

Synthesis of an Application-Specific Soft Multiprocessor System

Jason Cong, Guoling Han and Wei Jiang

Computer Science Department, University of California, Los Angeles
Los Angeles, CA 90095, USA

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, that need to be explored and optimized. This paper addresses the problem of synthesizing the application-specific multiprocessor system to minimize latency and resource usage 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, clustering and packing, for efficient generation of the multiprocessor architecture with application-specific optimized latency and resources. 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.

Categories and Subject Descriptors

B.8.2 [Performance Analysis and Design Aids] Design Space Exploration - *throughput, latency, pipelining*

General Terms

Algorithms, Performance, Design

Keywords

Design Space, Pipeline, Clustering, Labeling, Multiprocessor

1. INTRODUCTION

The costs and technical challenges of designing application-specific integrated circuits (ASIC) have increased substantially as

we move into nanometer technologies. This trend makes it more attractive to consider using field-programmable gate array (FPGA) platforms, which may provide comparable raw performance and more flexibility at a lower cost for many applications. However, the shrinking feature size predicted in Moore's law has resulted in an explosive growth in functionality and the amount of computing power available on a single FPGA platform. For example, the Xilinx Virtex-4 [22] FPGA platform can hold up to 200K logic cells, 10 Mbits on-chip Block RAMs, 10 Gbps high-speed IOs, 512 highly optimized DSP blocks, embedded micro-processors, various kinds of communication channels, etc. Today it is perfectly feasible to design a multiprocessor System-on-a-Chip on FPGA platforms. In particular, soft processors on FPGAs (e.g., MicroBlaze from Xilinx [22] and NIOS /NIOS II from Altera [20]) offer a higher level of abstraction and present new options to the embedded designers. Using soft processors as building blocks, designers can easily implement their applications on a set of soft processors, and smoothly migrate from the original software specification to the final implementation. 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. Moreover, modern field programmable devices can hold tens of soft processors in a single FPGA, which allows the designers to build a multiprocessor system by exploiting the parallelism in applications. The computation-intensive tasks could be partitioned into subtasks and be executed on multiple processors simultaneously. Partitioning tasks on multiple soft processors not only provides a performance benefit, but also makes the design more flexible for adaptation to last minute changes. As a result, developers can change the performance characteristics of their embedded systems right up to the time the product goes to the final test. The benefits of using soft multiprocessor systems have been demonstrated in several projects (e.g., [15]).

In general, a *field programmable multiprocessor system* (FP-MPS) consists of soft processor cores, logic blocks, memories, communication channels and peripherals. Unlike the traditional parallel computation system, our FP-MPS is application-specific in that it should provide just enough computing power to satisfy the performance and power requirements of the given application and thus saves the energy and implementation cost as much as possible. Therefore, designers need to carefully partition tasks, allocate resources, map tasks to the processing elements, and build

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FPGA'07, February 18-20, 2007, Monterey, California, USA.
Copyright 2007 ACM 978-1-59593-600-4/07/0002...\$5.00.

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 FP-MPS 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. Since most of these applications have requirements on stream throughput (e.g., 30 frames/sec for MPEG decoding), the multiprocessor systems should provide just enough performance to satisfy the required throughput. Meanwhile, it is desirable to minimize the application latency under the throughput constraint, i.e., the time elapsed from the input stream to the output stream. For instance, video conference applications always prefer a small latency to allow real-time conversation. Therefore, our FP-MPS synthesis system optimizes the application latency with a given throughput constraint. Given the application specified as a task graph, we construct a network of soft processors connected by point-to-point FIFOs. Our exploration tool automatically decides the number of soft processors used in the system, communication buffer sizes, and processor interconnections, as well as the mapping of tasks onto the synthesized system. Previous work optimizes either latency or throughput. To our knowledge, this work is the first to take both latency and throughput into consideration in FP-MPS synthesis.

The remainder of the paper is structured as follows. We first discuss related works on multiprocessor design space exploration and mapping in Section 2. Section 3 describes the multiprocessor architecture model used in our system and problem formulation. The synthesis flow and algorithm is presented in Section 4. We have applied our exploration tool to a set of task graphs generated by TGFF [2] and the motion JPEG application under various design constraints. The exploration results for the motion JPEG on Xilinx FPGA boards are presented in Section 5. We conclude the work with future directions in Section 6.

2. RELATED WORKS

Over the past two decades, there have been other efforts that involved solving the multiprocessor design space exploration problem. The previous work could be categorized into two groups based on their optimization objectives, i.e., optimizing latency or throughput. The work in [16] 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 [13] 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 [19] 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 [6] 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 [3] synthesizes real-time heterogeneous multiprocessor architectures using an adaptive multi-objective genetic algorithm. Similarly, Grajcar [5] uses a genetic algorithm based on list scheduling to minimize the makespan on a bus-based multiprocessor. The work in [9] adopts modulo scheduling to map programs onto a pipeline of multiprocessors. More recently, Jin [7] 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.

Our contributions in this work are twofold. First, it is the first work proposed that considers the throughput, latency and resources simultaneously with support of execution time variation. Second, we provide a complete synthesis flow for the design space exploration of a FP-MPS system. It includes efficient labeling, clustering and packing algorithms which take the communication cost and the variation of tasks' execution time into consideration. Actually our proposed algorithms can be applied to any homogeneous multiprocessor system. The proposed synthesis algorithms generate latency-optimal and area-minimized results for acyclic graphs subject to throughput constraint. They could 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.

3. PROBLEM STATEMENT

3.1 System Model

Dataflow process network model [12] is commonly used in designing and implementing signal processing applications. Especially the synchronous dataflow model [11] in which the rate of data produce and consumption are constants for every dataflow edge, is widely adopted because bounded memory requirement and deadlock-free can be determined statically in finite time. Since any synchronous dataflow graph can be transformed to a homogeneous task graph [17] 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. We assume that each task should be executed on one and only one processor. 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. For complicated embedded applications, the tasks' execution time may exhibit significant variation at run time. Since each task

could contain control structures (e.g., if-else), the variation can come from the execution of different paths depending on the received data. For the system with cache memory, the cache hit/miss will also lead to variation on the execution time. To take into consideration of the variation, we represent the execution time of task v as an independent discrete random variable X_v . The associated probability mass function $p_{X_v}(x)$ gives the probability that the execution time of task v equals x . For simplicity, we will use $p(x)$ to represent the mass function of the variable X_v if it is clear from the context.

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 $|stage(u) - stage(v)|$. 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:

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

Since inter-processor communication needs the processors to transmit and receive data, it also brings communication delay. The synthesis tool could use any communication model to calculate the communication costs. In this work, we assume that all the data communications are through point-to-point FIFOs, which are available on most of the platform FPGAs. Therefore, we use a linear model [4] 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 $Comm(e) = C_{init} + vl(e) \times C_{unit}$. 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.

3.2 FP-MPS 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.
- The **success probability** $Prob$ defines the minimal probability that the final design can meet the throughput goal.

In most of the stream applications, they can tolerate some percentage of throughput failure¹. Suppose that the allowed failure probability is x , the success probability is equal to $(1-x)$. Because of the variation on the tasks' execution time, using worst case execution times will overestimate the total execution time. The synthesis tool should take advantage of this tolerance to minimize the number of used processors and latency. An example will be given in the following to illustrate how the synthesis tool can leverage this information.

- 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. Figure 1 shows an example of a non-convex cluster. Node c receives data from the cluster while it also sends data back to the cluster.

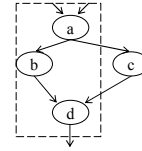


Figure 1. An example of non-convex cluster

We formulate our FP-MPS synthesis problem as follows.

Homogeneous FP-MPS Synthesis Problem: Given a task graph $G(V, E)$ with profiling information, user-specified throughput constraint and success probability $Prob$, construct a homogeneous FP-MPS system by (i) partitioning the tasks into convex clusters, and (ii) mapping the clusters onto the processors and inter-processor communication to FIFOs so that the application latency and number of used processors are minimized under the constraint that the required throughput can be satisfied with probability no less than $Prob$.

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 dependence if the cluster is not convex. For the example in Figure 1, 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 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.

¹ In general, designers can set the success probability to be 1 if the throughput must be satisfied in every case.

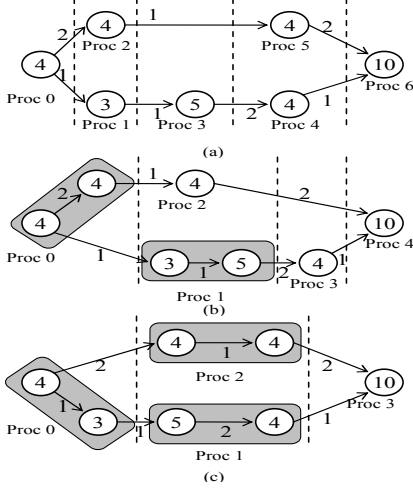


Figure 2. Pipelined execution example

Figure 2 shows an example with seven tasks. For this example, we assume that all the tasks have a fixed execution time. The nodes are labeled with the computation times (in us) and the edges are labeled with the communication times if inter-processor communication required. Executed on a single processor, the system cannot process new inputs until all the tasks have been executed once. Thus, the stage period is $34 us$. By exploiting the task-level parallelism and pipelining the tasks, we could improve the throughput. Figure 2 (a) shows a naive five-stage pipeline implementation with seven processors. On each processor, the total execution time, including the communication time to transmit data, is no more than $10 us$. Therefore, we could achieve a stage period of $10 us$ and a 3.4 X throughput improvement. Figure 2 (b) shows another pipeline implementation with five processors. The shaded cycles represent that the tasks covered by them are mapped onto a shared processor. It can also achieve the stage period of $10 us$. However, the latency has been reduced to four stages. Actually, the optimal solution, as shown in Figure 2 (c), only needs three pipeline stages and four processors. Although the three implementations achieve the same system throughput, their latencies and required processors could differ by a factor of 1.67 and 1.75 respectively. This example illustrates the possible huge exploration space in multiprocessor synthesis and the importance of optimizing the throughput, latency and resource usage at the same time.

As aforementioned, the synthesis tool should also use the variation on the tasks' execution time and failure tolerance to optimize the used processors. For the example in Figure 1, let us assume the possible execution time and probability for each task are as the following table.

Table 1. Execution time and possibility for each task

Task	(exec_time, probability)	(exec_time, probability)
a	(10, 20%)	(6, 80%)
b	(8, 50%)	(5, 50%)
c	(9, 50%)	(4, 50%)
d	(10, 20%)	(7, 80%)

Each task has two different execution times. For example, task A's execution time can be either 10 with probability 20% or 6

with probability 80%. For the simplicity, we assume that the communication cost is zero. Suppose the required stage period is no more than 15 and the success probability is 85%. If we only consider the worst case execution time, it needs four processors to satisfy the throughput. However, the task a and task b can be mapped to a single processor since the probability that their total execution time is no more than 15 is 90%. Similarly, $\{a, c\}$ and $\{b, d\}$ can also be mapped to a single processor. Therefore, we can obtain a solution with two processors. The first processor executes task a and c sequentially. Their data are sent to the second processor which process task b and d .

4. PROPOSED SYNTHESIS ALGORITHM

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 four 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. Finally, in the fourth step, we pack the task clusters to optimize the resource usage. With the synthesized result, we generate a hardware configuration for the underlying platform and corresponding software program. We will discuss the four steps in detail in the following sections. We will first present synthesis algorithms for task graphs with fixed execution times and the success probability equal to 1. The extension to variable execution times is discussed in the section 4.2.3.

4.1 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 could 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. 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 require that the execution time of a task should be no less than the communication time. This assumption is used throughout this paper.

4.2 Labeling and Clustering

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 [10][14][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 interconnect delay, is usually used. In multiprocessor synthesis, the inter-processor communication cost may vary significantly. Therefore, our labeling algorithm should take the non-uniform 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. More importantly, the traditional clustering algorithm depends on node duplication to guarantee the optimal latency. In a multiprocessor system, task duplication may lead to an incorrect execution result. For example, a file write operation may be executed twice if the associated task is duplicated. Interestingly, our proposed algorithm can generate the latency-optimal result without task duplication.

4.2.1 Labeling

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 could be obtained based on the following lemma.

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

Our labeling algorithm, which is similar to Lawler's algorithm [10], 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 non-decreasing 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. Figure 3 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 10 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 $\{a, b, c, d, e\}$ and communication cost 3, we cannot accommodate all the nodes in a single cluster. Therefore, the label of e is 1.

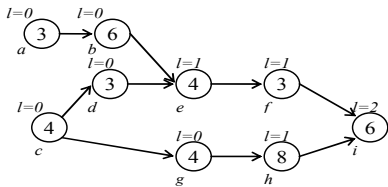


Figure 3. The labels of a task graph

4.2.2 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. Since a node may occur in multiple clusters,² a simple solution is to make replicas of the node and put them in every cluster. However, as explained above, task replication is not allowed in the multiprocessor synthesis problem. Therefore, a legalization step is needed to decide which cluster a shared node belongs. Intuitively, if a task node is moved out from one cluster, the minimal label for that cluster may increase. Interestingly, we find that if we move the nodes intelligently, the clustering solution after moving is still depth-optimal.

The legalization step processes nodes in their topological order. We use a set, denoted as S , to record the shared nodes among multiple clusters. If a node v is shared by multiple clusters, we put v into S and compute the internal communication cost from the nodes in S to other nodes for each cluster. We will put the nodes of S into the cluster with the largest internal communication cost if S is the only nodes that this cluster shares with others. The resulting cluster is called the *resident cluster* of v . Then S is cleared for the next node. Otherwise, we continue to look at the next node. After we decide the resident cluster for each shared node, v is removed from other non-resident clusters and direct inter-cluster communication links are added to transfer the result from the resident cluster. For the example in Figure 3, we may generate clusters $A=\{c, g\}$ and $B=\{c, d\}$ which share a common node c . The internal communication costs for clusters A and B are two and one respectively. Figure 4 shows the clustering result for the task graph in Figure 3. Notice that the node c belongs to two clusters in the original result. After legalization, c will reside in cluster A and an inter-cluster link, shown as the dotted arrow, is added between cluster $\{c, g\}$ and $\{d\}$.

Under our assumption that a task's computation time is no less than the communication time, we could draw the following conclusion:

LEMMA 3: The legalization process does not increase the total execution time for any cluster in the original solution.

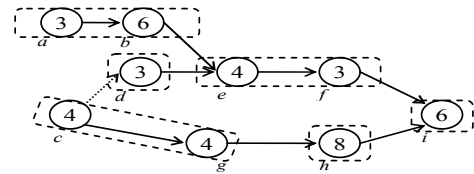


Figure 4. Clustering result

Given lemmas 1-3, we can prove the following theorem.

THEOREM 1: The labeling and clustering algorithms generate latency-optimal pipeline solutions for directed acyclic task graphs.

4.2.3 Extension to Tasks with Variable Execution Times

In the stage period checking step, if a task's execution time is a random variable, the value of the distribution function $F(T) =$

² Actually, all of the clusters have the same label value.

$\sum_{y \leq T} p(y)$ should be no less than *Prob*, where $p(y)$ is the probability mass function of the random variable. Otherwise, the task needs to be decomposed to sub-tasks until the criteria can be satisfied for each task. Similarly when we perform task decomposition, we require that the minimum execution time of a task should be no less than the communication time.

In the labeling step, it is easy to see that both Lemma 1 and Lemma 2 still hold. Therefore, our proposed algorithm can work on task graphs with variable execution times. When labeling node v , one key question is to check whether the node v can have the label L_v , which is the maximal label of v 's predecessors. Recall M_v denotes the set of predecessors with the maximal label. In another word, we need to calculate the probability that the total execution time of these tasks $S_v = M_v \cup \{v\}$ is no more than T . If X and Y are two independent random variables with probability distribution f and g , respectively, the probability distribution of the sum $X+Y$ is given by the convolution ($f * g$). Specifically, $(f * g)(T) = \sum f(t) * g(T-t)$ as t runs through all the possible values of X . This can be extended for multiple independent variables. Therefore, we obtain the probability distribution of S_v by calculating a multi-variable convolution function H_{S_v} , whose inputs are the corresponding random variables of the nodes in S_v . We can label the node v with the label L_v if and only if $H_{S_v}(T)$ is larger than *Prob*. Applying the modified labeling algorithm, we can see that the labeling and clustering algorithm are still latency-optimal for directed acyclic graphs.

4.2.4 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 [18]. The inter-iteration dependency will limit the minimum achievable stage period. Consider the example shown in Figure 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 [6].

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

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

This condition ensures that the inter-iteration data dependency is correctly maintained. For the example shown in Figure 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 Figure 5 (b), we can make a partition on the forward path as shown in Figure 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.

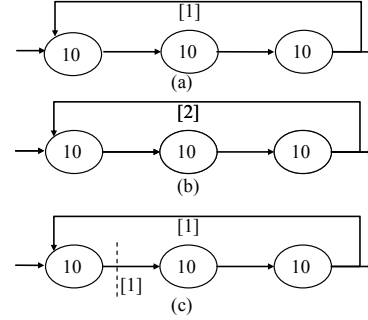


Figure 5. Cyclic task graph and its pipeline partition

4.3 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. The study in [8] shows that this strategy is at most 22% away from the optimal.

5. EXPERIMENTAL RESULTS

We implemented our algorithms in a C++/Unix environment. First, a set of benchmarks generated by TGFF [2] tool are used to evaluate the run time of the algorithm and illustrate the impacts when the variation of the execution time is considered. The TGFF tool is used to generate about 20 acyclic task graphs with 10 to 50 tasks for each graph. We set the tasks' execution time to be a binomial distribution. We have set different variation ranges in our experiment. The largest variations are set to be 1.5X, 1.75X, 2.0X, 2.25X and 2.5X of the average execution time respectively. For example, 2X means the longest execution time is about twice of the average execution time. The experiments show that our synthesis tool can finish all the computation within 1 second. The following figure shows the average normalized number of used processor with different success probability over all the benchmarks. The number of processors with 100% success probability is used as the baseline resource usage. The numbers of used processors are normalized to the baseline processor usage for all the cases. From this figure, we can see that the number of used processors decreases with the decreasing success probability. For example, the 2X curves shows that when the system only requires the target throughput to be satisfied with probability 85%, we can use only 72% of the baseline processor resources if the variation and probabilities are considered. In another word, the synthesis using static worst case time will waste about unnecessary 28% of

the resources. Even with a small failure tolerance (e.g. 5%), we can still see considerable processor reduction. In addition, the experiments show that when the execution times have a larger variation, we can achieve more resource savings with the same success probability. For example, with the success probability 90%, the 2.5X variation can achieve about 19% more resource reduction compared to the 1.5X variation. Therefore, the synthesis tool considering variation can see more benefits for those programs with huge variations.

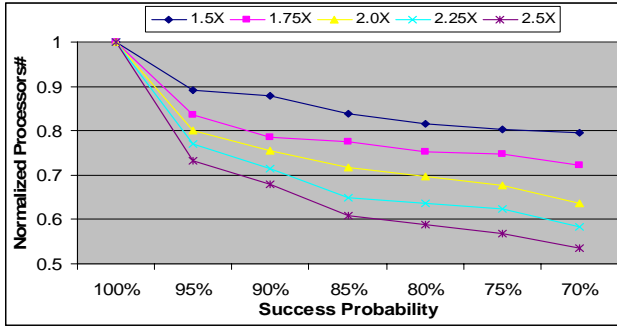


Figure 6. Avg. number of processors with different success probabilities

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 Figure 7 by taking advantage of the data parallelism. To evaluate the synthesis result, a Xilinx XUP Virtex II Pro development system [22] is used. The Microblaze [22], 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, that significantly shortens the development time.

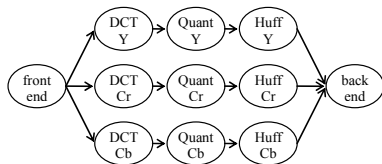


Figure 7. Coarse grain MJPEG task graph

We have applied our tool to explore the different design tradeoffs for the MJPEG example. The raw images have a dimension of 96x72. Since the tasks operate on 8x8 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 period 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 6X improvement. Figure 8 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.

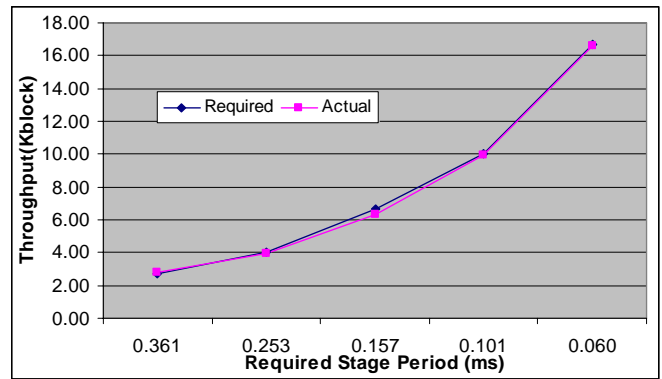


Figure 8. Estimated throughput vs. actual throughput

The number of pipeline stages and latency of these configurations are shown in Figure 9. When a tighter stage period is required, we usually need to increase the pipeline stages. As shown in Figure 9, 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, the actual latency, measured by the product of the number of pipeline stages and the stage period as shown in Table 2, 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. Figure 10 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. Finally, we show the detailed implementation cost for each configuration in Table 2. It shows that the packing step can effectively reduce the number of processors used. For the second configuration, the packing step reduces the required processors from four to two. On average, it saves resources by 27.9%.

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 relations are similar to [7]. We use the LPSolve [21] 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%.

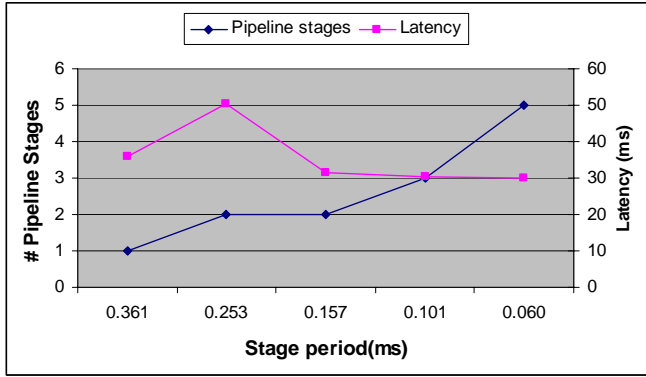


Figure 9. Latency of various implementations

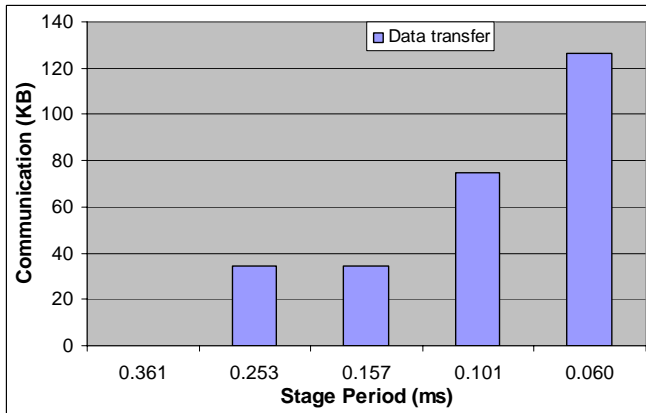


Figure 10. Communication costs of various implementations

6. CONCLUSIONS

In this paper we present the theory and a framework for synthesizing application-specific FP-MPS systems. A set of efficient algorithms, including labelling, clustering and packing, 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 co-processors, which would be more efficient for specific applications in terms of performance, cost and power.

ACKNOWLEDGMENT

This research is partially funded by MARCO/DARPA Gigascale Silicon Research Center (GSRC), National Science Foundation under grants CCF-0530261 and CNS 0647442, and a grant from Xilinx, Inc. under the California MICRO program.

The authors would like to thank Dr. Ivo Bolsen from Xilinx, Inc. for his stimulating discussion on synthesis of networks of soft-core processors. We also thank the UC Berkeley Metropolis group, including Prof. Alberto Sangiovanni-Vincentelli, Douglas Densmore and Abhijit Davare for providing us 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.*, pp. 460-465, 1999.
- [2] R.P. Dick, D.L. Rhodes, W. Wolf. TGFF: Task Graph For Free. In *Proc. of the 6th international workshop on Hardware/software codesign*, pp.97-101, 1998.
- [3] R.P. Dick and N.K. Jha. MOGAC: a Multiobjective Genetic Algorithm for Hardware-Software Cosynthesis of Distributed Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 920-935, October 1998.
- [4] H. El-Rewini, T. Lewis, and H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall, 1994.
- [5] M. Grajcar. Genetic List Scheduling Algorithm for Scheduling and Allocation on a Loosely Coupled Heterogeneous Multiprocessor System. In *Proc. of the 36th ACM/IEEE Conference on Design Automation*, pp. 280-285, New Orleans, Louisiana, USA, 1999.
- [6] P.D. Hoang and J.M. Rabaey. Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput. *IEEE Transactions on Signal Processing*, vol. 41, no. 6, June 1993.
- [7] Y. Jin, N. Satish, K. Ravindran, and K. Keutzer. An Automated Exploration Framework for FPGA-Based Soft Multiprocessor Systems. In *Proc. 2005 International Conference on Hardware/Software and System Synthesis (CODES-05)*, pp. 273-278, September, 2005.
- [8] D. Johnson. Approximation Algorithms for Combinatorial Problems. In *Journal of Computer and System Science* 9(3), pp. 256-278, 1974.
- [9] I. Karkowski and H. Corporaal. Design of Heterogenous Multi-processor Embedded Systems: Applying Functional Pipelining. In *Proc. of the 1997 Conference on Parallel Architectures and Compilation Techniques (PACT '97)*, pp. 156-165, San Francisco, CA, USA.
- [10] E. L. Lawler, K. N. Levitt, and J. Turner. Module Clustering to Minimize Delay in Digital Networks. *IEEE Transactions on Computers*, vol. C-18, no. 1, pp. 47-57, January 1966.
- [11] E. A. Lee and D. G. Messerschmitt, Synchronous Dataflow, In *Proc. of the IEEE*, vol. 75, no. 9, pp. 1235-1245, September 1987.
- [12] E. A. Lee and T. M. Parks, Dataflow Process Networks, In *Proc. of the IEEE*, pp. 773-799, May 1995.

- [13] S. Prakash and AC Parker. SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems. *Journal of Parallel and Distrib. Comp.*, vol. 16, pp. 338-351, 1992.
- [14] R. Rajaraman and D. F. Wong. Optimal Clustering for Delay Minimization. In *Proc. ACM Design Automation Conf.*, pp. 309-314, 1993.
- [15] K. Ravindran, N. Satish, Y. Jin, and K. Keutzer. An FPGA-Based Soft Multiprocessor System for IPv4 Packet Forwarding. In *Proc. 15th International Conference on Field Programmable Logic and Applications (FPL-05)*, pp. 487-492, August, 2005.
- [16] V. Sarkar and J. Hennessy. Compile-time Partitioning and Scheduling of Parallel Programs. In *Proc. the SIGPLAN 86 Symposium on Compiler Construction*, pp. 17-26, 1986.
- [17] S.Sriram and S.S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Dekker, Inc, 2000.
- [18] M. Wolf. The Definition of Dependence Distance. In *ACM Transactions on Programming Languages and Systems*, vol.16, issue 4, pp. 1114-1116, 1994.
- [19] W. Wolf, an Architectural Co-synthesis Algorithm for Distributed, Embedded Computing Systems, *IEEE Transactions on Very Large Scale Integration Systems*, vol. 5, issue 2, pp. 218-229, 1997.
- [20] Altera Corp., <http://www.altera.com>
- [21] LPsolve, <http://www.cs.sunysb.edu/~algorithm/implementation/lpsolve/implementation.shtml>
- [22] Xilinx Inc., <http://www.xilinx.com>.

Table 2 Resource usage for various implementations

Throughput(kblocks/s)	Stage period(ms)	Frequency(MHz)	Pipeline stages	Latency(0.1ms)	Resources				
					#MB wo packing	#MB w packing	#Mul	#RAM16	#Slice
2.77	0.361	100.00	1	3.6111	1	1	3	32	1121
3.96	0.253	100.00	2	5.0554	4	2	6	32	3332
6.35	0.157	100.00	2	3.1482	4	3	9	48	4087
9.91	0.101	100.00	3	3.0279	7	5	15	80	8421
16.61	0.060	100.00	4	2.4076	11	7	21	112	9494