Pipeline Schedule Synthesis for Real-Time Streaming Tasks with Inter/Intra-instance Precedence Constraints

Yi-Sheng CHIU, Chi-Sheng SHIH, Shih-Hao HUNG
cshih@csie.ntu.edu.tw
Embedded Systems and Wireless Networking Lab
Graduate Institute of Networking and Multimedia
Department of Computer Science and Information Engineering
National Taiwan University

***Abstract*—Heterogeneous multi-core platforms are widely accepted for high performance multimedia embedded systems. Although pipeline techniques can enhance performance for the multi-core platforms, data dependencies for processing compressed multimedia data makes it difficult, if not impossible, to automate pipelined design. In this paper, we target on multimedia streaming applications on heterogeneous multi-core platforms and develop the "Tile Piecing Algorithm" for pipelined schedule synthesis within the targeted applications and platforms. The algorithm gives an efficient way to construct a pipelined schedule. The performance evaluation result shows that the algorithm performs as well as the optimal algorithm to utilize the computation resource. On the other hand, the algorithm only takes hundreds of milliseconds to complete, which is less than one tenth of the running time for optimal algorithm. Last, the synthesized schedule is well packed. The short execution time and schedule make-span makes the algorithm more practical to be used during the run-time.**

I. INTRODUCTION

Pipeline execution on microprocessors provides better parallelism and throughput. Examples include *software pipelining* [1], [2], [3] in compiler community, *pipeline scheduling for system-level design* [4], [5], and *functional pipelining* [6], [7], [8] in DSP design. However, the processes for synthesizing pipeline schedules with intra/inter-instance precedence constraints in most design automation levels have so many design parameters that it is extremely difficult, if not impossible, to automate the process. Kim, Chang, and Cho [5] showed that when sequentially executing a MPEG4 video encoding process on a 50MHz microprocessor, it requires 13,675 cycles to encode one macro-block and can barely encode 10 frames per second. However, executing the process in pipeline manner on the same platform can increase the encoding rate to 25 frames per second. Chatha and Vemuri [4] also show that pipelined schedule can maximize the throughput performance for transformative applications in general. Although pipelined schedules can enhance the performance, it requires enormous computation to synthesize the optimal schedule. Chatha and Vemuri [4] employed the simulated annealing algorithm which finds near optimal schedules but requires extremely long computation times.

To shorten the delay caused by data transmission and storage, many applications only keep one full copy of redundant or closely related data points. While storing or transmitting data, the application omits the redundant data points and keeps the difference of closely related data points. The process is called *compression*. When presenting the information to users, the applications restore the data points by duplication or computation, which is called *decompression*, during the run-time. The compression and decompression process lead to data dependence among task instances in applications. Specifically, data dependency among macroblock decoding process makes it extremely difficult to synthesize functional parallelized schedule. Many published research works report their attempts on ESL design automation tools under different assumptions. Ikeda [9] and Kim [10] applied the pipelined schedule for chip design. The above works handcrafted pipelined schedules for different applications and demonstrated the use of pipelined schedules. However, no algorithms for general applications are developed. Our framework is similar to the one proposed by Chatha and others [4]. However, the above algorithms require heavy computation to synthesize the schedule. On the other hand, Huang and others [11] present a methodology for synthesizing streaming applications, modeled as task graphs, for pipelined execution on homogeneous multi-core architectures which balances the workload assigned to each core, and minimizes inter-core communication traffic. In this paper, we are concerned with heterogeneous multi-core platforms.

In this paper, we present the developed Tile Piecing Algorithm to construct a repeated pipelined schedule for the real-time streaming applications with intra/inter-instance precedence constraints. Compared to optimal algorithm, the tile piecing algorithm provides near optimal pipeline schedule to meet the constraints on precedence and throughput requirements. In the mean time, the tile piecing algorithm only requires one tenth of the execution time to synthesize the near optimal schedules. Furthermore, the synthesized schedule requires only one third to one forth of the memory space to store the schedule, which makes the pipeline scheduling approach practical to use.

The remainder of this paper is organized as follows. Section 2 defines the terms used in this paper and the targeted problem. The developed algorithm is presented in Section 3. The performance evaluation results of the algorithm are presented in Section 4. Finally, Section 5 concludes the paper.

II. TERMS AND PROBLEM DEFINITION

Thus far and in the remaining discussions, we use the terms used in EDA community and real-time computing community. The targeted platforms are bus-based systems which consist of processing elements, buses, and shared memory. A *processing element*, denoted by PE, can be a microprocessor, digital signal processor, or application special designed IC. The bus system can be a single-bus system or multi-bus system. Examples are AMBA by ARM, Coreconnect by IBM, and

978-3-9810801-7-9/DATE11/©2011 EDAA

Wishbone by OpenCores. The shared memory on each bus is used as a storage for data exchange.

A *task* consists of a sequence of computation works each of which can be completed on one of the PEs. We denote tasks by τ_1, τ_2, ..., etc. In this paper, we are concerned with tasks that repeat themselves, called *periodic task*s. The minimal time interval between any two consecutive arrivals of task τ_i is its *task period*, denoted by p_i. The execution of task τ_i at every period is called a *task instance*, denoted by τ_i^k where k is greater than zero.

The execution of task τ_i can be divided into a sequence of functions executed on different processing elements. We call each of the functions a *subtask* of task τ_i, denoted by $\tau_{i,j}$ for $j \geq 1$. The execution of each subtask is assumed to be a synchronous request. The execution order of subtasks can be sequential, conditional branched, or looped. Execution time $e_{i,j}$ of subtask $\tau_{i,j}$ is the amount of time needed to complete subtask $\tau_{i,j}$. In real-time computing literature, execution time of a task usually refers to worst case execution time of the task so as to complete its execution in any case before a given time instance. However, this definition is best suitable for time critical real-time applications and not suitable for time sensitive or soft real-time applications. In this paper, we do not limit the execution time to be the worst case execution time and allow the system designers to use either worst case execution time or average execution time for execution time of a subtask. We also assume that each subtask is mapped to a specific type of processing elements on the platform.

Precedence constraint graph (PCG) for task τ_i, denoted by G_i, defines the precedence constraints among task instances and subtasks. A precedence constraint graph G_i is a directed graph and consists of a set of vertex and a set of edges, denoted by $G_i = (\mathbf{V}_i, \mathbf{E}_i)$. Vertex $v_{i,j} \in \mathbf{V}_i$ represents subtask $\tau_{i,j}$ of task τ_i. Edge $\overrightarrow{jk} \in \mathbf{E}_i$ where $j \neq k$ is a directed edge from vertex $v_{i,j}$ to vertex $v_{i,k}$ and represents that destination subtask $\tau_{i,k}$ cannot start before source subtask $\tau_{i,j}$ completes. In other words, subtask $\tau_{i,k}$ is *causally related* to and has a *precedence constraint* with subtask $\tau_{i,j}$.

Precedence constraint exist between two subtasks of same task instance or different task instances. The former is called *intra-instance* precedence constraint and the later is called *inter-instance* precedence constraint. For every two subtasks, there is at most one intra-instance precedence constraint. For every two subtasks, inter-instance precedence constraint may exist for several task instances. Hence, there can be more than one inter-instance precedence constraint.

In general, precedence constraint among subtasks are caused by data dependence among them. When there is no dependence between two subtasks, it does no harm to say that there is no precedence constraint between the two subtasks. To define inter/intra-instance precedence constraint, we take into account the dependence on the input data for subtasks. *Precedence constraint dimension* of task τ_i, denoted by D_i, is the number of dimensions to partition the input data of subtask $\tau_{i,j}$ for $j \geq 1$ so as to uniquely identify the input data for each subtask. For instance, a H.264 video data can be divided into frames along the time line, slices in each frame, and macro-block in each slice. Its partition dimension is three. *Precedence vector*, denoted by $\phi = (u_1, u_2, \ldots, u_{D_i})$, defines the precedence constraint between two subtasks of task τ_i, and is a D_i element vector of which each element u_m for $1 \leq m \leq D_i$ is a non-positive integer. The m-th element of the precedence vector, u_m, is the difference between m-th index of processed data identification of two subtasks. The non-positive integer refers to that the set of data is processed before the subtask starts. When two subtasks are not causally related, the precedence vector is undefined. When two subtasks of same task instance process the same set of data and are causally related, its precedence constraint vector is a zero vector and there is an intra-instance precedence constraint. When one subtask takes the output of the subtask of difference task instance as its input, there exists an inter-instance precedence constraint between them and their precedence vector is not a zero vector.

Precedence constraint size on edge $\overrightarrow{jk}$, denoted by $s_{j,k}$, is the number of precedence constraints between two subtasks. When precedence constraint size between two subtasks is zero, they are not causally related. Otherwise, they are causally related. *Precedence vector set*, denoted by $\mathbf{\Phi}_{j,k} = \{\phi_1, \phi_2, \ldots, \phi_{s_{j,k}}\}$, is the set of precedence vectors on edge $\overrightarrow{jk}$.

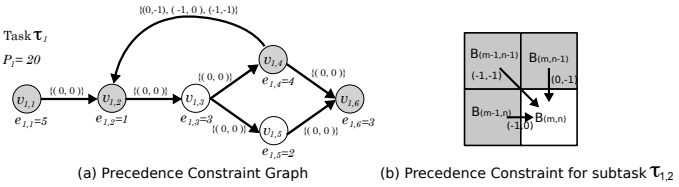

Fig. 1. Precedence Constraint Graph for Task τ_1

Figure 1 illustrates data dependence graph for task τ_1. The figure on right-hand side shows that the input data are partitioned on two dimensions: horizontally and vertically. Each data block is denoted by $B_{i,j}$. The arrows in the figure represent the data dependence while decoding data block $B_{m,n}$. In this example, subtask $\tau_{1,2}$ has three precedence constraints which require the computed results from task $\tau_{1,4}$ while processing data-block $B_{m-1,n-1}$, $B_{m,n-1}$, and $B_{m-1,n}$. The figure on left-hand side shows the data dependence graph of task τ_1. Task τ_1 has six subtasks, represented by six vertexes $v_{1,j}$ for $1 \leq j \leq 6$. Subtask $\tau_{1,2}$ is causally related to subtask $\tau_{1,1}$ and subtask $\tau_{1,4}$. Precedence vector on edge $\overrightarrow{12}$ is a zero vector and shows that these two subtasks of same task instance have a precedence constraint. When subtask $\tau_{1,2}$ processes data-block $B_{m,n}$, it also requires the output data from subtask $\tau_{1,4}$ when subtask $\tau_{1,4}$ processes data-block $B_{m-1,n-1}$, $B_{m,n-1}$, and $B_{m-1,n}$. The precedence vector set for subtask $\tau_{1,2}$ and $\tau_{1,4}$ is $\mathbf{\Phi}_{4,2} = \{(-1,-1), (0,-1), (-1,0)\}$.

Pipeline scheduling synchronizes the execution of processing elements in the system so as to reduce the overhead of data exchange and synchronization on multi-core platforms. For instruction level pipeline scheduling, the processing elements are synchronized on every clock cycle. For system level pipeline scheduling, the processing elements are synchronized on a fixed-length time interval. *Pipeline stage* for a set of processing elements is a time interval within which a set of allocated subtasks are scheduled. A subtask can start, resume,

and be interrupted in a pipeline stage. The length of one pipeline stage is *stage length*, denoted by l_π, and is a constant for all the processing elements on the targeted platform. The pipeline stages for all the processing elements on the targeted platform start at same time instant. In other words, they are synchronized.

Pipeline schedule for PE_i, denoted by π_i, defines the execution order of subtasks for a given time interval on processing element PE_i. The schedule on every processing element is grouped based on pipeline stage. At the start of each pipeline stage, the processing elements are instructed to write data to and read data from shared buffers. Within a pipeline schedule, *scheduled stage* for subtask $\tau_{i,j}^k$, denoted by $\rho_{i,j}^k$, is the index of the pipeline stage within which $\tau_{i,j}^k$ is scheduled. *Schedule makespan* for pipeline schedule π, denoted by Z_π, is the length of the shortest pipeline schedule that repeats itself. *PE utilization* for pipeline schedule π, denoted by u_π, is the ratio of the total time intervals scheduled for subtasks to total available processing time within schedule makespan Z_π. The higher PE utilization, the better resource utilization for the system. When PE utilization is one, there is no idle time within the pipeline schedule for all the processing elements.

A pipelined schedule for real-time workload needs to meet given timing constraints and precedence constraints. Precedence constraint is defined between subtasks of same task instance and of different task instances. Definition 2.1 defines this constraint for pipeline schedule π.

Definition 2.1: [Precedence Constraint] Given precedence constraint graph $G_i = (\mathbf{V}_i, \mathbf{E}_i)$ for task τ_i and a schedule π_i, precedence constraint of task τ_i is met if $\rho_{i,k}^{x+\phi_l[m]} - \rho_{i,j}^x \leq \phi_l[m]$ for $\max(1, \parallel \phi_l[m] \parallel) \leq x \leq \frac{Z_\pi}{l_\Pi}$) for $1 \leq m \leq D_i$, $\forall \phi_l \in \mathbf{\Phi}_{j,k}$,$\forall \overrightarrow{jk} \in \mathbf{E}_i$.

Throughput constraint, defined below, is the minimum number of completed task instances for every time unit.

Definition 2.2: [Throughput Constraint] Given precedence constraint graph $G_i = (\mathbf{V}_i, \mathbf{E}_i)$ for task τ_i and a schedule π_i, throughput constraint of task τ_i is met if $\frac{Z_{\pi_i}}{p_i} \geq$ the throughput constraint of task τ_i.

A *feasible pipelined schedule* for a set for periodic tasks is a pipeline schedule which meets the throughput constraints for all the tasks and the precedence constraints for all the subtasks. Below provides its formal definition.

Definition 2.3: [Feasible Pipelined Schedule] Given a set of independent tasks, $\mathbf{T} = \{\tau_1, \tau_2, \ldots, \tau_N\}$, and their data dependence graphs, $\mathbf{G} = \{G_1, G_2, \ldots, G_N\}$, pipeline schedule π is feasible if throughput constraint defined in Definition 2.2 and precedence constraint defined in Definition 2.1 for all tasks in task set $\mathbf{T}$ are met.

The targeted problem is to find a feasible schedule with minimal schedule makespan for a set of tasks and a set of processing elements such that the throughput constraint for all the tasks are met and the precedence constraint among all the subtasks are met. We define the targeted problem as follow.

Definition 2.4: [Pipeline Schedule Synthesis Problem] Given a set of independent tasks, $\mathbf{T} = \{\tau_1, \tau_2, \ldots, \tau_N\}$, and their precedence constraint graphs, $\mathbf{G} = \{G_1, G_2, \ldots, G_N\}$, find a feasible pipelined schedule π, defined in Definition 2.3, such that its schedule makespan Z_π is minimized.

III. Tile Piecing Algorithm for Pipeline Schedule Synthesis

We developed the *Tile Piecing algorithm*, TPA for short, to synthesize pipelined schedules. The rationale of the algorithm resembles the process for tile workers to put several artworks on one cement wall. The goal is to put as many artworks as possible on the wall and none of them overlaps with each other. TPA has three major phases: *initialization phase*, *PE allocation and schedule synthesis phase*, and *schedule refinement phase*. The first phase initializes the required data structure; the second phase allocates subtasks to proper processing elements and schedules them to meet timing and precedence constraints. The last phase refines the schedule to shorten the make-span of the schedules. Figure 2 shows an example task set, which is

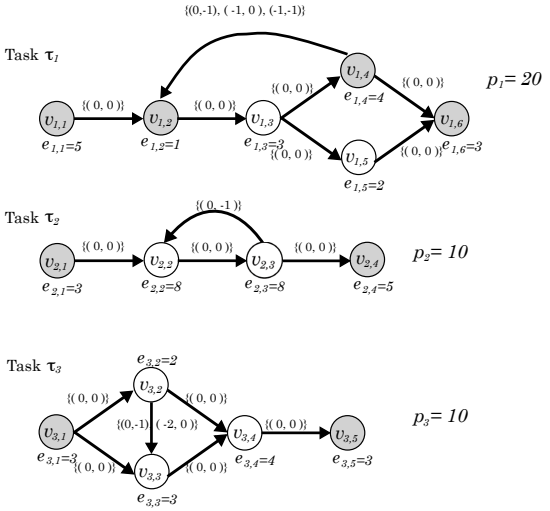

Fig. 2. An example task set of pipelined schedule synthesis

used to illustrate the algorithm. In this example, there are three tasks $\mathbf{T} = \{\tau_1, \tau_2, \tau_3\}$ and three PCGs $\mathbf{G} = \{G_1, G_2, G_3\}$. There are two types of processing elements: DSP and MPU, represented by white circles and grey circles.

A. Initialization Phase

In this phase, the algorithm estimates the number of processing elements required for each type. The estimation considers the case that only the tasks with same task period are scheduled on same processing element. For each type of processing element PE_x, the algorithm groups the tasks based on their task periods into period classes. *Period class* for task period p, denoted by PC_p, consists of a set of periodic tasks whose periods are p. Then, the algorithm sums up the required computation time of all the subtasks in each period class. Last, the estimated number of processing element PE_x required for period class PC_p, denoted by $\hat{N}_{x,p}$, is the ceiling ratio of total execution time to period p.

Take Figure 2 as an example. First, the algorithm groups the tasks into two period class PC_{20} and PC_{10}: $PC_{10} = \{\tau_2, \tau_3\}$ and $PC_{20} = \{\tau_1\}$. The total execution time for all the subtasks in PC_{10} is 14 on MPU and 25 on DSP. Hence, the estimated number for MPU and DSP are 2 and 3. Similarly, the estimated number of MPU and DSP required for PC_{10} are 1 and 1. In the last step, the algorithm sums up $\hat{N}_{MPU,10}$ and $\hat{N}_{MPU,20}$ to estimate the number of MPU required, which is $\hat{N}_{MPU} = 3$. That for DSP is $\hat{N}_{DSP} = 4$.

Last, we set stage length to the greatest common divider of all periods, which is 10 in this example. If the computed stage length is less than the execution time of any subtask, we will logically split the execution of that subtask into several subtasks so that the execution time of each split subtask is less than the stage length.

B. PE allocation and schedule synthesis phase

In the second phase, the algorithm allocates the subtasks to its mapped processing element type and synthesizes a feasible schedule. In the first step, the algorithm constructs *Processor Allocation Table*, denoted by *PAT*, to allocate subtasks to specific processing elements. Next, it takes into account intra-instance and inter-instance precedence constraints.

PE allocation problem is a two-dimension bin packing problem, which is NP-complete. TPA adapts the worst fit algorithm to obtain a near optimal solution for PE allocation. For each period class, the subtask with longest execution time is selected and allocated to its mapped processing element so that the remain available processing time of the PE is the greatest. In a PAT, the column and row index, denoted by α and β, represent the index of processing elements and that of pipeline stages. Each cell in the table stores the index of the allocated subtasks. The number of rows and columns of PAT is the ratio of maximal period class to stage length and the total estimated number of processing elements, which are computed in the first phase.

Take the task set in Figure 2 as an example. Figure 3 shows their PAT. Because the maximum class period is 20, the *stage length* is $GCD(10, 20) = 10$, and the total estimated number of required processors is 7, the PAT is a 2×7 table. Task τ_1 is in $PC_{(20)}$, so the subtasks of task τ_1 are allocated in the first row and second row of PAT.

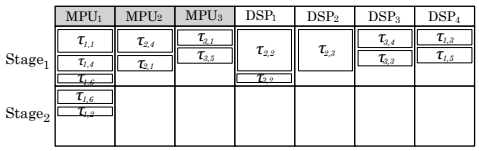

Fig. 3. Processor Allocation Table for $\{\tau_1, \tau_2, \tau_3\}$

The second step schedules the subtasks having intra-instance precedence constraints. Because the main concept of pipeline schedule is to synchronize the schedule for different processing elements at the boundary of pipeline stages. Hence, when there is any intra-instance precedence constraint, the two subtasks need to be properly scheduled. Otherwise, the schedule may fail to meet timing constraint. For instance, when the two subtasks are mapped to different processing elements and are scheduled in the same pipeline stage, the succeeding subtask will wait for the make-span of the schedule to receive the required data.

The algorithm traverses the precedence constraint graph in depth-first order to schedule all the subtasks having intra-instance precedence constraint. It starts from the source subtask, which has no incoming edge, in the graph and set 1 as its pipeline stage. For each subtask at the head of the queue $\tau_{i,j}$, the algorithm adds subtask $\tau_{i,k}$ into the queue if $\phi_{j,k}$ is a zero vector. Then, it uses the following rules to set the schedule stages of subtask $\tau_{i,j}$ and $\tau_{i,k}$.

- [Rule 1] When $\alpha_{i,j} = \alpha_{i,k}$ and
  - [1.1] $\beta_{i,j} = \beta_{i,k}$, $\rho_{i,k} = \rho_{i,j}$
  - [1.2] $\beta_{i,j} \neq \beta_{i,k}$, $\rho_{i,k} = \rho_{i,j} + \frac{p_i}{l_\pi}$
- [Rule 2] When $\beta_{i,j} \neq \beta_{i,k}$ and
  - [2.1] $\alpha_{i,j} < \alpha_{i,k}$, $\rho_{i,k} = \rho_{i,j} + (\alpha_{i,k} - \alpha_{i,j})$
  - [2.2] $\alpha_{i,j} \geq \alpha_{i,k}$, $\rho_{i,k} = \rho_{i,j} + (\alpha_{i,k} - \alpha_{i,j}) + \frac{p_i}{l_\pi}$

Rule 1.1 schedules the two subtasks in same pipeline stage because the two subtasks are allocated to the same processing element and there is no need for schedule synchronization. Rule 1.2 postpones the execution of the causally related subtask for another period because they are allocated to different processing elements and schedule synchronization is required. Rule 2 is concerned with the case that the two subtasks are allocated to different pipeline stages. When the source subtask is allocated to an earlier pipeline stage, the destination subtask is scheduled after its source subtask for the difference of their allocated pipeline stages. Otherwise, the destination subtask must be delayed for another period. Figure 4 shows the schedule result for task τ_1. In this example, four pipeline stages are needed to schedule six subtasks of task τ_1.

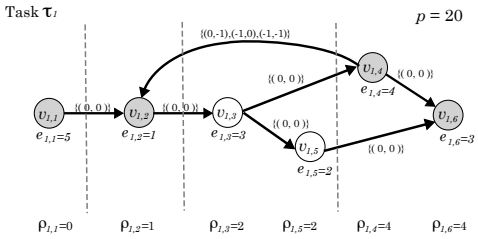

Fig. 4. PCG Partition for Task τ_1

The third step schedules subtasks having inter-instance precedence constraint based on their data region. *Maximum data dependent region* for task τ_i, denoted by $\mathcal{R}_i$, is the minimal data partition rectangle which are required for a task instance of task τ_i to complete its computation. In the maximum data dependent region, the data block which is required for subtask $\tau_{i,1}$ is called *core block* of the maximum data dependent region. Take task τ_1 in Figure 2 as an example. All zero vectors in G_1 represent the intra-instance precedence constraint; three non-zero vectors represent the inter-instance precedence constraints. Specifically, to process macro-block $B_{m,n}$, subtask $\tau_{1,2}$ requires the output data of subtask $\tau_{1,4}$ while processing macro-block $B_{m,n-1}$, $B_{m-1,n}$, and $B_{m-1,n-1}$. Hence, $\mathcal{R}_i$ is a 2x2 macro-block region. Figure 5(a) illustrates the maximum data dependent region of task τ_1. The block on the lower right is the core block of the region. The maximum data dependent region of task τ_2 and τ_3 are shown in Figure 5(b) and (c).

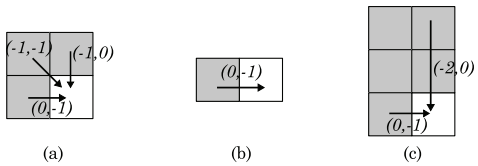

Fig. 5. Maximal Data Dependent Region for task τ_1, τ_2, and τ_3 in Figure 2

Greatest stage difference of task τ_i, denoted by δ_i, is the greatest absolute difference between scheduled stages of two causally related subtasks. In other words, $\delta_i = \max\{|$

$\rho_{i,k} - \rho_{i,j} \mid \| \overrightarrow{jk} \in \mathbf{E}_i\}$. For example, the greatest stage difference of task τ_1 is $3 = \max\{1, 1, 0, 2, 0, 3\}$. The difference between the scheduled stages of two causally related subtasks show how many pipeline stages the two subtasks should be scheduled apart. When task τ_i is the only task in the system, it also means that the system will be idle for δ_i pipeline stages. To better utilize the resource, TPA schedules other independent task instances in these pipeline stages. *Schedule data region* for task τ_i, denoted by $\mathcal{S}_i$, is a data region containing δ_i adjacent data dependency regions.

Last, the algorithm starts to synthesize the pipeline schedule based on the schedule data region of each task. A stack is used to determine the order of processing the data blocks in the schedule data region. Below show the steps for determining the order:

Step 1: Push the core block to the stack.

Step 2: Pop one data block from the stack to be current block:

a) Push the referenced and un-processed data blocks of current block into stack.

b) Schedule the subtask to process current block and mark the block as processed.

Step 3: When the stack is not empty, go to Step 2.

The above steps assure that all the inter-instance precedence constrain are considered. How to schedule the subtasks to process the data blocks in the stack are listed below:

Rule 1: When current block $B_{m,n}$ has no referenced data block:

1) If no subtask $\tau_{i,1}$ is scheduled at stage $\rho_{i,1}$, subtask $\tau_{i,1}$ is scheduled at stage $\rho_{i,1}$. Otherwise, subtask $\tau_{i,1}$ is scheduled at $\rho_{i,1} + \frac{p_i}{l_\pi} \times j$ where j is a minimum integer.
2) The other subtasks of task τ_i are scheduled based on PAT_i accordingly.

Rule 2: When current block $B_{m,n}$ has any referenced data block: Two subtasks $\tau_{i,j}$ and which have inter-instance precedence constraint are selected to schedule. In other words, subtask $\tau_{i,j}$ and $\tau_{i,k}$ of edge $\overrightarrow{jk}$ which has a non-zero precedence vector are selected. We define the *Stage Dependency Function* for subtask $\tau_{i,j}$, denoted by $S(\tau_{i,j}, B_{m,n})$ to calculate the pipeline stage to put the subtask by its source subtasks and the data block. For all $\overrightarrow{jk} \in E_i$, $S(\tau_{i,k}, B_{m,n}) = \max\{S(\tau_{i,j}, B_{m,n})\} + X$ where

$$X = \begin{cases} 0 & \text{if } \alpha_{i,j} = \alpha_{i,k} \text{ and } \beta_{i,j} = \beta_{i,k} \\ \text{Height of PAT} & \text{if } \alpha_{i,j} = \alpha_{i,k} \text{ and } \beta_{i,j} \neq \beta_{i,k} \\ \alpha_{i,k} - \alpha_{i,j} & \text{if } \alpha_{i,j} < \alpha_{i,k} \\ \text{Height of PAT} + (\alpha_{i,k} - \alpha_{i,j}) & \text{if } \alpha_{i,j} > \alpha_{i,k} \end{cases}$$

C. Schedule Refinement Phase

In the last phase, the algorithm tries to shorten the schedule so as to reduce the memory requirement to store the schedule. While synthesizing pipelined schedule, it is not inevitable to have idle time at the start of the schedule. This is mainly caused by the inter-instance precedence constraint. In this step, the algorithm concatenates two generated schedules and shifts the schedule along the time line to have a shorter schedule. The refinement phase terminates when the schedule for any subtasks in two schedules overlap.

Parameter	Range
The number of subtasks	5 ∼ 10
The execution time of subtasks	1 ∼ Period Class
The period class of PCG	10, 20 ,30
The max chain of PCG	4 ∼ 9
The backward referenced edges	1 ∼ 5
The number of labels for each edge	1 ∼ 3
The label value of edges	-3 ∼ 0

TABLE I
THE WORKLOAD PARAMETER OF SYNTHETIC PCGS

IV. PERFORMANCE EVALUATION

A. Performance Metrics and Workload

To evaluate the performance of TPA, we conduct extensive performance evaluations to assure the coverage of experiments. The algorithm is compared against a simulated annealing algorithm, which can find optimal schedule when it runs for a long time, and a greedy heuristic, which can find a schedule in a short time. The greedy algorithm constructs the pipelined schedule by starting at the initial macro-block, and finding its surrounding macro-blocks to construct the pipelined schedule gradually.

Four performance metrics are used to evaluate the performance. *Average processor utilization* is the ratio of occupied processing time to available processing time and is a real number between 0 and 1. *Throughput* is the number of tasks that meet their throughput constraint to that miss their throughput constraints. *Runtime overhead* measures the completion time of the scheduling algorithm. Last, *schedule make-span* is the number of pipeline stages in the synthesized schedule. When the schedule is synthesized off-line for online execution, the length of schedule make-span leads to the required memory space to store the schedule. The shorter the better.

The performance evaluations are conducted against H.264 decoding workload and synthetic workloads to assure the coverage. The decoding applications are simulated over a heterogeneous multi-core platform containing two MPUs and two DSPs. The execution time of the H.264 decoding task was simulated based on the execution time ratio of a H.264 case presented in [12]. In H.264 decoding process, intra predication and inter predication subtask are computation intensive subtask and are mapped to DSPs. Table I lists the parameters for synthesizing PCGs. In our experimental PCGs, there is no deadlock reference edges.

B. Evaluation Results

Figure 6 shows the results for simulating the execution of H.264 decoding task. The results show that TPA schedules the task to meet its throughput constraint as well as optimal exhaustive search algorithm does. The greedy algorithm which conducts the best effort scheduling policy fails to meet the throughput constraint.

The following presents the evaluation results for synthetic workload. In the first part of the results, only the experiments in which all the PCGs have feasible schedules are counted; the experiments in which any of the PCGs do not have feasible schedule are discarded. Figure 7 shows the processor utilization with all PCGs meet their expected throughput. The x-axis is the number of PCGs of the system, and the y-axis is the average processor utilization of the used processors. The

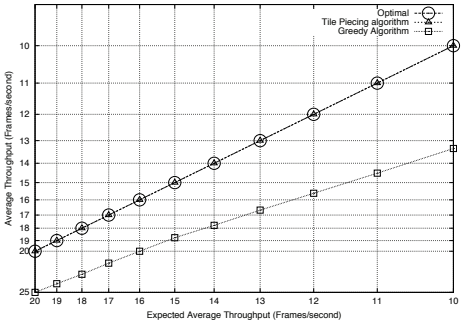

Fig. 6. Throughput for H.264 Decoding Process

results show that TPA can schedule the subtasks to achieve the near optimal utilization. Figure 8 shows the runtime overhead

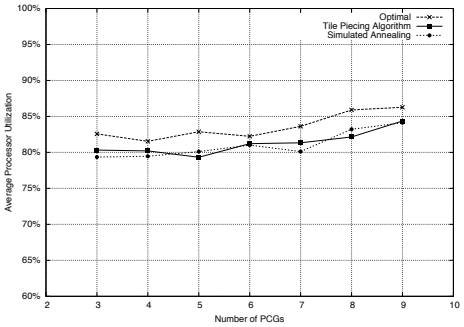

Fig. 7. Average PE Utilizations for the algorithms

of different algorithms. The runtime overhead of optimal case is enormous high, and the computation time for TPA is only one tenth of that of the optimal algorithm. It only takes hundreds of milliseconds to synthesize the schedule, which makes it possible to synthesize the schedule during the run-time.

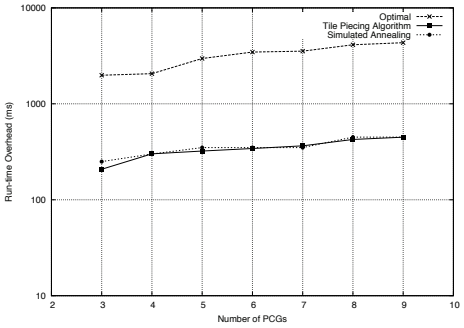

Fig. 8. Average runtime overhead

Figure 9 shows the comparison of make-spans. The x-axis is the umber of PCGs, and the y-axis is the make-span of the pipelined schedule. In Figure 9, we can see the make-span of the pipelined schedule produced by TPA is only one third of that of "Optimal" and "Simulated Annealing Algorithm". In addition, when TPA is used, the make-span does not increase while the number of PCGs increases. When the make-span is shorter, the system needs less memory space to store the schedule.

V. Summary

The number of processing elements on a heterogeneous multi-core platform increase gradually, so the pipeline technique is more and more important on the multi-core platforms. In this paper, we present the developed Tile Piecing Algorithm for synthesizing pipelined schedules. We take the data dependencies of multimedia applications into account, and model the

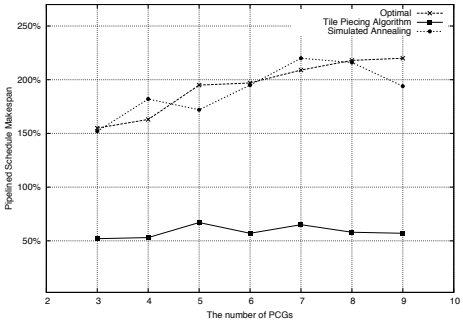

Fig. 9. Average make-span for the algorithms

data dependency relations into the Data Dependency Graph. At the same time, the algroithm considers the heterogeneity of multi-core platforms and allocate the subtasks of applications to the suitable processors. Performance evaluation results show the developed Tile Piecing Algorithm could construct a pipelined schedule with high processor utilization which is closed to optimal solution, and the makespane of pipelined schedules are minimized. The short running time of the algorithm and the lenght of makespan makes it much more practical to synthesize the schedule during the run-time.

Acknowledgment

This work is supported in part by Excellent Research Projects of National Taiwan University, NTU-ERP-99R80304, part by a grant from the Minister of Enconomic Affairs 98-EC-17-A-01-S1-03, and part by a grant from the NSC program NSC 99-2220-E-002-025 and 99-2221-E-002-177-MY3.

References

[1] M. S. Lam, "Software pipelining:an effective scheduling technique for VLIW machines," in *ACM*, SIGPLAN 1988.

[2] B. R. Rau, "Iterative modulo scheduling: an algorithm for software pipelining loops," in *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994, pp. 63 – 74.

[3] V. H. Allan, "Software pipelining," in *ACM Press*, 1995, pp. 367 – 432.

[4] Karam S. Chatha and Ranga Vemuri, "Hardware-software partitioning and pipelined scheduling of transformative applications," in *IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS*, 6 2002.

[5] Wonjong Kim, June-Young Chang, and Hanjin Cho, "Pipelined scheduling of functional hw/sw modules for platform-based soc design," in *ETRI Journal*, Oct 2005, pp. 533 – 538.

[6] Liang-Fang Chao, Andrea S. LaPaugh, and Edwin Hsing-Mean Sha, "Rotation scheduling: A loop pipelining algorithm," in *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, 3 1997.

[7] Cheng-Tsung Hwang, Yu-Chin Hsu, and Youn-Long Lin, "Scheduling for functional pipelining and loop winding," in *The 28th ACM/IEEE Design Automation Conference*, 1992, pp. 764 – 769.

[8] NOHBYUNG PARK and ALICE c . PARKER, "Sehwa: A software package for synthesis of pipelines from behavioral specifications," in *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN.*, 1988.

[9] M. Ikeda, T. Kondo, and K. Nitta, "SUPERENC:MPEG-2 video encoder chip," in *IEEE Press*, 8 1999.

[10] Seong-Min Kim and Ju-Hyun Park, "Hardware-software implementation of MPEG-4 video codec," in *ETRI Journal*, December 2003, pp. 11–24.

[11] Po-Kuan Huang, Matin Hashemi, and Soheil Ghiasi, "Joint throughput and energy optimization for pipelined execution of embedded streaming applications," in *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, San Diego, California, USA, 2007, pp. 137–139.

[12] Xiaoli Zhao, Pin Tao, Shiqiang Yang, and Fei Kong, "Computation offloading for h.264 video encoder on mobile devices," *IMACS Multiconference on "Computational Engineering in Systems Applications"(CESA)*, October 4-6 2006.