose a scheduling algorithm for this architecture. In our method, the ETCs of the blocks in partition \( \Pi \) are determined first, and then the execution order of each block is determined according to their dependence relations. The blocks that can be executed simultaneously are assigned in a wavefront. The blocks in the same wavefront are scheduled on the host processor and coprocessors according to their ETCs.

**Algorithm 3. (Scheduling Algorithm)**

**[Input]**
- \( \text{WPG=(P,E)} \): the initial weighted partition dependence graph before the weight and execution order of blocks are determined.

**[Output]**
- \( \text{An execution wavefront schedule} \ W = \{ Wf_1, Wf_2, \ldots \} \) where \( Wf_i = \{ \text{PH}(b_{p1} \ldots b_{pb}), \text{PM}(b_{c1} \ldots b_{cd}) \} \) in which \( \text{PH}(b_{p1} \ldots b_{pb}) \) means that blocks \( b_{p1} \ldots b_{pb} \) will be assigned to \( \text{PH} \)Host in wavefront \( i \), \( \text{PM}(b_{c1} \ldots b_{cd}) \) means that blocks \( b_{c1} \ldots b_{cd} \) will be assigned to \( \text{PM} \)Mem in wavefront \( i \).

**[Intermediate]**
- \( W \): a working set of blocks to be visited.
- \( \text{wf_tmp}, \text{ph_sch}, \text{pm_sch} \): working sets of blocks for wavefront scheduling.
- \( \text{ph_tmp(h)}, \text{pm_tmp(m)} \): working arrays to store the blocks for wavefront scheduling.
max_wf: the maximum number of wavefront.
max_pred_O(b_i): the maximum execution order for all b_i’s predecessor blocks.
min_pred_O(b_i): the minimum execution order for all b_i’s predecessor blocks.
PHW(b_i): ETC of b_i for host processor.
PMW(b_i): ETC of b_i for coprocessor.

[Algorithm]
/* Initialization and ETC evaluation for each block */
for each b_i ∈ P do
  W_i (PH, PM) = ETC Evaluation (I_i, S_i)
  O_i = 0
end for

/* Execution order assignment */
W = P
for each b_i with no predecessors do
  O_i = 1
  W = W- {b_i}
end for
done = False
max_wf = 0
while done = False AND W ≠ ∅ do
  done = True
  for each b_i ∈ W do
    if min_pred_O(b_i) = 0 then
      done = False
    else
      O_i = max_pred_O(b_i) + 1
      W = W - {b_i}
      max_wf = max(max_wf, O_i)
    end if
  end for
end while

/* Scheduling */
for j = 1 to max_wf
  store all of b_i whose O_i = j in wf_tmp
  while done = False do
    done = False
    divide wf_tmp into two subsets a, b such that a ∪ b = wf_tmp and a ∩ b = ∅
    if [PHW(a)-PMW(b)] is minimum for all possible a and b then
      wf_j = {PH(a), PM(b)}
    end if
  end while
end for

3.4 Block Refinement
As we mentioned before, we must consider the diverse capabilities of processors if we want to achieve load balance in partitioning the job. According to the result of ETC evaluation, loop splitting is used to divide the original block into two parts when the workload is unbalance between the host processor and coprocessor. A simple Loop-Splitting algorithm is presented below:

Algorithm 4. (Block Refinement)
[Input]
WPG (P, E)
[Output]
WPG (P, E) /* After block refinement */

[Algorithm]
Step 1. Identify a block to be split from WPG.
Step 2. Compute the workload ratio by the weight of P.Host over that of P.Mem. (The weights can be obtained from WPG)
Step 3. Split the iteration space of the block by this ratio into two blocks, and modify WPG according to their dependence relations.
Step 4. If there are blocks to be split, go to Step 1.

4 TCH Architectures
In this section, we propose two classes of TCH platform to be used to examine our COSPIM system. The first TCH class is the intelligent memory system whose general organization is shown in Fig. 2. As we mentioned before, intelligent memory is mainly used to reduce the performance gap between the processor and memory. This architecture consists of two types of processors: the off-the-shelf processors in the ordinary computer and the coprocessors in the memory system. The coprocessors are integrated with DRAM with shorter memory access latency compared to the host processor. The detailed description of this architecture can be found in [2][5].

The second TCH class we proposed is a general asymmetric shared memory multiprocessor environment, whose organization is sketched in Fig. 3. The difference between these two TCHs is their memory access capability. For the experiment in the next section, we propose three intelligent memory systems (FlexRAM, Fast FlexRAM, and FlexRAM-2). Their representative parameters are listed in Table 1.
Experimental Results

The code generated by COSPIM is targeted to FlexRAM simulator [8] developed by IA-COMA Lab in UIUC. This simulation environment models dynamic superscalar multiprocessor and detailed memory behaviors cycle by cycle. Based on this platform, we simulate the four TCH architectures whose configurations and parameters are listed in Table 1.

The first TCH, FlexRAM, is derived from the original FlexRAM architecture, which contains one host processor and one coprocessor (memory processor). The organization is the same as in Fig. 2. As we have mentioned, the coprocessor has less computation power and working frequency but has better memory access capability. The second TCH, Fast FlexRAM, is the same as FlexRAM except that its working frequency is increased to simulate an environment of more powerful memory processor. As the result, the memory access latency is decreased and the performance is better than the typical memory processor. The third TCH, FlexRAM-2, simulates the architecture where the memory processors operates in a working frequency higher than the host processor. The organization is the same as in Fig. 2. The fourth TCH, asymmetric multiprocessor, contains two kinds of processors with different capabilities in a shared-memory fashion. The organization is shown in Fig. 3. In order to demonstrate the speedup, we simulate the standard environment for each TCH by using the host processor only.

The evaluated applications include four programs: swim from SPEC95, strmm from BLAS3, fft from [9],

![Fig. 2. The organization of general intelligent memory systems.](image1)

![Fig. 3. The organization of general asymmetric shared memory multiprocessor environment.](image2)

**Table 1. Configurations and parameters of four TCH architectures.**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Architecture</th>
<th>Intelligent Memory system</th>
<th>Asymmetric Mixed processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working Frequency</td>
<td>FlexRAM</td>
<td>Fast FlexRAM</td>
<td>FlexRAM-2</td>
</tr>
<tr>
<td>Superscalar Issue</td>
<td>800 MHz</td>
<td>800 MHz</td>
<td>400 MHz</td>
</tr>
<tr>
<td>Width/Type</td>
<td>(Out-of-order)</td>
<td>(Out-of-order)</td>
<td>(Out-of-order)</td>
</tr>
<tr>
<td>L1 Cache Size</td>
<td>32 KB</td>
<td>32 KB</td>
<td>32 KB</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>256 KB</td>
<td>256 KB</td>
<td>256 KB</td>
</tr>
<tr>
<td>Branched penalty</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Host Processor Memory Delay Cycles</td>
<td>88</td>
<td>88</td>
<td>88</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Architecture</th>
<th>Intelligent Memory system</th>
<th>Asymmetric Mixed processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar Issue</td>
<td>400 MHz</td>
<td>800 MHz</td>
<td>400 MHz</td>
</tr>
<tr>
<td>Width/Type</td>
<td>(In-Order)</td>
<td>(In-Order)</td>
<td>(In-Order)</td>
</tr>
<tr>
<td>L1 Cache Size</td>
<td>16 KB</td>
<td>16 KB</td>
<td>16 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Branched penalty</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Co-Processor Memory Delay Cycles</td>
<td>17</td>
<td>14</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Architecture</th>
<th>Intelligent Memory system</th>
<th>Asymmetric Mixed processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Working Frequency</td>
<td>100 MHz</td>
<td>100 MHz</td>
<td>100 MHz</td>
</tr>
<tr>
<td>Host Processor Memory Access Latency</td>
<td>262.5 ns</td>
<td>262.5 ns</td>
<td>262.5 ns</td>
</tr>
<tr>
<td>Co-Processor Memory Access Latency</td>
<td>50.5 ns</td>
<td>41.6 ns</td>
<td>50.5 ns</td>
</tr>
</tbody>
</table>
and a synthetic program synth. Their performances running on the above platforms are shown from Fig. 4 to Fig. 7 respectively, where "std-" stands for the environment using the host processor only; "opt-" denotes that the application is executed on the host processor and coprocessor simultaneously. The breakdown is divided into four parts: executing useful instructions (Busy), waiting for other processors (Sync), waiting for the memory (Mem), and other hazards (Other). We find that our COSPIM system obtains speedup not only on FlexRAM architecture but also on other TCH environments. Notice that fft gets less benefit in Fig. 6. The reason is that a lot of time is spent on computation rather than memory access. In this case, fft is more suitable for the coprocessor with higher computation power such as FlexRAM-2. Even so, the profits are also earned by the COSPIM system.

As a summary, Table 2 illustrates the speedup to the standard environment using PHost only (std-) on four different environments.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Platform</th>
<th>Swim</th>
<th>Strmm</th>
<th>FFT</th>
<th>Synthetic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FlexRAM</td>
<td>1.50</td>
<td>1.41</td>
<td>1.10</td>
<td>1.85</td>
</tr>
<tr>
<td></td>
<td>Fast FlexRAM</td>
<td>1.65</td>
<td>1.47</td>
<td>1.10</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>Asymmetric Mixed processors</td>
<td>1.05</td>
<td>1.32</td>
<td>1.10</td>
<td>1.52</td>
</tr>
<tr>
<td></td>
<td>FlexRAM-2</td>
<td>1.73</td>
<td>1.70</td>
<td>1.09</td>
<td>1.91</td>
</tr>
</tbody>
</table>
6 Conclusion and Future Works
In this paper, we proposed program decomposition and scheduling mechanisms for intelligent memory architectures to exploit the computing power of the host processor and coprocessor. The algorithms are not only applicable on FlexRAM architecture but able to be used on other TCH environments. From the experimental results, we find that COSPIM can extract up to twice speedup in two-processor TCH environment. This reveals the superiority of our COSPIM system. However, we only presented a framework for two-processors TCH environment to simplify the problem in this paper. Currently we still try to expand this framework for more processors TCH and enhance the capabilities of our COSPIM system.

7 Acknowledgement
This work is supported in part by the National Science Council of Republic of China, Taiwan under Grant NSC 94-2213-E-033-032.

References: