Synthesis Algorithm for Application-Specific Homogeneous Processor Networks

Jason Cong, *Fellow, IEEE*, Karthik Gururaj, Guoling Han, and Wei Jiang

***Abstract*—The application-specific multiprocessor system-on-a-chip is a promising design alternative because of its high degree of flexibility, short development time, and potentially high performance attributed to application-specific optimizations. However, designing an optimal application-specific multiprocessor system is still challenging because there are a number of important metrics, such as throughput, latency, and resource usage, which need to be explored and optimized. This paper addresses the problem of synthesizing an application-specific multiprocessor system for stream-oriented embedded applications to minimize system latency under the throughput constraint. We employ a novel framework for this problem, similar to that of technology mapping in the logic synthesis domain, and develop a set of efficient algorithms, including labeling and clustering for efficient generation of the multiprocessor architecture with application-specific optimized latency. Specifically, the result of our algorithm is latency-optimal for directed acyclic task graphs. Application of our approach to the Motion JPEG example on Xilinx's Virtex II Pro platform FPGA shows interesting design tradeoffs.**

***Index Terms*—Clustering, design space, labeling, multiprocessor, task-level pipeline.**

I. Introduction

THE costs and technical challenges of designing application-specific integrated circuits (ASIC) have increased substantially as we move into nanometer technologies. At the same time, the exponential increase in silicon capacity has made it possible to integrate multiple processors in a single chip. This has fueled great interest among microprocessor manufacturers such as Intel [29] and AMD [30] and has led to the development of general-purpose computers with multiple cores on a single chip. In fact, a large IC (over a billion transistors) in today's CMOS technology (e.g., 45 nm) can easily hold hundreds to thousands of simple cores (of the complexity 100 K to 1 M transistors[1]). Therefore, it is possible to use a complex network of customized processors for efficient implementation of a specific application. With this goal in mind, we introduce the concept of application-specific processor network (ASPN). In general, an ASPN consists of a set of processor cores chosen from a library of predesigned processors, memories, communication channels, and peripherals connected in an application specific manner, to provide just enough computing power to satisfy the performance and power requirements of the given application and thus saving the energy and implementation cost as much as possible. The IBM Cell processor [16] may be viewed as an ASPN targeted towards gaming applications. Effectively, we are extending the current standard-cell based ASIC design methodology to application-specific processor-based design methodology—instead of synthesizing a RTL design into a network of standard cells, we now map a system-level description (often specified in C, C++, or SystemC) into a network of processors. ASPN-based design methodology raises the level of design abstraction, thus reducing the design complexity. Since most of the hardware details are abstracted out by the ISA of the processors, developers, especially the software programmers, can comfortably work with the programming languages and environments that they are familiar with. The smooth transition enables higher productivity and significantly shortens the time-to-market period. ASPN-based design methodology also allows reuse of the processor library. Additionally, it provides greater flexibility, both during design time as well as at deployment time. Obviously, there is great similarity between the ASPN concept and the MPSoC design style which has been practiced recently [5], [9]. However, in today's MPSoCs, there is typically one or several of embedded processors, determined *a priori*, together with a large amount of customized logic, while we emphasize a processor-centric methodology, where the number, type, and configuration of the processors are determined at the synthesis time.

For an ASPN-based design, designers need to carefully partition tasks, allocate resources, map tasks to the processing elements, and build application-specific communication networks. Obviously, the huge design space makes exploration extremely difficult for human designers. Currently, designers must iteratively go through the partitioning, mapping and simulation in order to find the optimal architecture for the applications. This process is time-consuming, tedious and error-prone.

To ease these challenges, our work is to develop a systematic approach for ASPN synthesis. Our aim is to build an automatic exploration tool to help designers construct the optimal architectures and mapping solutions. Our system targets throughput-constrained, stream-oriented applications, such as multimedia and network applications, for which the computation and communication time can be obtained easily in advance. Since most of these applications have requirements on stream throughput (e.g., 30 frames/s for MPEG decoding), the multiprocessor systems should provide just enough performance

Manuscript received October 26, 2007; revised March 15, 2008. First published March 16, 2009; current version published August 19, 2009. This work was supported in part by the MARCO/DARPA Gigascale Silicon Research Center (GSRC), in part by the National Science Foundation under Grants CNS 0647442 and CNS-0725354, and in part by a grant from Xilinx, Inc., under the California MICRO program.

The authors are with the Computer Science Department, University of California, Los Angeles, CA 90095 USA (e-mail: cong@cs.ucla.edu; karthikg@cs.ucla.edu; leohgl@cs.ucla.edu; wjiang@cs.ucla.edu.).

Digital Object Identifier 10.1109/TVLSI.2008.2004874

[1]For example, the first version of the Intel 486 Processor has about 1 M transistors [29].

to satisfy the required throughput. Meanwhile, it is desirable to minimize the application latency under the throughput constraint, i.e., the time elapsed to process an individual data set. For instance, video conference applications always prefer a small latency to allow real-time conversation. Therefore, our ASPN synthesis system optimizes the application latency with a given throughput constraint. Given the application specified as a task graph, we construct a network of homogeneous processors connected by point-to-point FIFOs. Our exploration tool automatically decides the number of processors used in the system, communication buffer sizes, and processor interconnections, as well as the mapping of tasks onto the synthesized system.

The remainder of the paper is structured as follows. We first discuss related works on multiprocessor design space exploration and mapping in Section II. Section III describes the multiprocessor architecture model used in our system and problem formulation. The synthesis flow and algorithm is presented in Sections IV and V. We have applied our exploration tool to a set of task graphs generated by TGFF [3] and the motion JPEG application under various design constraints. The exploration results for the motion JPEG on Xilinx FPGA boards are presented in Section VI. We conclude the work with future directions in Section VII. The preliminary results of this work have been published in [2]. We have extended the work by allowing a more general pipeline execution model to achieve better performance (see Section V).

II. Related Works

Over the past two decades, there have been other efforts that involved solving the multiprocessor design space exploration problem. The previous work can be categorized into two groups based on their optimization objectives, i.e., optimizing latency or throughput. The work in [20] proposes a compile time scheduling and clustering technique to minimize the parallel execution time. Given a task graph, it first iteratively merges the best pair of blocks greedily to minimize the critical path length. After a partition is obtained, it applies list scheduling to schedule the blocks onto processors. In the SOS system [17], the multiprocessor synthesis and task mapping problem is solved by creating a mixed-integer linear programming model (MILP) to minimize the system latency. The model is composed of a set of relations that ensures proper ordering of various events in the task execution, as well as completeness and correctness of the system. The work in [25] proposed a heuristic solution to synthesize a heterogeneous multiprocessor system. The objective is to minimize the hardware cost under latency constraint. After obtaining the initial solution, their algorithm iteratively reallocates tasks and processing elements to minimize cost. On the throughput optimization side, Hoang [8] proposes a heuristic approach to schedule a DSP program onto multiprocessors for maximizing system throughput. It applies a modified list scheduling to schedule the tasks onto a given number of processors. MOGAC [4] synthesizes real-time heterogeneous multiprocessor architectures using an adaptive multi-objective genetic algorithm. Similarly, Grajcar [7] uses a genetic algorithm based on list scheduling to minimize the makespan on a bus-based multiprocessor. The work in [11] adopts modulo scheduling to map programs onto a pipeline of multiprocessors. In this architecture, multiprocessors are organized in a pipelined fashion where each stage processes a subset of the tasks for system throughput improvement. Pipelining, a commonly used technique for both hardware and software design, has been extensively used at instruction-level and loop-level previously. [11] and our work have applied the concept to task-level for multiprocessor designs. In the rest of the paper, we will use pipeline to represent task-level pipelining for the purpose of simplicity. More recently, Jin [10] formulates the task mapping with a fixed number of processors as an integer linear programming problem (ILP). Due to the complexity of the ILP, it would be difficult for this approach to scale to larger problem sizes. [22] is the only work that optimizes latency in the presence of a throughput constraint. However, they can only handle programs simply composed of a chain of tasks. In contrast, our proposed algorithm can take general directed-acyclic task graph as input and provide latency optimal solutions under given throughput constrains.

Our contributions in this work are twofold. First, it is the first work proposed that considers the throughput and latency simultaneously for general task graphs. Second, we provide a complete synthesis flow for the design space exploration of an ASPN system. It includes efficient labeling, scheduling and clustering algorithms which take the inter-processor communication cost into consideration. The proposed synthesis algorithms generate latency-optimal results for acyclic graphs subject to throughput constraint. They can also be easily extended to cyclic data flow graphs. We demonstrate our methods and tools by running real-life applications on the Xilinx FPGA platform, which adds more credibility compared to the pure simulation methods.

III. Problem Statement

A. System Model

Dataflow process network model [14] is commonly used in designing and implementing signal processing applications. Especially, the synchronous dataflow model [13] in which the rates at which data is produced and consumed are constants for every dataflow edge, is widely adopted because bounds on the memory requirement and deadlock-free execution order can be determined statically in finite time. Since any synchronous dataflow graph can be transformed to a homogeneous task graph [21] where all the tasks are executed at the same rate, we will assume that the application is represented as a periodic task graph, which is executed repeatedly to process the incoming data streams, in our system. The task graph is a directed graph $G(V, E)$ in which each node represents a task and each edge corresponds to the data communication between the tasks. Each edge $e(u, v)$ in the task graph will be mapped to a physical communication channel if task u and task v are mapped to different processors. Otherwise, they are executed on the same processor and just use the memory to communicate data.

Each vertex in the task graph is annotated with the estimated task computation time. The computation time is measured by profiling the task on an architecture simulator or a real processor. Note that some tasks may have variable execution time caused by different execution paths and memory behaviors, we will use the worst case time (i.e., maximal execution time) in

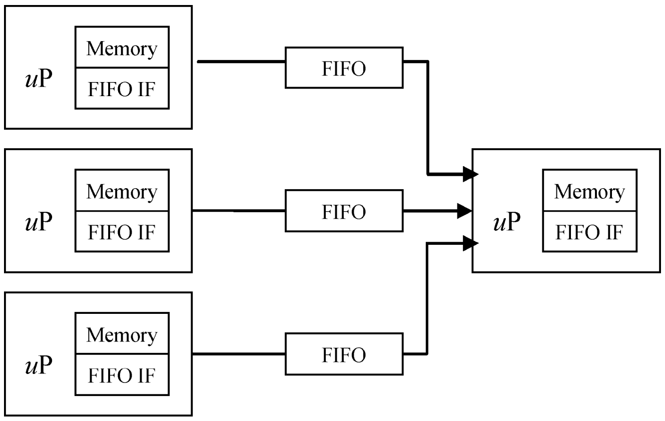

Fig. 1. Example of homogeneous processor network.

order to guarantee the hard-constrained throughput requirement. Each edge in the task graph has an attached attribute: volume (vl). The communication volume specifies the amount of data that need to be transferred for one execution. The communication volume can be obtained from profiling or user specification. If the sending task and the receiving task are mapped to the same processor, we only need internal memory to pass data so that the inter-processor communication is avoided. If they are mapped to different processors, the inter-processor communication, which is usually much slower than the intra-processor communication, is required. The inter-processor communication also requires memory storage to buffer the incoming data at the destination side. The buffer requirement of edge $e(u,v)$ on a processor p is determined by the communication volume and the distance of the pipeline stages between the source and destination tasks. We denote the pipeline stage where the task u is executed as stage(u). The distance of two pipeline stages (u and v) is defined as $|\text{stage}(u)-\text{stage}(v)|$. Scheduling a task v on processor p requires that buffers hold data from all of the input edges. Thus, the buffer requirement on destination processor p running task v, denoted as buffer(v,p), is

$$\text{buffer}(v,p)=\sum_{e(u,v)\in\text{input}(v)}(vl(e)\times|\text{stage}(v)-\text{stage}(u)|).$$

In our work, we target the architecture template of homogeneous processors connected by point-to-point FIFOs (as illustrated by an example in Fig. 1). Each microprocessor has its own local memory to store all the instructions and data. The inter-task communication and synchronization are done by means of FIFOs. If inter-processor communication is required, the sender must explicitly send data to the receiver through FIFOs. The FIFO sizes are determined by the calculation of buffer(v,p). The execution of tasks will be blocked if they read from an empty FIFO or write to a full FIFO. Since inter-processor communication needs the processors to transmit and receive data, it also brings communication delay. The synthesis tool can use any communication model to calculate the communication costs. In this paper, we use a linear model [6] in our work to model the FIFO communication cost. Specifically, if the startup cost to initiate a data transfer is C_{init}, and the cost to transfer one data unit is C_{unit}, the estimated communication cost will be calculated as

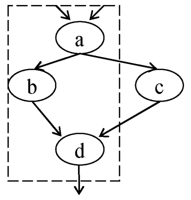

Fig. 2. Example of non-convex cluster.

$\text{Comm}(e)=C_{\text{init}}+vl(e)\times C_{\text{unit}}$. Although other more complicated communication models could be easily plugged in, this simple model is accurate enough to capture the point-to-point FIFO communications. Actually, our synthesis algorithm will not depend on this communication model, and it is quite easy to replace it with models that reflect the target architecture.

B. Homogeneous ASPN Synthesis Problem

Before we present the formal problem formulation, we define some notations used in this paper.

- The **stage period *T*** is defined as the reciprocal of the throughput. When we cluster a set of tasks into one pipeline stage on one processor, their total execution time should not be larger than T. The total execution time is defined as the sum of their computation time and communication time.
- In a pipelined multiprocessor system, the **application latency** is defined as the elapsed time from the data input streams to the output streams. Since the stage period is fixed with a given throughput constraint, the latency is determined only by the difference between the input stage and output stage.
- A task cluster S is **convex** if there exists no path from a node u in S to another node v in S and involves a node w that is not in the cluster. Fig. 2 shows an example of a non-convex cluster. Node c receives data from the cluster while it also sends data back to the cluster.

We formulate our ASPN synthesis problem as follows.

1) Homogeneous ASPN Synthesis Problem: Given a task graph $G(V,E)$ with profiling information, user-specified throughput constraint, construct a homogeneous ASPN system by 1) partitioning the tasks into convex clusters, and 2) mapping the clusters onto the processors and inter-processor communication to FIFOs so that the application latency is minimized under the constraint that the required throughput can be satisfied.

The convex constraint is to ensure the existence of a feasible scheduling. Since we assume that the tasks in a cluster will be executed at the same pipeline stage, correct scheduling may not exist to ensure the data dependency if the cluster is not convex. For the example in Fig. 2, the pipeline stage of node c should be after the pipeline stage of $\{a,b,d\}$ due to the edge (a,c). However, edge (c,d) indicates a reversed order, which generates a contradiction. In addition to the convex constraint, the synthesis result should also honor the data dependency relationship in the task graph. If task u sends data to task v, the consumer task v cannot execute until all the data are available. Thus, if two adjacent tasks are clustered together and scheduled at the same pipeline stage, their execution order within the processor

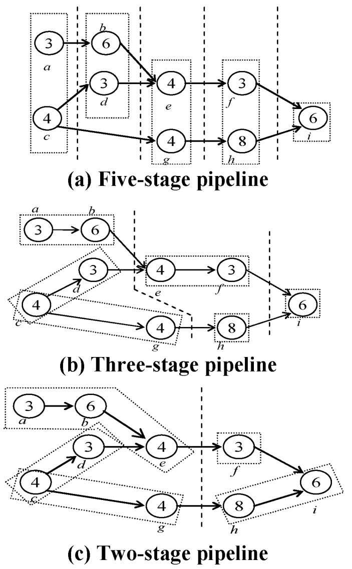

Fig. 3. Pipelined execution example.

should ensure that u executes first. If they are mapped to different pipeline stages, the data producer should be scheduled at the prior data consumer stage.

Fig. 3 shows an example with nine tasks. For this example, we assume that all the tasks have a fixed execution time. The nodes are labeled with the computation times (in μs). In this example, for any pair of tasks connected by an edge, the communication time is assumed to be one if the two tasks are mapped to different processors. Executed on a single processor, the system cannot process new inputs until all the tasks have been executed once. Thus, the stage period is 41 μs. We assume that we wish to obtain a stage period of 14 μs. By exploiting the task-level parallelism and pipelining the tasks, we can improve the throughput. Fig. 3(a) shows a naive five-stage pipeline implementation with nine processors. Note that the tasks in each dotted box are mapped onto one processor in this figure, and the dotted lines represent pipeline stages. We can find that on each processor the total execution time, including the communication time to transmit data, is no more than 9 μs. Therefore, we could achieve a stage period of 14 μs and a 3.4 X throughput improvement. Fig. 3(b) shows another pipeline implementation with six processors. It can also achieve the stage period of 14 μs. However, the latency has been reduced to three stages. Actually, the optimal solution, as shown in Fig. 3(c), only needs two pipeline stages and five processors. Although the three implementations achieve the same system throughput, their latencies and required processors could differ by a factor of 1.67 and 2.5, respectively. This example illustrates the possible huge exploration space in multiprocessor synthesis and the importance of optimizing the throughput and latency at the same time.

The previous example (which we will use throughout the remainder of this paper) demonstrates a simple pipeline execution model, called sequential execution model, where no communication is allowed between processors within the same pipeline stage. Alternatively, if we have a loose stage period constraint and the task graph contains enough task-level parallelism, it is possible to construct a parallel implementation for each pipeline stage. By allowing inter-processor communication within a pipeline stage, we can take full advantage of the task-level parallelism and achieve better latency. In the rest of this paper, we will call it parallel execution model.

Given the task graph, profiling information, and the design constraints, our algorithm, called ARMS (Application-specific, Rate-constrained Multiprocessor Synthesis), solves the multiprocessor synthesis problem in three steps. The first step, called stage period checking, calculates the theoretic lower bound of the stage period. The next two steps, task labeling and clustering are then performed to obtain a latency-optimized solution. With the synthesized result, we finally generate a hardware configuration for the underlying platform and corresponding software program. Note that the synthesis algorithms only differ at the labeling step for the two aforementioned models. To differentiate them, we call the algorithm ARMS-S and ARMS-P for the sequential execution and the parallel execution respectively.

IV. ARMS-S

A. Stage Period Checking

The stage period T is the total time budget to execute tasks on one processor in one pipeline stage. In a pipelined implementation, data will iteratively enter the system and be processed every T time units. Since a computation node can be scheduled onto only one processor, the minimal stage period should be no less than the computation time of any task node in the graph, i.e.,

$$T \geq \mathrm{ExecTime}(v)\ \forall v. \tag{1}$$

The users should decompose those tasks with the largest computation time if the desired throughput cannot be achieved. Usually, decomposition might incur more communication. In order to balance the computation and communication cost, we assume that the execution time of a task should be no less than the communication time, i.e.,

$$\mathrm{ExecTime}(p) \geq \sum_{r} \mathrm{CommTime}(p \rightarrow r)\, \forall r \in \mathrm{Successors}(p). \tag{2}$$

Otherwise, it may not be beneficial to decompose a task further since the communication cost will be larger than the gain from task decomposition. This assumption is used throughout this paper.

B. Labeling

In the following description, we will first assume that the task graph is a directed acyclic graph (DAG) and present a latency-optimal labeling and clustering algorithm. We shall then discuss the extension of the algorithm to cyclic graphs.

We observe that the problem of finding the minimum latency clustering for multiprocessor synthesis is analogous to the problem of circuit clustering for delay minimization in VLSI physical design, which has been well investigated [12], [18], [1]. Circuit clustering problem tries to minimize the longest path delay subject to the cluster capacity constraint. A simplified general delay model, which assumes a uniform

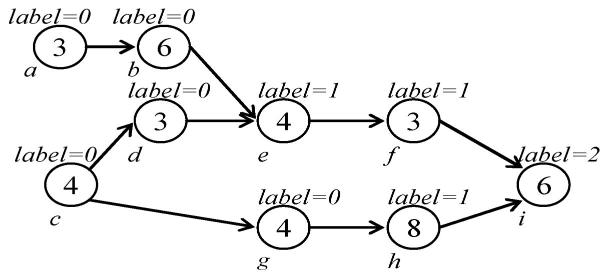

Fig. 4. Labels of a task graph.

interconnect delay, is usually used. In multiprocessor synthesis, the inter-processor communication cost may vary significantly. Therefore, our labeling algorithm should take the nonuniform communication cost into consideration. In addition, the circuit clustering assumes that each cluster has a limited number of inputs and outputs. This is not necessarily true in the multiprocessor synthesis scenario.

For a given directed acyclic task graph $G(V, E)$, let N_v be the subgraph consisting of the node v and its predecessors in G. We define the node label as the earliest pipeline stage where v can be executed in an optimal clustering of N_v. We can see that for a node v, the minimum label in any clustering of G is at least the label obtained in the N_v. This can be derived from the fact that any cluster in G induces a cluster in N_v. In addition, we have the following observation.

Lemma 1: The total execution time of a cluster increases monotonically when adding more predecessors into the cluster.

Let M_v be the set of nodes in the predecessors of v which have the maximal label L_v, the minimum label of v can be obtained based on the following lemma.

Lemma 2: The label of node v exceeds L_v by at most 1.

The labeling algorithm, which is similar to Lawler's algorithm [12], assigns labels to each node of the graph in a topological order. Each primary input (PI) node, i.e., that node does not have any predecessors, can be executed at the first pipeline stage. Therefore, we assign label 0 to it. Since labeling is done in a topological order, a node v is not processed until all of its predecessors have been labeled. We just need to sort the predecessors in the nondecreasing order based on the value of their labels. Then we determine whether those nodes with the greatest label value could be clustered together with node v, and set v's label accordingly based on Lemma 2.

Fig. 4 shows a task graph with the labeling result. We assume that each edge has one unit of communication cost, except edge (c, g), whose communication cost is two. The stage period constraint is 14 time units. The labels of PIs are set to be 0. Since node a and b can be clustered together with a total execution time of 10, b's label is 0 as well. Similarly, we set the label of node d to be 0. Now we proceed to label e. Since the total execution time of a, b, c, d and e is 23, which is the sum of the computation time of the tasks $\{\text{a, b, c, d, e}\}$ and communication cost 3. We cannot accommodate all the nodes in a single cluster. Therefore, the label of e is 1.

When the ARMS-S algorithm labels a task v, it looks at all the predecessors of v which have the highest label. Suppose k tasks are labeled before task v is considered. Since we proceed in topological order, in the worst case, all the k tasks are the predecessors of v and all have the same label. Thus, in the worst case, for each task to be labeled, all the tasks labeled in previous steps need to be considered. Hence, the worst case complexity of ARMS-S is given by $O(\sum_{i=1}^{n-1} i) = O(n(n-1)/2) = O(n^2)$ where n is the number of tasks in the task graph.

C. Clustering

In this step, a latency-optimal clustering of the task graph will be generated using the labels and clusters computed in the previous phase. We maintain a list L of nodes, which are the roots of the clusters in the optimal solution. Initially, we put all the primary output (PO) nodes, i.e., those nodes do not have any successors, on the list L. Each time we take a node from the list and generate a cluster rooted at the node, which includes all its predecessors with the same label, the immediate input nodes to this cluster are pushed to the list L. Then, this process is repeated until L is empty. For the example shown in Fig. 4, we first put the primary output node i into the list L. Since its label is different from the labels of all its inputs, we generate a cluster consisting of node i only. Then, the node i's inputs (f and h) are put to L and processed similarly. Fig. 3(b) shows the final clustering solution.

Based on Lemma 1 and 2, we calculate the minimal label for each node by traversing the graph in topological order. The clustering step will generate latency optimal solution finally. Therefore, we can prove the following theorem.

Theorem 1: The labeling and clustering algorithms generate latency-optimal pipeline solutions for directed acyclic task graphs in polynomial time.

D. Extension to Cyclic Graphs

The labeling and clustering algorithms work well for DAG. We need to enhance the algorithm to handle those cyclic data flow graphs. A loop in a data flow task graph represents the inter-iteration data dependency. The tasks in a loop require data generated from the previous iterations. This data dependency is specified by the *dependence distance*, which is defined as the difference between the iteration instances of the target and source iterations [24]. The inter-iteration dependency will limit the minimum achievable stage period. Consider the example shown in Fig. 5(a). There are three tasks, each with execution time of 10 μs. The first task performs the calculation with the new input data and the data generated from the previous iteration. Thus, the dependence distance for the back edge is one as annotated. We can see that any pipeline partition cannot satisfy the inter-iteration data dependency. In general, suppose that the sum of the dependence distance along a cycle is D, the cycle cannot be partitioned into more than D stages [8].

Let G be a task graph and let C be a cycle in G. By removing the back edge, we can follow the same routine to obtain a clustering solution. Suppose that the sum of the dependence distance along cycle C is D, the labeling result is valid if and only if

$$\max_{\forall v \in C}(\text{label}(v)) - \min_{\forall u \in C}(\text{label}(u)) \leq D - 1.$$

This condition ensures that the inter-iteration data dependency is correctly maintained. For the example shown in Fig. 5(a), all of the labels along that cycle should be the same. If the dependence distance on the back edge is two as shown in

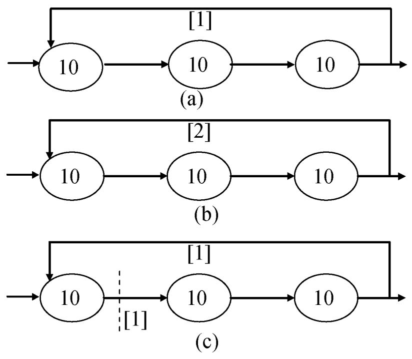

Fig. 5. Cyclic task graph and its pipeline partition.

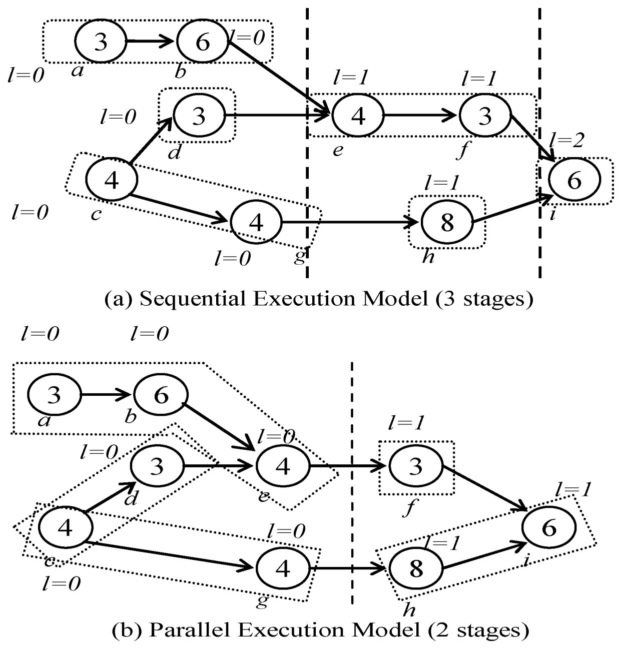
l=0
a
b
l=1
l=2
c
d
e
f
g
h
i
(a) Sequential Execution Model (3 stages)
(b) Parallel Execution Model (2 stages)

Fig. 6. Optimal clustering solution.

Fig. 5(b), we can make a partition on the forward path as shown in Fig. 5(c). This can be viewed as moving the dependence distance around to maximize throughput. For cyclic graphs, the inequality is checked for every cycle after labeling. If it cannot be satisfied, the throughput constraint needs to be reduced in our system, and we perform binary search to find the optimal throughput.

E. Packing

The labeling and clustering algorithm generates optimized clustering for latency minimization. Each cluster of tasks occupies one processor, and its total execution time can be less than the stage period. Therefore, some processors may not have been fully utilized. The packing phase tries to reduce the used processors by merging the clusters while maintaining the throughput. Unfortunately, this packing problem is NP-complete since the bin-packing problem, known to be NP-complete, can be reduced to it in polynomial time. In our work, a simple yet efficient heuristic, namely first fit decreasing, is used. We sort the clusters in decreasing order according to the value of their total execution time, and then pack the clusters following the sorted order. For each cluster, the algorithm searches the clusters in front of it to find the largest one possible to pack with it, if their total execution time does not exceed the stage period. We repeat this procedure until all the clusters have been processed once.

V. ARMS-P

The relaxation of sequential execution model per pipeline stage may help us to get ASPN systems with lower application latency. Continuing to use our example in Fig. 4, we observe that the optimal application latency is 3 pipeline stages as shown in Fig. 6(a) if the sequential model is used. However, the optimal application latency can be reduced to 2 pipeline stages as shown in Fig. 6(b). Let us look at node e in the parallel execution model. In this clustering solution, inputs of e are executed in parallel on two processors. The finish times for task b and d are 9 and 7, respectively. Taking the communication time from d to e into account, the starting time of e is 9 and its finish time is 13. Therefore, we can label node e as 0.

Let us first define the notations used in the following sections.

- The **ready/release** time for a task in cluster C is defined as the minimum time at which the task can begin execution in cluster C. The ready time for an input i of a cluster C, denoted by **ReadyTime(i)**, is defined as the minimum time at which the input data from i arrives at cluster C. This includes the communication time if node i and cluster C are mapped to different processors. The ready time of a task is the maximum ready time among all its inputs.
- The minimum finish time **MinTime(v)** is defined as the minimum time at which task v completes execution. For instance, MinTime(e) is 13 for the previous example.
- **CommTime($i \rightarrow v$)** is defined as the communication time between node i and its successor v.
- **ExecTime(v)** is defined as the execution time of node v.
- **FinishTime(C)** is defined as the completion time for cluster C according to the optimal schedule for C.

A. Complexity of Parallel Execution Model

The parallel execution model is significantly more complicated because inherently we need to solve a multiprocessor scheduling subproblem for each pipeline stage.

To ensure that a node v finishes execution in pipeline stage k, we need to ensure that node v and all of its predecessors executing on stage k can finish execution by time T where T is the stage period specified in the problem. We observe that the problem we are tackling is at least as hard as multiprocessor scheduling problem for task graphs with precedence constraints:

Given a set of tasks $S = \{s_1, s_2, \ldots, s_n\}$ with precedence constraints and identical execution times of one time unit each, map the tasks to an arbitrary number of identical processors and schedule them such that the total makespan of set S is less than a specified time T'.

This scheduling problem has been proved to be NP-Hard [23]. Hence, we have the following lemma.

Lemma 4: The homogeneous ASPN synthesis problem is NP-Hard if the parallel execution model is allowed.

In the sequential execution scenario, all inputs of a cluster were available at the start of the pipeline stage. However, the first thing we observe for the parallel execution model is that

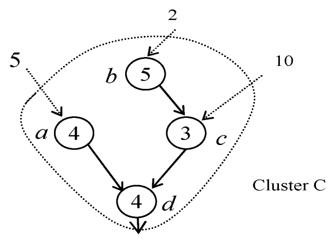

Fig. 7. Scheduling tasks within a cluster.

the finish time of a cluster C depends on the ready times of its inputs and the corresponding scheduling solution. The problem of scheduling tasks on a uniprocessor system to minimize total latency subject to precedence constraints is well studied and an optimal schedule can be found in $O(n^2)$ time (where n is the number of tasks in the cluster) by listing the tasks in nondecreasing order of their release/ready times [15].

For example, consider the cluster $C = \{a, b, c, d\}$ shown in Fig. 7. The number inside each node is the task's execution time. Since we consider a single cluster C, there is no communication cost between the tasks of cluster C. The ready times of the inputs of C are 5, 2, and 10 as shown in the figure. We determine the ready times of tasks a, b, c, and d to be 5, 2, 10, and 13, respectively. Thus, the optimal scheduling of tasks is given by $b \rightarrow a \rightarrow c \rightarrow d$ with which cluster C finishes execution at time 18. Note, that we had to choose whether task b or a will be the first to be executed. However, since the ready time of a is larger than that of b, scheduling task a first would lead to a schedule with which execution of C finishes at time 21.

We also observe that the MinTime of each task needs to be stored apart from its label. This time is then used to compute the ready time of its successors and to compute the ready time for any cluster that has the current task as one of its inputs. Thus, for each task v, we store a tuple $\langle L, \mathrm{MinTime}(v)\rangle$ where L is the label of v. Note that we still only consider convex clusters for parallel execution model. Next a branch and bound algorithm and effective pruning techniques will be presented.

B. Bounds of MinTime

Our clustering approach computes the minimum finish time of tasks in topological order. Interestingly, we find that the lower and upper bounds of $\mathrm{MinTime}(v)$ can be calculated based on the MinTime values of its inputs, which is useful for pruning the search space.

The lower bound of $\mathrm{MinTime}(v)$ is

$$\mathrm{LowerBound} = \max_{\forall i \in \mathrm{inputs}(v)} (\mathrm{MinTime}(i)) + \mathrm{ExecTime}(v). \quad (3)$$

Proof: Consider the task v shown in Fig. 8 with its inputs $S = \{i | i \in \mathrm{inputs}(v)\}$. The minimum finish time of the input i of v is given by $\mathrm{MinTime}(i)$. Task v cannot start execution until all it inputs are ready. Hence, task v cannot begin execution earlier than $\mathrm{MinTime}(i)$. So, the lower bound on the $\mathrm{MinTime}(v)$ is given by (3). □

The upper bound of $\mathrm{MinTime}(v)$ is

$$\mathrm{UpperBound} = \max_{\forall i \in \mathrm{inputs}(C)} (\mathrm{MinTime}(i) + \mathrm{Comm}(i \rightarrow v)) + \mathrm{ExecTime}(v). \quad (4)$$

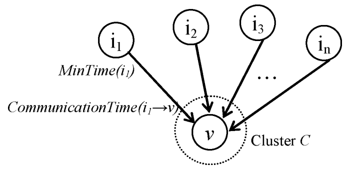

Fig. 8. Example of bounds calculation.

Algorithm 1: ***ARMS-P***

```
Initialize <L, MinTime> of all tasks with only primary inputs to <0, 0> and for other tasks to <∞, ∞>
Initialize $C_{list} \leftarrow f$
$S \leftarrow$ Topologically ordered set of tasks
while S is not empty do
  $v \leftarrow$ Extract next task from S
  $L \leftarrow \max_u \{L_u | u$ is an immediate predecessor of $v\}$
  Obtain LowerBound for MinTime(v)
  Initialize cluster $C \leftarrow \{v\}$
  if LowerBound violates time period constraint then
    /*Move task v to next pipeline stage*/
    $L_v \leftarrow L + 1$
    MinTime(v) $\leftarrow$ ExecTime(v) + L*TimePeriod
    $C_v \leftarrow C$
  else
    Initialize $M_v$
    Order tasks in $M_v$ in reverse topological order
    Determine optimal schedule of C
    /*Find the best among all clusters rooted at v*/
    <MinTime(v), $C_v$> $\leftarrow$ Min(FindMinTime(C, v), <FinishTime(C), C>)
    if MinTime(v) violates time period constraint then
      /*Move task v to next pipeline stage*/
      $L_v \leftarrow L + 1$
      MinTime(v) $\leftarrow$ ExecTime(v) + L*TimePeriod
      $C_v \leftarrow C$
    else
      $L_v \leftarrow L$
    end if
  end if
  Add $C_v$ to list $C_{list}$
end while
Select clusters for tasks in reverse topological order
```

Fig. 9. ARMS-P algorithm.

Proof: Consider the task v shown in Fig. 8. Now, suppose we create a new cluster C consisting only of task v. The ready time for input i of the cluster C is given by $(\mathrm{MinTime}(i) + \mathrm{Comm}(i \rightarrow v))$. Hence, node v cannot begin execution earlier than $t = \underset{\forall i}{Max}(\mathrm{MinTime}(i) + \mathrm{Comm}(i \rightarrow v))$. Thus, we have obtained a cluster C rooted at task v which can finish execution in time $t'_c = t + \mathrm{ExecTime}(v)$. Since $\mathrm{MinTime}(v) \leq \mathrm{FinishTime}(C')$ for all clusters C' rooted at v, t'_C, which is the same as the right hand side in (4), is an upper bound on $\mathrm{MinTime}(v)$. □

C. Branch and Bound Algorithm

We now present the branch and bound algorithm for clustering tasks to obtain a solution with optimal latency. In our algorithm, shown in Fig. 9, we compute the values of the tuple $\langle L,\ \mathrm{MinTime}\rangle$ of the tasks in topological order. Because we label the node in topological order, the values of $\langle \mathrm{L},\ \mathrm{MinTime}\rangle$ for all inputs of v have been computed already.

Algorithm 2: *FindMinTime(C, v)*

```
    Input:
     A cluster $C$ rooted at $v$
    Return:
     A tuple $<time, C_{min}>$ such that:
5:  $C$ is a sub-cluster of $C_{min}$ $C_{min}$ is convex
    $C_{min} = \arg\min_{C_i} FinishTime(Ci) \{C_i \mid C \text{ is a sub-cluster of } C_i\}$
     $time = FinishTime(C_{min})$
10: Algorithm
    $p \leftarrow$ Extract next task from $M_v$
    $C' \leftarrow C + p$
    $mintime_1 \leftarrow +\infty$     $mintime_0 \leftarrow +\infty$
15: if $C'$ is convex then
     if In cluster $C'$ $FinishTime(p) < ReadyTime(p)$ for input $p$ of
     cluster $C$ then
       /*$C'$ may be better than $C$ - Find its optimal schedule*/
       Determine optimal schedule for $C'$
       $mintime_0 \leftarrow FinishTime(C')$
20:    end if
     /*$C_1$ is the best among all clusters which have $C$ as sub-cluster
     and include $p$*/
     $<minTime_1, C_1> \leftarrow FindMinTime(C', v)$
    end if
25: /*$C_2$ is the best among all clusters which have $C$ as sub-
    cluster but do not include $p$ */
    $<mintime_2, C_2> \leftarrow FindMinTime(C, v)$
    $<mintime, C_{min}> \leftarrow Min(<mintime_0, C'>, <mintime_1, C_1>, <mintime_2, C_2>)$
    Push $p$ to the front of $M_v$
30: return $<mintime, C_{min}>$
```

Fig. 10. FindMinTime function.

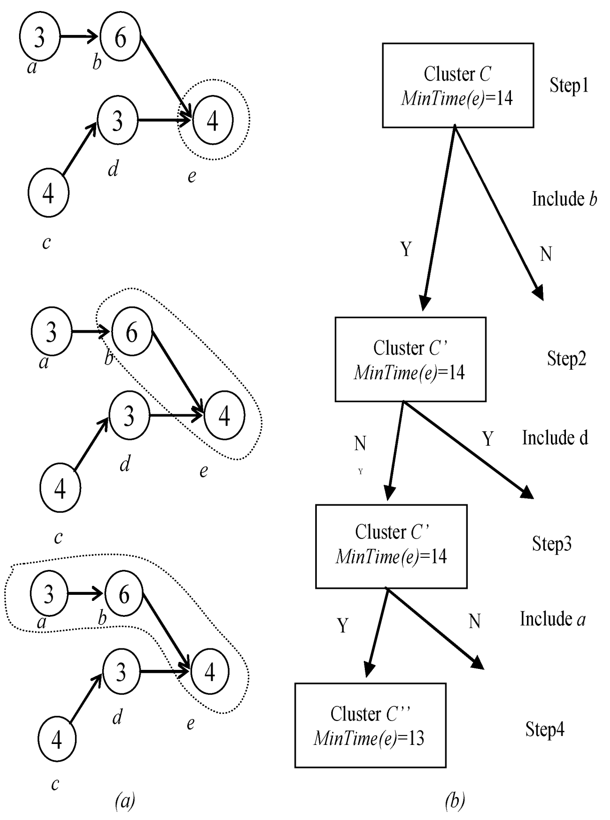

Fig. 11. Example of decision tree.

When considering task v, we construct a decision tree for enumerating a series of convex clusters rooted at node v consisting of nodes in the set M_v. The tasks in M_v are placed in a list l in reverse topological order. Initially, the cluster C consists of a single task v. Each node of the decision tree takes the next task p in l and decides whether to grow cluster C by adding p to C (lines 15–23 in Fig. 10) or to keep C by ignoring p and moving to the next node in the list l (line 27 in Fig. 10).

Let the highest label among all tasks in N_v be L_v. We initialize $\text{MinTime}(v)$ to be its upper bound before the search begins. For each cluster C rooted at v, we first determine the ready time for all of its inputs. This is obtained as follows:

$$\text{ReadyTime}(i) = \text{MinTime}(i) + \text{Comm}(i \to C) \quad i \in \text{inputs}(C). \quad (5)$$

We then compute the minimum ready time for each task in cluster C by traversing the subgraph in topological order. The tasks in C are then scheduled to run in nondecreasing order of their ready times, which gives us an optimal schedule for cluster C (lines 18–19 in Fig. 10). With this schedule, we determine the finish time of cluster C, and hence the finish time of task v. If this finish time is less than the current $\text{MinTime}(v)$, we update $\text{MinTime}(v)$ accordingly.

Using the previous example, we show in Fig. 11 the working of the algorithm when it examines task e in the graph. Part (a) of Fig. 11 shows how the cluster rooted at e grows while part (b) shows what the decision tree looks like. The MinTime of the inputs of e have been computed to be 7 and 9 for d and b, respectively. The lower and upper bounds of $\text{MinTime}(e)$ are 13 and 14, respectively. We create a new cluster $C = \{e\}$, with the $\text{MinTime}(e) = 14$ and start building the decision tree. Task b is first considered and included in cluster C. The new cluster $C' = \{b, e\}$ has an optimal scheduling order of b followed by e. The finish time of cluster C' is 14, which is the same as the current value of $\text{MinTime}(e)$. Suppose, we then consider task a and include it in cluster C' to get a new cluster C''. The finish time of C'' is 13 and hence, we update the value of $\text{MinTime}(e)$. Note that 13 equals to the lower bound of $\text{MinTime}(e)$; therefore, we can terminate the search immediately.

D. Pruning Techniques

The number of clusters rooted at any task v might become unmanageably large if we simply enumerate all convex clusters. Hence, we introduce several effective pruning techniques to achieve acceptable computation times. The pruning techniques can be classified into two kinds.

1) Bound Based: These techniques utilize the bounds computed on the MinTime of a task as well as the stage period constraint given in the problem.

Stage Period-Based Pruning: If the *LowerBound* of v is greater than the stage period T, then there does not exist any cluster rooted at v such that v can be labeled L_v. Hence, we need not enumerate any cluster rooted at v and label v as $\text{L}_v + 1$ straightaway.

Ready Time-Based Pruning: The second pruning technique avoids computing the optimal schedule for a given cluster if it is able to determine that the cluster cannot lead to a better solution. Consider cluster C rooted at task v and a task u that provides

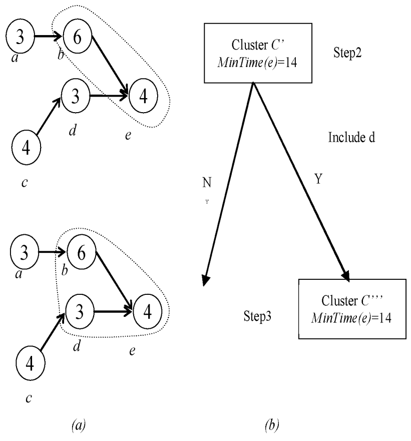

Fig. 12. Example of ready time based pruning.

input data for some task in cluster C. Let us denote by t_u the value

$$t_u = \text{MinTime}(u) + \text{Comm}(u \rightarrow C).$$

Essentially, t_u represents the time at which the output of u is available for cluster C. Now, consider cluster $\{C, u\}$, and compute the time at which the output of u is available to the rest of the tasks in the cluster $\{C, u\}$. Denote this value by t'_u

$$t'_u = \underset{\forall i \in \text{inputs}(u)}{\text{Max}} (\text{MinTime}(i) + \text{Comm}(i \rightarrow u)) + \text{ExecTime}(u).$$

If $t'_u \geq t_u$, then we need not compute the optimal schedule for cluster $\{C, u\}$. This is because including task u in cluster C does not improve the minimum starting time of any node in C and hence $\{C, u\}$ cannot give a better solution than C (lines 16–17 in Fig. 10). However, we still need to grow further along the inputs of u. For the example in Fig. 12, if we consider the branch which adds node d to cluster C', we get a new cluster C''' as shown in Fig. 12. In this cluster, t'_d is given by 8 which is the same as the value of t_d in cluster C'. We note that in cluster C''', the minimum ready time of any task will not decrease and the value of MinTime(e) does not decrease. Thus, we can avoid determining the optimal schedule for cluster C'''.

2) Structure Based: The structure-based pruning technique is based on the fact that we shall consider only convex clusters. As explained in the previous section, we use a decision tree to enumerate convex clusters rooted at a task v. Suppose we are at node t in the decision tree, C is the cluster built when we reach node t and u is the next task to be considered. If adding task u to C makes the resulting cluster $\{C, u\}$ nonconvex, we can prune the whole branch at node t that adds task u to C (line 15 in Fig. 10). Recall that the decision tree is constructed in reversed topological order. We can guarantee that there does not exist a set of tasks S in the subtree of t such that $\{C, u, S\}$ is a convex cluster.

For example, in Fig. 13, suppose we reach the node in the decision tree where we have cluster $C = \{e\}$, task b is not added and task a is being considered for inclusion in C. This makes the cluster $\{C, a\}$ nonconvex as shown in Fig. 13, and hence we need not proceed further along the branch, i.e., we need not consider tasks c and d along this branch.

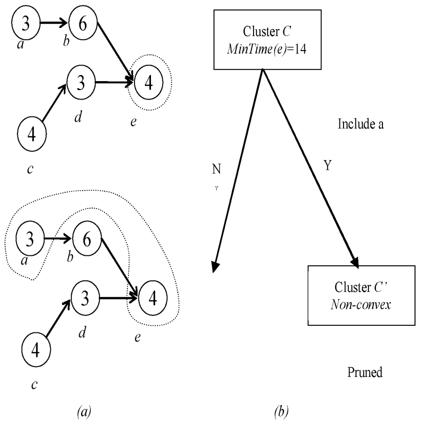

Fig. 13. Structure-based pruning.

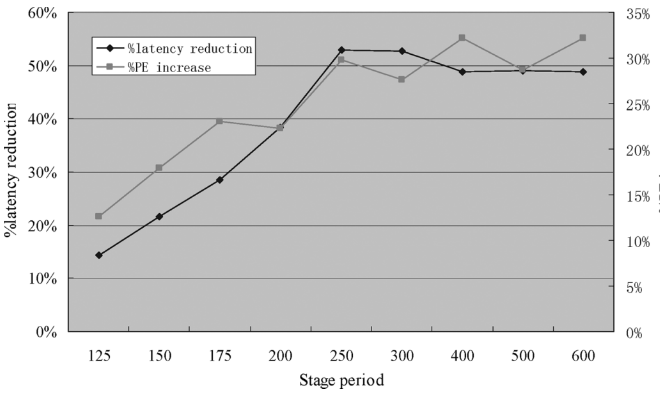

Fig. 14. Latency reduction under various stage period constraints.

VI. Experimental Results

We implemented our algorithms in a C++/Unix environment. First, a set of task graphs generated by TGFF [3] tool are used to evaluate the run time of the algorithm. The TGFF tool is first used to generate 20 acyclic task graphs with 25 to 100 tasks for each graph. We set the tasks' average execution time to be 100 and the variation to be 50. Fig. 14 shows the comparison of the sequential execution and parallel execution models in terms of application latency. The curve shows the percentage of latency reduction due to parallel execution under various stage period constraints. When the stage period is tight, there is no substantial latency improvement. Increasing the stage period from 120 to 250, we can see that the latency reduction improves significantly thanks to the task-level parallelism. On average, the highest average reduction is about 52%. When we further increase the stage period, latency improvement drops slowly because sequential model can grow a large cluster as well under loose period constraints. The associated cost with parallel execution is more processor usage. On average about 30% more processors will used to achieve 50% latency reduction.

We have also measured the effectiveness of our pruning techniques. Thirty tasks graphs were generated by TGFF with various number of task nodes. We set the stage period to be infinite so that the performance of the labeling algorithm can be accurately measured. As shown in Fig. 15, if the number of nodes in the graphs is less than 20, the speedup is negligible.

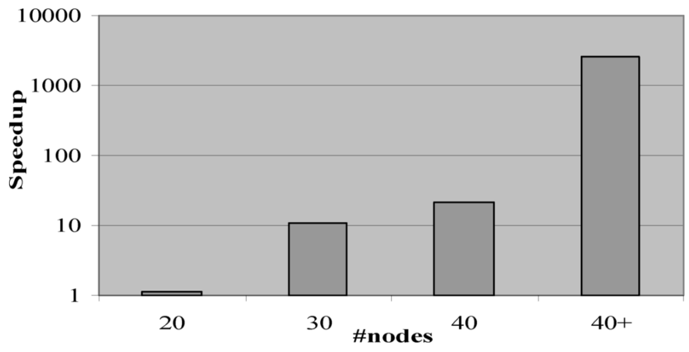

Fig. 15. Synthesis speedup with pruning techniques.

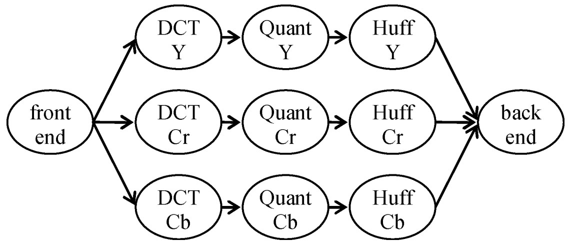

Fig. 16. Coarse grain MJPEG task graph.

When the graph size increases, the average speedup goes up to 11 X and 20 X on average for graphs with less than 30 and 40 nodes, respectively. When graph sizes increase to more than 40 nodes, pruning techniques bring significant performance improvement over the brute-force search. On average, the speedup is about 2500 X. For some cases, the run time is tremendously long (more than 20 hours) with the brute-force search while the search with our proposed pruning needs less than five seconds. Therefore, our proposed pruning techniques can effectively reduce the computation time while proving optimal solutions.

In addition, a real life benchmark, Motion JPEG (MJPEG) encoder, which was provided to us by the UC Berkeley Metropolis Group, is also used as an example to evaluate our system. MJPEG is a video codec where each video frame is separately compressed into a JPEG image. The resulting quality of intraframe video compression is independent from the motion in the image which differs from MPEG video. MJPEG is best suited for broadcast resolution interlaced video or IP-based video cameras. It consists of data preprocessing, discrete cosine transform, quantization, and Huffman encoding. To optimize the throughput, we process in parallel the three color components Y, Cb and Cr shown in Fig. 16 by taking advantage of the data parallelism. To evaluate the synthesis result, a Xilinx XUP Virtex II Pro development system [28] is used. The Microblaze [28], which is a single-issue, in-order RISC soft core provided by Xilinx, is used as our processor. The dedicated point-to-point inter-core communication links are implemented with the Xilinx's Fast Simplex Link. The Xilinx's EDK 8.1 and ISE 8.1 are used to construct and synthesize the processor network on the FPGAs. Our tool can automatically generate the hardware and software configuration for EDK, which significantly shortens the development time.

We have applied our tool to explore the different design tradeoffs for the MJPEG example. The raw images have a dimension

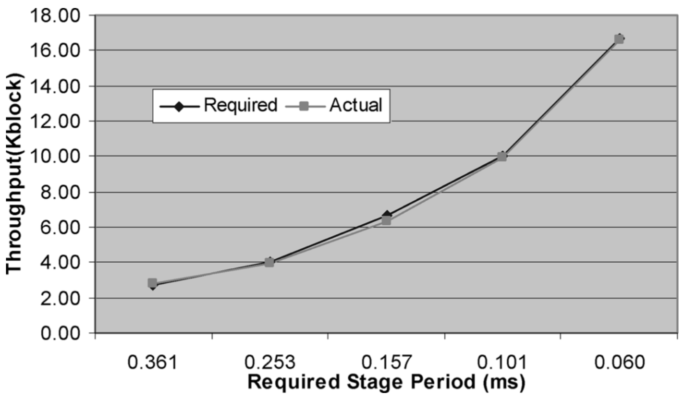

Fig. 17. Estimated throughput versus actual throughput.

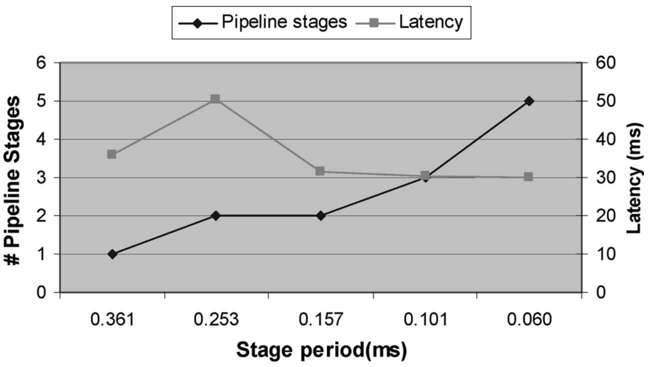

Fig. 18. Latency of various implementations.

of 96×72. Since the tasks operate on 8×8 blocks, we measured the throughput of the system with the number of processed blocks per second. We use our tool to search all the interesting design points. If all the tasks are implemented on a processor, we have the upper bound for the stage period. Since the execution time of DCT module is the largest among all the tasks, we can use its execution time as the lower bound for the feasible stage period. Then the tool tries all the stage periods in this range. For those generated configurations which use the same number of processors, we only keep the result with the smallest stage period. Finally, the tool finds five different configurations. The higher the throughput required, the more resources used to process tasks simultaneously. In the five configurations, the slowest processing occurs when using a single processor to achieve a throughput of 2.76 Kblocks/s. When using seven processors, we could achieve 16.6 Kblocks/s, which is more than 6 X improvement. Fig. 17 shows the required and actual throughput in the synthesis results. The actual throughput is at most 4.7% and on average 1.9% away from the required throughput. So we can see that our cost model and synthesis algorithm is accurate enough to capture the system behavior. As a result, the estimated throughput is very close to the actual throughput measured on the FPGA board.

The number of pipeline stages and latency of these configurations are shown in Fig. 18. When a tighter stage period is required, we usually need to increase the pipeline stages. As shown in Fig. 18, the single processor implementation has only one pipeline stage, while other implementations need 2, 3, and 5 pipeline stages, respectively, to process the data. However,

TABLE I
RESOURCE USAGE FOR VARIOUS IMPLEMENTATIONS

					Resources				
Throughput(kblocks/s)	Stage period(ms)	Frequency(MHz)	Pipeline stages	Latency(0.1ms)	#MB	#Mul	#RAM16	#Slice	FIFO (bytes)
2.77	0.361	100.00	1	3.6111	1	3	32	1121	0
3.96	0.253	100.00	2	5.0554	2	6	32	3332	960
6.35	0.157	100.00	2	3.1482	3	9	48	4087	1280
9.91	0.101	100.00	3	3.0279	5	15	80	8421	1920
16.61	0.060	100.00	4	2.4076	7	21	112	9494	3520

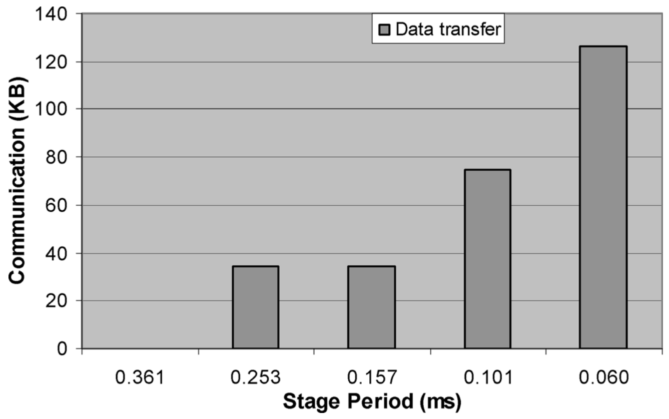

Fig. 19. Communication costs of various implementations.

the actual latency, measured by the product of the number of pipeline stages and the stage period as shown in Table I, may not increase since the stage period is shortened. The seven-processor implementation achieves the best throughput and latency. We also compare the communication costs among all the implementations. Fig. 19 shows the data transfer of various implementations. The data communication increases significantly when we map the tasks onto more processors. When all of the tasks are scheduled on a processor, there is no data transfer. The seven processor implementation needs to transfer 126.6 kB of data for each frame.

Apart from the experiments presented above, we also compare our results with the ILP solutions. In our ILP formulation, we try to minimize the resources under a throughput constraint. The ILP constraints are similar to [10]. We use the LPsolve [27] as the ILP solver and set the timeout to be four hours. Our experiment shows that ILP solutions take an unacceptable amount of time for some configurations (more than four hours), while our tool can finish all the computation in less than one second. For the five different throughput constraints, the results generated by our tool use the same number of processors as the ILP results. Moreover, most of the ILP solutions need more pipeline stages to process data as the ILP formulation does not consider latency minimization. In the worst case, the number of pipeline stages increases by 50%.

VII. CONCLUSION

In this paper, we present the theory and a framework for synthesizing ASPN systems for stream-oriented embedded applications. A set of efficient algorithms have been proposed to optimize latency and resources under the throughput constraint. This framework can help designers quickly explore the design space and make preferred tradeoffs. Extensive experiments show the efficiency of our approach. The application of our approach to the MJPEG encoding example shows interesting results by trading off costs for performance. In our future work, we are going to investigate synthesis techniques and extend our current work for application-specific heterogeneous multiprocessor system synthesis. The systems contain various heterogeneous processing elements such as programmable cores, DSP cores, and coprocessors, which would be more efficient for specific applications in terms of performance, cost, and power.

ACKNOWLEDGMENT

The authors would like to thank Dr. I. Bolsen from Xilinx, Inc., for his stimulating discussion on synthesis of networks of soft-core processors. They would also like to thank the UC Berkeley Metropolis Group, including Prof. A. Sangiovanni-Vincentelli, D. Densmore, and A. Davare for providing them the Motion JPEG example.

REFERENCES

[1] J. Cong, H. Li, and C. Wu, "Simultaneous circuit partitioning/clustering with retiming for performance optimization," in *Proc. ACM Design Automation Conf.*, 1999, pp. 460–465.
[2] J. Cong, G. Han, and W. Jiang, "Synthesis of an application-specific soft multiprocessor system," in *Proc. 15th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Monterey, CA, Feb. 2007, pp. 99–107.
[3] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graph for free," in *Proc. 6th Int. Workshop Hardware/Software Codesign*, Mar. 1998, pp. 97–101.
[4] R. P. Dick and N. K. Jha, "MOGAC: A multiobjective genetic algorithm for hardware-software cosynthesis of distributed embedded systems," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 17, no. 10, pp. 920–935, Oct. 1998.
[5] S. Dutta, R. Jensen, and A. Rieckmann, "Viper: A multiprocessor SOC for advanced set-top box and digital TV systems," *IEEE Design Test*, vol. 18, no. 5, pp. 21–31, Sep. 2001.
[6] H. El-Rewini, T. Lewis, and H. Ali, *Task Scheduling in Parallel and Distributed Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1994.
[7] M. Grajcar, "Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system," in *Proc. 36th ACM/IEEE Conf. Design Autom.*, New Orleans, LA, 1999, pp. 280–285.
[8] P. D. Hoang and J. M. Rabaey, "Scheduling of DSP programs onto multiprocessors for maximum throughput," *IEEE Trans. Signal Process.*, vol. 41, no. 6, pp. 2225–2235, Jun. 1993.
[9] A. Jerraya and W. Wolf, *Multiprocessor Systems-on-Chip*. New York: Elsevier, 2005.
[10] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer, "An automated exploration framework for FPGA-based soft multiprocessor systems," in *Proc. Int. Conf. Hardware/Software Codesign Syst. Synth.*, Sep. 2005, pp. 273–278.
[11] I. Karkowski and H. Corporaal, "Design of heterogenous multi-processor embedded systems: Applying functional pipelining," in *Proc. Conf. Parallel Architectures Compilation Tech. (PACT '97)*, San Francisco, CA, 1997, pp. 156–165.
[12] E. L. Lawler, K. N. Levitt, and J. Turner, "Module clustering to minimize delay in digital networks," *IEEE Trans. Comput.*, vol. C-18, no. 1, pp. 47–57, Jan. 1966.

[13] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[14] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proc. IEEE*, vol. 83, no. 5, pp. 773–799, May 1995.

[15] J. K. Lenstra, A. H. G. R. Kan, and P. Brucker, "Complexity of machine scheduling problems," *Ann. Discrete Math.*, vol. 1, pp. 343–362, 1977.

[16] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The design and implementation of a first-generation CELL processor," in *ISSCC Dig. Tech. Papers*, Feb. 2005, pp. 184–185.

[17] S. Prakash and A. C. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *J. Parallel Distrib. Comput.*, vol. 16, pp. 338–351, 1992.

[18] R. Rajaraman and D. F. Wong, "Optimal clustering for delay minimization," in *Proc. ACM Design Autom. Conf.*, 1993, pp. 309–314.

[19] K. Ravindran, N. Satish, Y. Jin, and K. Keutzer, "An FPGA-based soft multiprocessor system for IPv4 packet forwarding," in *Proc. 15th Int. Conf. Field Programmable Logic Applicat.*, Aug. 2005, pp. 487–492.

[20] V. Sarkar and J. Hennessy, "Compile-time partitioning and scheduling of parallel programs," in *Proc. SIGPLAN'86 Symp. Compiler Construction*, 1986, pp. 17–26.

[21] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. New York: Marcel Dekker, 2000.

[22] J. Subhlok and G. Vondran, "Optimal use of mixed task and data parallelism for pipelined computations," *J. Parallel Distrib. Comput.*, vol. 60, no. 3, pp. 297–319, 1997.

[23] J. D. Ullman, "NP-complete scheduling problem," *J. Comput. Syst. Sci.*, vol. 10, pp. 384–393, 1975.

[24] M. Wolf, "The definition of dependence distance," *ACM Trans. Programming Lang. Syst.*, vol. 16, no. 4, pp. 1114–1116, 1994.

[25] W. Wolf, "An architectural co-synthesis algorithm for distributed, embedded computing systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 5, no. 2, pp. 218–229, Jun. 1997.

[26] Altera Corp. [Online]. Available: http://www.altera.com.

[27] LPsolve. [Online]. Available: http://www.cs.sunysb.edu/~algorith/implement/lpsolve/implement.shtml.

[28] Xilinx, Inc. [Online]. Available: http://www.xilinx.com.

[29] Intel Corp. [Online]. Available: http://www.intel.com.

[30] Advanced Micro Devices, Inc. [Online]. Available: http://www.amd.com.

Jason Cong (S'88–M'90–SM'96–F'00) received the B.S. degree from Peking University, Beijing, China, in 1985, and the M.S. and Ph.D. degrees, both from University of Illinois at Urbana-Champaign, Urbana, in 1987 and 1990, respectively, all in computer science.

Currently, he is a Professor and the Chairman of the Computer Science Department of University of California, Los Angeles (UCLA). He is also a Codirector of the VLSI CAD Laboratory. He served on the ACM SIGDA Advisory Board, the Board of Governors of the IEEE Circuits and Systems Society, and the Technical Advisory Board of a number of electronic design automation (EDA) and silicon IP companies, including Atrenta, eASIC, Get2Chip, Magma Design Automation, and Ultima Interconnect Technologies. He was the Founder and President of Aplus Design Technologies, Inc., until it was acquired by Magma Design Automation in 2003. Currently, he serves as the Chief Technologist Advisor at Magma. Additionally, he has been a Guest Professor at Peking University since 2000. His research interests include CAD of VLSI circuits and systems, design and synthesis of system-on-a-chip, programmable systems, novel computer architectures, nanosystems, and highly scalable algorithms. He has published over 230 research papers and led over 30 research projects supported by Defense Advanced Research Projects Agency, NSF, SRC, and a number of industrial sponsors in these areas.

Dr. Cong received the Best Graduate Award from Peking University in 1985, and the Ross J. Martin Award for Excellence in Research from the University of Illinois at Urbana-Champaign in 1989. He received the NSF Young Investigator Award in 1993, the Northrop Outstanding Junior Faculty Research Award from UCLA in 1993, and the ACM/SIGDA Meritorious Service Award in 1998. He has received three best paper awards including the 1995 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN Best Paper Award, the 2005 International Symposium on Physical Design Best Paper Award, and the 2005 ACM Transaction on Design Automation of Electronic Systems Best Paper Award. He also received SRC Inventor Recognition Awards in 2000 and 2006, and the SRC Technical Excellence Award in 2000. He served on the technical program committees and executive committees of many conferences, such as Asia and South Pacific Design Automation Conference, Design Automation Conference (DAC), Field-Programmable Gate Arrays, International Conference on Computer Aided Design, International Symposium on Circuits and Systems, International Symposium on Physical Design (ISPD), and International Symposium on Low Power Electronics and Design (ISLPED), and several editorial boards, including the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS and the ACM Transactions on Design Automation of Electronic Systems.

Karthik Gururaj received the B.S. degree from IIT Madras, Chennai, India. He is currently pursuing the Ph.D. degree at the University of California, Los Angeles.

His research interests include processor-based synthesis for low power and reliability.

Guoling Han received the B.S. and M.S. degrees in computer science from Peking University, Beijing, China, in 1999 and 2002, respectively, and the Ph.D. degree from University of California, Los Angeles, in 2007.

His current research interests include platform-based system-level synthesis, and compilation techniques for reconfigurable systems and high-level synthesis.

Wei Jiang received the B.S. and M.S. degrees in computer science from Peking University, Beijing, China, in 1999 and 2002, respectively. He is currently pursuing the Ph.D. degree at the University of California, Los Angeles.

His current research interests include program analysis and transformation techniques for behavioral and system-level synthesis.