Synthesis of Application Specific Heterogeneous Pipelined Multiprocessor Systems

Haris Javaid Sri Parameswaran

University of New South Wales, Australia
{harisj,sridevan}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-0911
May 2009

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

This paper describes a rapid design methodology to create a pipeline of processers to execute streaming applications. The methodology has two separate phases. The first phase uses a heuristic to rapidly search through a large number of processor configurations (configurations differ by the base processor, the additional instructions and cache sizes) to find the near Pareto front. The second phase utilizes either the above heuristic or an ILP (Integer Linear Programming) formulation to search a smaller design space to find an appropriate final implementation. By the utilization of the fast heuristic with differing runtime constraints in the first phase, we rapidly find the near Pareto front. The second phase provides either an optimal or a near optimal solution. Both the ILP formulation and the heuristic find a system with the smallest area, within a designer specified runtime constraint. The system has efficiently explored design spaces with over 10^{12} design points.

We integrated this design methodology into a commercial design flow and evaluated our approach with different benchmarks (JPEG Encoder, JPEG Decoder and MP3 Encoder). For each benchmark, the near Pareto front was found in a few hours using the heuristic (took several days for the ILP). The results show that the average area error of the heuristic is within 2.5% of the optimal design points (obtained using ILP) for all benchmarks.

1 INTRODUCTION

The miniaturization of transistors, coupled with the demand for functionality, performance, and low power has resulted in the emergence of Multi Processor System on Chip (MPSoC) based embedded devices in the market. MPSoCs can be categorized either as homogeneous or heterogeneous. Homogeneous MPSoCs contain identical processing entities, whereas Heterogeneous MPSoCs use different processing elements. These processing elements can be coprocessors, general purpose processors, or application specific architectures such as ASICs or DSPs. Heterogeneous MPSoCs usually have a smaller footprint than homogenous MPSoCs, and consume less power by mapping an application's tasks onto the most suitable processing elements. The recent emergence of Application Specific Instruction Set Processors (ASIPs) [1, 2, 3] has seen the use of a homogeneous platform to produce heterogeneous processors for seamless use in embedded applications. Both coarse- and fine-grained parallelism in an application can be exploited using ASIPs in an MPSoC. An ASIP's architecture and instruction set can be tuned based upon the needs of a specific task, improving performance while minimizing area. Thus, an ASIP exhibits the flexibility of a processor, yet has the customizability of an ASIC.

Heterogeneous pipelined multiprocessor architectures are where processing entities are connected in a pipelined fashion via queues. The incoming data stream is processed by each pipeline stage which may contain one or more processing elements. The data stream goes through each stage until the output is finally written by the last pipeline stage. Each of the stages can be customized to suit a particular part of the application which is executed by that pipeline stage processor(s). For example, a five stage pipelined implementation of JPEG Encoder can be Color Space Conversion, DCT, Quantization, Huffman Encoding and writing to file. The first stage reads the raw image macro block and performs RGB to YCbCr conversion and level shifting. The second stage performs DCT on the macro block input by the first stage. In such an implementation, while one stage is busy processing one macro block other stages will be busy processing other macro blocks in a typical pipelined fashion. Thus, pipelined heterogeneous MPSoCs provide a practical implementation platform for streaming applications with high performance gains and reduced area footprint [4].

In this paper, we present a design methodology for implementing streaming applications onto pipelined heterogeneous multiprocessor systems. Since streaming applications inherently benefit from pipelined implementations, the presented design flow will help designers implement efficient application specific multiprocessor systems. A partitioned application is taken and the standalone tasks of the application are assigned to processors (ASIPs) in the pipeline. An iterative design flow is used to obtain a design space (consisting of ASIP configurations) which satisfies a set of criteria. Once an approximate design space is selected, either ILP or the heuristic (with design space pruning) can be used to obtain the final design point (set of ASIP configurations) with minimum area under runtime constraint provided by the designer.

The rest of the paper is organized as follows: Section 2 provides an overview of the work done in the MPSoC domain and Section 3 provides the application model and the pipelined architecture. The problem addressed in this work is formalized in Section 4. Section 5 explains the proposed design methodology with Section 5.1, Section 5.2 and Section 5.3 detailing the pruning algorithm,

ILP formulation and the heuristic used in our design flow respectively. Section 6 provides the experimental setup with the results presented in Section 7. Finally, Section 8 summarizes the paper.

2 RELATED WORK

Numerous multiprocessor architectures have been used to implement multimedia applications. For example, a real time video and graphics management system was described in [5] and an HDTV system in [6]. Researchers have also explored different techniques such as loop pipelining and pipelined scheduling of tasks to speed up applications using multiprocessor architectures [7, 8, 9]. In contrast to these works, we focused on mapping streaming applications on ASIP-based pipelined systems to achieve high performance with reduced area footprint owing to heterogeneity of pipelined MPSoCs.

Integer Linear Programming (ILP) [10] is a widely used technique in design space exploration and optimization of multiprocessor architectures. Batista et al. [11] and Kuang et al. [12] used Mixed Integer Linear Programming (MILP) and ILP respectively to schedule pipelined execution of an application task graph on heterogeneous multiprocessor architectures. The authors in [13] used MILP for optimization of a shared bus multiprocessor architecture, but the approach becomes impractical as the available pool of processing elements increase. None of the above works specifically targeted ASIP-based pipelined systems which is a viable implementation platform for streaming applications [4].

Since ILP based approaches can be slow for designing complex systems, researchers have proposed heuristics to efficiently explore the design space of multiprocessor systems. Although heuristics do not guarantee an optimal solution, they provide remarkable improvements in the time for design space exploration. Sun et al. [14] examined multi-ASIP systems, simultaneously mapping and scheduling tasks, in addition to custom instruction selection for ASIPs. In [14], the runtime of the system is minimized given an area budget for the custom instructions. The authors in [15] represented applications as cyclic directed graphs and explored mapping and partitioning of the application onto pipelined multiprocessor architecture. The work in [15] finds a multiprocessor system with minimum latency and number of processors under throughput constraints, but targets only homogeneous pipelined systems.

Implementation of streaming applications onto heterogeneous pipelined multiprocessor systems using ASIPs was explored in [16] and [17]. A heuristic is proposed in [16] to rapidly explore the design space consisting of available ASIP configurations. A multiprocessor design with maximum performance gain per unit area increase with respect to a single processor system is considered as optimal and chosen by the heuristic. In contrast, this paper focuses on the problem of finding a design with minimum area while a given runtime constraint is satisfied. This is because streaming applications exhibit soft real time constraints and there is no need to add on extra area once the runtime constraint is satisfied. Furthermore, we used a more accurate runtime calculation equation (Section 3.2) by taking into account cold cache start. Comparing with [16], this work uses design space pruning with either ILP or a novel heuristic to obtain an optimal or near optimal design.

In contrast to [16], Javaid et al. [17] used ILP to find an optimal design for an

ASIP-based pipelined multiprocessor system. The presented ILP formulation considered single pipeline systems (i.e., only one processor per stage) and only a case study was performed on JPEG. Thus, the work in [17] lacks a generic design methodology in contrast to our work, which also targets generic pipelined multiprocessor systems (generic means each stage of the pipeline can contain any number of processors (ASIPs) in parallel, resulting in what is referred to as multiple pipeline multiprocessor systems). Furthermore, our ILP formulation is different and results in a reduced number of variables, in turn reducing the complexity of the ILP. We have also generalized the pruning technique presented in [17] for multiple pipeline systems. Thus, this work proposes a complete framework (ILP and a novel heuristic) for design of pipelined multiprocessor systems.

To summarize, our work differs from all of the above in the following ways:

1. A complete ILP formulation (with design space pruning algorithm) for selection of ASIP configurations in a generic pipelined multiprocessor system is presented.
2. A novel heuristic for design space exploration of an ASIP-based pipelined multiprocessor system is proposed and compared with the ILP.
3. A designer-driven framework is presented by integrating the proposed ILP and the heuristic in a commercial design flow to obtain efficient multiprocessor implementations.

To the best of our knowledge, this is the first complete work using both ILP and a heuristic to address the implementation of application specific systems in the context of heterogeneous pipelined multiprocessor systems.

3 BACKGROUND

3.1 Application Model

Sequential applications with the following characteristics are targeted in this work: one, the application contains a kernel which is executed multiple times; and two, the operations in the kernel are independent of each other so that their execution can be overlapped. For example, a JPEG encoder application has a kernel which is executed multiple times (equal to the number of macro blocks in the input image). Note that we refer to the number of times an application kernel is executed as the number of iterations of that application. Secondly, the JPEG kernel consists of five major operations: reading and color space conversion; DCT; Quantization; Huffman Encoding; and, writing to file, all of which are independent of each other. Thus, a sequential implementation of the JPEG encoder can be partitioned into five standalone tasks. These tasks can be executed on different processors communicating with each other via queues. Since suitably pre-partitioned benchmark programs are not available, we partitioned the benchmark applications manually as described in Section 6. We created four partitioned applications: JPEG Encoder Single Pipeline (JESP); JPEG Encoder Multiple Pipeline (JEMP); JPEG Decoder (JD); and, MP3 Encoder (MP3E). Figure 3.1 shows the partitioned applications where tasks are connected through queues denoted by the arrows. Due to limited space, the names are not shown on the partitioned tasks. JPEG Encoder has been partitioned in two differing ways for the purpose of comparison (refer to Section 7).

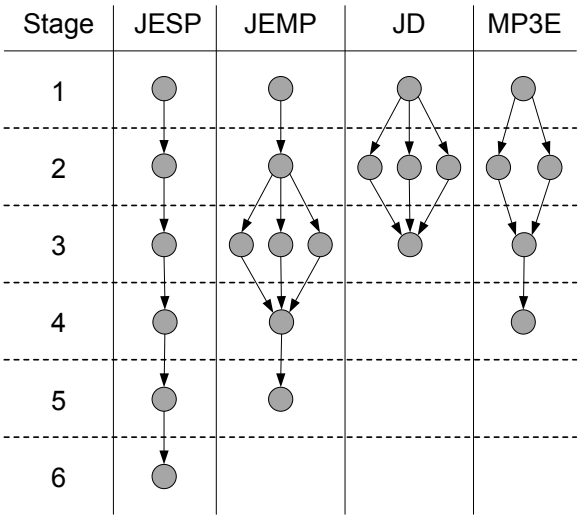

Figure 3.1: Benchmark Applications

3.2 Pipelined Multiprocessor Architecture

A pipelined system consists of processing entities connected in a pipeline using queues (FIFOs) which allow communication at a much higher bandwidth, devoid of the contention typically exhibited by a shared bus architecture. Each stage of the pipeline can contain one or more processors (ASIPs) to execute some part of the application. We assume that the depth of each pipeline stage is one, that is, there cannot be more than one ASIP in series in a pipeline stage. However, there can be more than one ASIP in parallel and we refer to that pipeline stage as a parallel pipeline stage. For example, stage 3 of JEMP in Figure 3.1 will be a parallel pipeline stage (when mapped to a pipelined system). Further to this assumption, the output of an ASIP in stage i can only be connected to other ASIPs in stage $i+1$. A pre-partitioned application is taken and the tasks of the application are mapped onto these ASIPs. Each ASIP in the pipelined system has a number of available configurations. ASIP configurations differ by the additional instructions they contain and by the sizes of their instruction and data caches. Additional instructions for an ASIP are generated according to the task mapped on that particular ASIP. A commercial ASIP design tool from Tensilica Inc. [3] is used to automatically generate ASIP configurations from a base processor. Base processors can be identical for each ASIP in the pipelined system or can be different. The processor configuration generation is controlled using a parameter which we refer to as the overhead granularity. The granularity parameter specifies the minimum amount of difference (in terms of gates) between two sets of additional instructions for the same base processor. Thus, lowering the granularity will generate more sets of different additional instructions. Permutation of the sets of additional instructions, and instruction and data cache sizes make up the tailored configurations for each of the ASIPs. Area of an ASIP configuration includes base processor, additional instructions (if any) and instruction and data cache sizes.

To accurately determine the execution time of an application (system run-

time) on an ASIP-based pipelined system, one has to exhaustively simulate every considered design point (possible combination of ASIP configurations). Since there are several million combinations which are possible, it is not feasible to simulate all of them to accurately determine the system runtime. For each combination, the simulation time is several minutes therefore, it may take years to simulate millions of combinations. To overcome this problem and to determine the runtime quickly, we use an estimation equation.

$$\mathbb{R} = \mathrm{R}^{init}(s_1) + \sum_{i=1}^{M} \mathrm{L}^1(s_i) + (I-1) \times \mathrm{L}(s_{critical}) + \mathrm{R}^{final}(s_M) \quad (3.1)$$

$$\begin{aligned} \textit{where} \qquad \mathrm{R}^{init}(s_1) &= \max_{1 \le j \le N_0} \{\mathrm{R}^{init}(p_{1,j})\} \\ \mathrm{R}^{final}(s_M) &= \max_{1 \le j \le N_M} \{\mathrm{R}^{final}(p_{M,j})\} \\ \mathrm{L}^1(s_i) &= \max_{1 \le j \le N_i} \{\mathrm{L}^1(p_{i,j})\} \\ \mathrm{L}(p_{i,j}) &= \textstyle\sum_{k=2}^{I} \mathrm{L}^k(p_{i,j})/(I-1) \\ \mathrm{L}(s_i) &= \max_{1 \le j \le N_i} \{\mathrm{L}(p_{i,j})\} \end{aligned}$$

Given a pipelined multiprocessor system, system runtime can be calculated using Equation 3.1. $R^{init}(p_{i,j})$, $R^{final}(p_{i,j})$, $L^1(p_{i,j})$ and $L(p_{i,j})$ refer to initialization time, finalization time, first latency (due to the cache misses on cold start) and averaged latency of processor j in pipeline stage i respectively, while s_i stands for pipeline stage i. I, N_x and M refer to the number of iterations of the application (being run on the pipelined system), number of processors in pipeline stage x and total number of pipeline stages respectively. First latency of a processor is the first iteration's execution time while averaged latency is the average execution time for rest of the $I-1$ iterations. Initialization and finalization time is the time to execute the non-kernel operations which are executed only once at the start and end of the application. The equation calculates the system runtime by summing up the initialization time of the first stage, time to fill up the empty pipeline, time spent by the critical stage once the pipeline is filled and finalization time of the last stage. The *max* functions in the equation make certain that for parallel pipeline stages, the processor with the worst latency is used as the stage latency because it will hide the latency of other processors in that stage. The latencies of each processor used in this equation include the computation and net communication time of each processor, which means communication stalls are excluded. This is because slower processors stall for the critical processor in the pipelined system and communication stalls do not contribute to the system runtime as they are hidden in the latency of the critical processor.

For an ASIP-based pipelined system where each ASIP can be configured in one of the available configurations, system runtime with a particular set of ASIP configurations can be calculated using Equation 3.1 given individual configuration latencies. Thus, all the ASIP configurations are simulated separately with their respective tasks mapped on them to record the timings and area information. These recorded values can then be used to calculate system runtime and system area for any combination of ASIP configurations. The area of the

pipelined system is the summation of the area cost of all the processors, where area of a processor is measured in terms of gates. Since pre-partitioned applications are used, and the designer knows how much data will be transferred between two processers and how fast data will be generated in the streaming applications, determining the size of FIFOs should be straightforward. Such calculations are beyond the scope of this paper. However, the size of FIFOs will be constant, thus FIFO areas are not included in the design space exploration. In this work, we also do not consider the area of the processor memories (except for caches).

4 PROBLEM DEFINITION

A partitioned application can be represented as a Directed Acyclic Graph (DAG) where vertices represent tasks and edges represent the data dependencies between the tasks. Thus, an application partitioned for a pipeline design can be represented as graph G^a

$$G^a = (V, E)$$

The set V contains the tasks of the partitioned application:

$$V = \{v_{i,j} : 1 \leq i \leq M, 1 \leq j \leq N_i\}$$

where M is the number of stages and N_i is the number of tasks in stage i. Each $v_{i,j}$ represents a standalone task that can be mapped to an ASIP in the pipelined system. Data dependencies between the stand alone tasks are given by

$$E = \{(v_{i,j}, v_{i+1,l}) : 1 \leq i \leq M-1, 1 \leq j \leq N_i, 1 \leq l \leq N_{i+1}\}$$

A pipelined multiprocessor system can also be represented as graph G^p

$$G^p = (P, F)$$

where P is the set of ASIPs in the pipelined system given as

$$P = \{p_{i,j} : 1 \leq i \leq M, 1 \leq j \leq N_i\}$$

while F represents the FIFO channels between the ASIPs for communication

$$F = \{(p_{i,j}, p_{i+1,l}) : 1 \leq i \leq M-1, 1 \leq j \leq N_i, 1 \leq l \leq N_{i+1}\}$$

The set of configurations for each ASIP $p_{i,j}$ is represented as

$$P_{i,j} = \{p_{i,j,k} : 1 \leq i \leq M, 1 \leq j \leq N_i, 1 \leq k \leq K_{i,j}\}$$

where $K_{i,j}$ is the total number of configurations for ASIP $p_{i,j}$. Each ASIP configuration $p_{i,j,k}$ is annotated with a 4-tuple number $(R^{init}|R^{final}|0, L^1, L, A)$ where A stands for the area of that particular configuration. The first element of the tuple is equal to R^{init} for ASIP configurations in the first stage, R^{final} for those in the last stage and 0 for the others. For example, $p_{2,2,1}$ is associated with $(0, L^1_{2,2,1}, L_{2,2,1}, A_{2,2,1})$ representing the first configuration of second ASIP

in stage 2. These tuples represent the individual processor latencies and area which can be used to calculate system runtime and system area.

Mapping of a partitioned application on a pipelined system can be defined as a mapping from G^a to G^p. Each task $v_{i,j}$ is mapped to corresponding ASIP $p_{i,j}$ depicting mapping of tasks onto ASIPs. Similarly, data dependencies from G^a are mapped to corresponding FIFOs in G^p. Once the tasks are mapped to ASIPs and data dependencies to FIFOs in the pipelined system, one configuration of each ASIP is selected to obtain a set of ASIP configurations as the final design. The cost function to be minimized is defined as

$$A = \sum_{i=1}^{M} \sum_{j=1}^{N_i} \mathbb{A}(p_{i,j})$$

where function $\mathbb{A}$ returns the area of ASIP $p_{i,j}$. The minimization of A is performed while ensuring the system runtime (calculated using Equation 3.1) remains within the runtime constraint R_c provided by the designer. Using the definitions above, the problem of selection of configurations (one for each ASIP) with respect to minimization of cost function A and runtime constraint R_c is solved. This selection problem is the design space exploration problem being solved in this paper, and a design flow (refer to Section 5) is proposed for rapid exploration of large design spaces.

5 DESIGN FLOW

The overall design methodology to address the selection problem (described in Section 4) is shown in Figure 5.1. The methodology consists of two phases: Design Space Generation; and, Design Space Exploration. The designer provides the partitioned application, pipeline architecture and runtime constraint as input. The partitioned application is provided as separate C code files for each of the ASIPs in the pipelined system.

As shown in Figure 5.1, the design space generation phase is determined by the base processors, the overhead granularity, and the instruction and data cache configurations. Note that in this design flow the base processors are specified by the designer. While it is possible to include the selection of the base processors in the design flow, experience suggests that the designer prefers to choose the base processors. ASIP configurations are generated using a commercial tool from Tensilica Inc. [3] as described in section 3.2. Using the input parameters, a designer can control the amount of design space to be generated for a particular application. Executables (for each task) are generated using the provided C file and the generated sets of additional instructions for that particular task. A simulation environment from Tensilica Inc. [3] is used to simulate all the ASIP configurations separately with the generated executables. Simulation results are used to record the timing and area values for all the ASIP configurations which will be used in the exploration phase.

The design space exploration phase shown in Figure 5.1 is based on a heuristic and/or a 0-1 ILP formulation. At the output of this phase, one configuration of each ASIP is selected so that the resulting pipelined system has minimum area while its runtime is less than runtime constraint R_c. The problem of selecting ASIP configurations can either be formulated as a 0-1 ILP problem (and

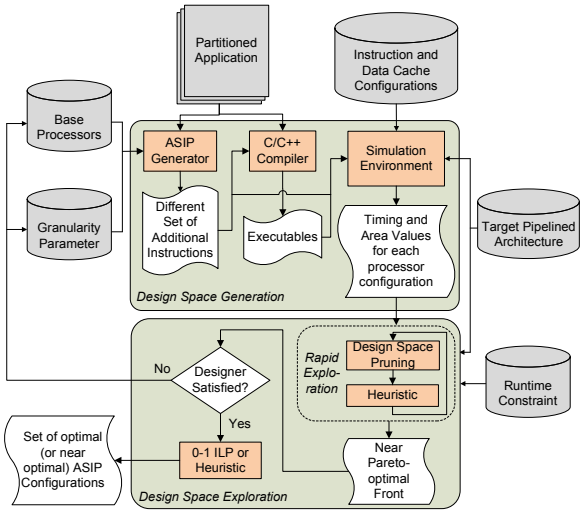

Figure 5.1: Proposed Design Flow

solved using an ILP solver) or solved using the heuristic. The first step is to perform a rapid design space exploration (shown in Figure 5.1) with the help of the heuristic and obtain the near Pareto front of the design space. The near Pareto front will reveal the area trend of the design space and will guide the designer to change the input parameters such as base processors, overhead granularity, etc. to obtain a new design space with reduced area footprint. The rapid exploration phase uses the design space pruning algorithm (briefly explained in section 5.1) and the heuristic (explained in section 5.3). The rapid exploration phase calls the pruning algorithm and the heuristic with all possible runtime constraints that can be supported by the generated design space in steps of R_s – starting from the minimum runtime of the design space until the maximum runtime of the design space. R_s is the step amount between the runtime constraints and is specified by the designer. This provides the designer with the near Pareto front of the design space in just a few hours for a typical multimedia application.

The Rapid Exploration in Figure 5.1 can be performed as many times as required until a design space is found which satisfies the designer. This approach provides an opportunity to tweak the design in several hours (excluding simulation time) as compared to ILP which may take days as shown in Section 7. After the approximate design space is finalized, 0-1 ILP (with design space pruning and provided runtime constraint) can be used (only at the end) to obtain the optimal design as shown in Figure 5.1. If the 0-1 ILP times out, the heuristic can be used instead to obtain the near optimal set of ASIP configurations. The designer can also use a different partitioning of the application and change the instruction and data cache configurations for generation of the new design space (which is not shown in Figure 5.1 due to space restrictions).

5.1 Design Space Pruning

A number of ASIP configurations can exceed the runtime constraint specified by the designer, thus those ASIP configurations can be removed from the design space. A generalized version of a pruning algorithm in [17] is used in this work. This generalization is applied to pipeline systems with parallel pipeline stages (stages having more than one ASIP in parallel). If the designer provided runtime constraint is greater than the maximum runtime supported by the design space, not even a single configuration is removed. Similarly, for a runtime constraint less than the minimum runtime of the design space, all the configurations of each ASIP are removed resulting in an empty design space.

Three different execution times are defined for each ASIP configuration: Latency, First Latency (FL) and Processing Time (PT) as follows:

$$\mathit{Latency}(p_{i,j,k}) = L_{i,j,k} \quad 1 \leq i \leq M$$

$$\mathit{FL}(p_{i,j,k}) = \begin{cases} R^{init}_{i,j,k} + L^{1}_{i,j,k} & i = 1 \\ R^{final}_{i,j,k} + L^{1}_{i,j,k} & i = M \\ L^{1}_{i,j,k}, & i \neq 1, i \neq M \end{cases}$$

$$\mathit{PT}(p_{i,j,k}) = \begin{cases} R^{init}_{i,j,k} + L^{1}_{i,j,k} + (I-1) \times L_{i,j,k} & i = 1 \\ R^{final}_{i,j,k} + L^{1}_{i,j,k} + (I-1) \times L_{i,j,k} & i = M \\ L^{1}_{i,j,k} + (I-1) \times L_{i,j,k} & i \neq 1, i \neq M \end{cases}$$

The runtime equation described in Section 3.2 (Equation 3.1) can also be interpreted as the summation of PT for critical pipeline stage and FL for the rest of the stages. The basic idea of the pruning algorithm (shown in Algorithm 1) is to analyze each ASIP configuration separately and calculate the minimum system runtime that can be supported by that ASIP configuration. For example, by using the first ASIP's first configuration in stage 1 ($p_{1,1,1}$) and the minimum PT for the critical stage (if stage 1 is not critical) and minimum FL for the noncritical stages, system runtime R is calculated using Equation 3.1. The calculated R will be the minimum runtime supported by ASIP configuration $p_{1,1,1}$ because minimum possible execution times for all the other stages have been used. Thus, if R violates the runtime constraint R_c, configuration $p_{1,1,1}$ cannot be present in the optimal solution (and thus is deleted) and the algorithm proceeds to the next ASIP configuration.

Algorithm 1 shows our approach in detail to prune the design space. First of all, stage configurations with respect to minimum Latency, FL and PT are obtained (lines 1-4). The function *MinStageFL* (line 2) selects one configuration for each ASIP in the given pipeline stage which has the minimum FL. For example, one configuration for each ASIP in stage 3 of JEMP (Figure 3.1) having minimum FL will be selected. Then, from amongst the three selected ASIP configurations, the one having the worst FL is selected as stage configuration. FL of the selected stage configuration is the minimum FL of that stage and is stored in *stage_minFL* array. Similarly, other stage configurations with minimum PT and minimum Latency are returned by *MinStagePT* and *MinStageL* functions respectively and stored in the corresponding arrays.

Algorithm 1: Design Space Pruning Algorithm

Input: $p_{0,0}, p_{0,1}, ...p_{M,N_M}$ where $p_{i,j}$ array contains tuples associated with each configuration of ASIP $p_{i,j}$, and runtime constraint R_c

```
1  for i=1 to M do
2      stage_minFL[i] = MinStageFL(S_i);
3      stage_minPT[i] = MinStagePT(S_i);
4      stage_minL[i] = MinStageL(S_i);
5  crit = Max(stage_minL) where crit refers to stage number
6  L_c = Latency(stage_minPT[crit]);
7  for i=1 to M do
8      for j=1 to N_i do
9          Initialize R_i = 0;
10         for k=1 to M do
11             if k != i and k != crit then
12                 R_i += FL(stage_minCL[k]);
13         for k=1 to K_{i,j} do
14             Initialize R = R_i;
15             if i == crit then
16                 if L_{i,j,k} > Latency(stage_minL[i]) then
17                     R += PT(p_{i,j,k});
18                 else
19                     R += PT(stage_minPT[i]);
20             else
21                 if L_{i,j,k} > L_c then
22                     R += PT(p_{i,j,k}) + FL(stage_minCL[crit]);
23                 else
24                     R += PT(stage_minPT[crit]) + Max(FL(p_{i,j,k}),
                           FL(stage_minFL[i]));
25             if R > R_c then
26                 Delete configuration p_{i,j,k}
```

Once the configurations of each stage with respect to minimum FL, PT and Latency are available, the algorithm selects the initial critical stage having the worst minimum latency (line 5) which is then stored as L_c (line 6). The algorithm goes through all configurations of each ASIP at the stage level, such that all the ASIPs in stage 1 are analyzed first (lines 7-26). At this instant, the sum of the minimum FL of all the stages except the initial critical stage and the current stage is calculated (lines 10-12). This step ensures that minimum possible execution time of each noncritical stage is used for runtime calculation. Then, if the current stage is the critical stage, PT of the current stage is used to calculate the runtime (lines 15-19). However, if this is not the case, the algorithm needs to decide the critical stage. This is because if the current configuration's L is greater than L_c (initial critical stage's Latency) then current stage is the critical stage and its PT is used to calculate runtime with minimum FL of initial critical stage (lines 21-22). Otherwise minimum PT of initial critical stage and FL of current stage is used for runtime calculation (lines 23-24). The final runtime R calculated in this way will be the minimum runtime supported by

this particular ASIP configuration because minimum possible execution times for all the other pipeline stages have been used. The calculated R is used to either retain or delete the current ASIP configuration (lines 25-26).

The algorithm goes through each ASIP's configurations only once. Thus, its complexity is $O(\sum_{i=1}^{M} N_i \times K_{i,j,max})$ where $K_{i,j,max}$ is the number of configurations for the ASIP with the maximum number of configurations from amongst all the ASIPs in the pipelined system. The percentage reduction of the design space depends on the runtime constraint R_c provided. Therefore, the lower the value of R_c the larger the design space that will be pruned. Note that the optimality of the 0-1 ILP will not be affected due to the pruning, since only configurations which can never be part of the optimal solution are deleted. However, it will reduce the design space which in turn reduces the complexity of 0-1 ILP, and enables the use of ILP to obtain optimal designs from large design spaces (upto 10^{16} design points).

5.2 ILP Formulation

The ASIP configuration selection problem can be stated as follows: *Given a pipelined architecture with ASIP configurations and the system runtime constraint R_c, one configuration is selected for each ASIP such that system area is minimized while system runtime satisfies the provided runtime constraint R_c.* The selection problem is directly mapped to a 0-1 ILP problem and is formulated as follows.

Variables & Objective Function

Binary variables are used to select ASIP configurations.

1. $x_{i,j,k}$ *equals 1* if configuration k of ASIP $p_{i,j}$ is selected.
2. $s_{m,n,o}$ variables are used for parallel pipeline stages. For parallel stages (for example, stage 3 of JEMP in Figure 3.1), one configuration is selected for each ASIP using $x_{i,j,k}$ variables. However, from amongst the selected ASIP configurations the one having the worst latencies is used for runtime calculation, and is referred to as the stage configuration. $s_{m,n,o}$ *equals 1* if configuration o of ASIP $p_{m,n}$ is selected as stage configuration for parallel pipeline stage m (and is used for runtime calculation).

Objective function is to minimize the area of the system which is the summation of the area of all the ASIPs in the pipelined system.

$$\text{Minimize} \sum_{i=1}^{M} \sum_{j=1}^{N_i} \sum_{k=1}^{K_{i,j}} A_{i,j,k} x_{i,j,k}$$

Constraints

The different constraints applicable to the configuration selection problem are listed below.

1. Only one configuration can be selected for an ASIP.

$$\sum_{k=1}^{K_{i,j}} x_{i,j,k} \quad = \quad 1 \qquad \forall\ i,j$$

2. For a parallel pipeline stage, only one ASIP configuration can be selected as the stage configuration.

$$\sum_{j=1}^{N_i} \sum_{k=1}^{K_{i,j}} s_{i,j,k} \quad = \quad 1 \qquad \forall\ i \textit{ where } N_i > 1$$

3. From amongst the ASIP configurations selected using $x_{i,j,k}$ in a parallel pipeline stage, one configuration is selected as the stage configuration.

$$s_{i,j,k} - x_{i,j,k} \leq 0 \qquad \forall\ i,j,k \textit{ where } N_i > 1$$

4. An ASIP configuration is chosen as the stage configuration in a parallel pipeline stage (for example, stage 3 of JEMP in Figure 3.1) only if its L^1 is maximum amongst the L^1s of the other selected ASIP configurations in that stage.

$$\max_{1 \leq k \leq K_{i,j}} (L^1_{i,j,k}) \times [1 - s_{i,n}] + \sum_{k=1}^{K_{i,n}} L^1_{i,n,k} s_{i,n,k}$$
$$\geq \sum_{k=1}^{K_{i,j}} L^1_{i,j,k} x_{i,j,k} \qquad \forall\ j,n \textit{ where } 1 \leq j,n \leq N_i,$$
$$j \neq n \textit{ and } N_i > 1$$

where $s_{i,n}$ is the summation of all $s_{i,n,k}$ for ASIP $p_{i,n}$. There will be $N_i - 1$ such constraints for each ASIP in parallel pipeline stage i as it has to be compared against every other ASIP in that stage. Thus, in total we have $N_i(N_i - 1)$ such constraints for stage i which ensures that the worst ASIP configuration with respect to L^1 is selected as the stage configuration.

5. System runtime must be less than or equal to designer's runtime constraint. This constraint is written for each processor considering that processor as the critical processor of the pipelined system.

$$\sum_{n=1}^{N_0} \sum_{k=1}^{K_{0,n}} R^{init}_{0,n,k} s_{0,n,k} + \sum_{m=1}^{M} \sum_{n=1}^{N_m} \sum_{k=1}^{K_{m,n}} L^1_{m,n,k} s_{m,n,k}$$
$$+ (I - 1) \times \sum_{k=1}^{K_{i,j}} L_{i,j,k} x_{i,j,k}$$
$$+ \sum_{n=1}^{N_M} \sum_{k=1}^{K_{M,n}} R^{final}_{M,n,k} s_{M,n,k} \leq R_c \qquad \forall\ i,j$$

where R_c refers to runtime constraint.

ILP formulation presented here differs from the one presented in [17] as it takes into account parallel pipeline stages. Furthermore, the critical processor selection is embedded into the equations in constraint 5 whereas the authors in [17] used more variables, which makes the ILP formulation in [17] more complex and time consuming.

Algorithm 2: Heuristic

Input: $p_{0,0}, p_{0,1}, ...p_{M,N_M}$ where $p_{i,j}$ array contains tuples associated with each configuration of ASIP $p_{i,j}$, and runtime constraint R_c

```
1 Calculate L_upper
2 for i=0 to M do
3     for j=0 to N_i do
4         for k=0 to K_{i,j} do
5             if L_{i,j,k} > L_upper then
6                 Delete configuration p_{i,j,k}
7 for i=0 to M do
8     for j=0 to N_i do
9         Configuration with minimum area is selected for ASIP p_{i,j} from
          amongst the remaining configurations
```

5.3 Heuristic

As ILP is exhaustive in the worst case and the focus of this work is to target large design spaces, we propose a novel heuristic which can be used for rapid design space exploration. The heuristic is shown in Algorithm 2. First of all, an upper bound of the latency of the pipelined system is calculated using the provided runtime constraint R_c as shown in Equation 5.1. The equation subtracts the maximum possible L^1s of all the stages from R_c and calculates an upper limit of the latency which is applied over all the pipeline stages. Thus, all the ASIP configurations having L greater than L_{upper} are removed. From the remaining configurations, one configuration for each ASIP is selected which has the minimum area. This ensures that the system runtime (calculated using selected configurations) will always be less than the runtime constraint, since the sum of maximum L^1s for all the stages is subtracted before calculating L_{upper}. It should be noted that for some ASIPs all the configurations could have been removed, and thus the heuristic will not be able to find a solution. An error signal is generated where a solution could not be found. This is most likely to happen for runtime constraints that are very close to minimum runtime of the design space.

$$L_{upper} = \frac{R_c - \sum_{i=1}^{M} \max_{1 \leq j \leq N_i} \{\max_{1 \leq k \leq K_{i,j}} (L^1_{i,j,k})\}}{(I-1)} \tag{5.1}$$

Like the pruning algorithm, the heuristic analyzes each configuration of all the ASIPs only once. Hence, its complexity is $O(\sum_{i=1}^{M} N_i \times K_{i,j,max})$.

6 EXPERIMENTAL SETUP

We integrated the proposed methodology into a commercially available design environment from Tensilica Inc. [3]. The Xtensa LX family of processors with RB-2007.1 toolset is used for our experiments. This toolset includes a C/C++ compiler, Instruction Set Simulator (ISS) and the Xtensa Modeling Protocol (XTMP – used to simulate multiprocessor platforms). ISS is used to simulate

the LX processors, while multiprocessor systems are described and simulated in the XTMP to verify functionality. FIFOs are used to communicate between processors. FIFO interface includes push and pop functions used by the connected processors to write to and read from a FIFO respectively. A push to a full FIFO or a pop from an empty FIFO stalls the processor. XTMP uses ISS to simulate the individual processors of a multiprocessor system and thus, can generate profiling information such as clock cycle count for an application, etc.

The RB-2007.1 toolset includes the Xtensa PRocessor Extension Synthesis (XPRES), which generates tailored processor directly from the C/C++ code, given a base processor. XPRES analyzes the C code and automatically generates new instructions and custom register files which are described in Tensilica Instruction Extension (TIE) language. The generated additional instructions may consist of a combination of fused operations, vector operations, FLIX instructions [18] and specialized operations [19]. Using the overhead granularity parameter, XPRES automatically generates multiple TIE files reflecting different sets of additional instructions. For example, if *granularity = 5000 gates*, then size of ASIP configuration $p_{i,j,k} >= p_{i,j,k-1}$ *+ 5000* excluding the size of caches. TIE files are compiled through TIE compiler and seamlessly combined with the base processor to generate tailored processors.

The partitioned benchmarks shown in Figure 3.1 were created for experimental purposes and validation of our approach. All the applications are partitioned on logical boundaries. For example, five stages for JEMP in Figure 3.1 are: Color Space Conversion; Level Shifting; DCT and Quantization; Huffman Encoding; and, writing to file. The three processors in stage 3 process Y, Cb and Cr components of a macro block in parallel. Other applications are also partitioned in a similar way.

We used *lp_solve* [20], a free application, to solve the formulated *0-1 ILP* problem. *lp_solve* reads an input file in LP format and outputs a text file specifying the values of the decision variables. The design space pruning algorithm and heuristic are implemented in Perl. The whole process of design space generation and exploration as shown in Figure 5.1 is automated (in Perl) and integrated into the Tensilica's design environment [3].

7 RESULTS & ANALYSIS

The results are presented in two parts: first, we compare the effectiveness of the heuristic with the 0-1 ILP; and second, we show how our methodology can be used to tweak designs in several hours using JPEG decoder as an example. We developed 4 benchmarks to test our methodology as shown in Figure 3.1. Table 7.1 shows the seven different base processors (columns 2-8) that were used to generate ASIP configurations. PIF stands for processor interface to instruction and data cache, while MAC16 and MUL16 refer to a MAC and a MUL unit respectively. Using these base processors, we generated design spaces for the benchmarks as shown in Table 7.2. Columns 2-7 show the pipeline stages and the number of configurations generated for an ASIP in a particular pipeline stage. For example, 288 configurations for all the three ASIPs in stage 2 of JD (Figure 3.1) were generated. Since JD had only a 3 stage pipeline design, columns $S_4 - S_6$ contain no data. Column 8 shows the total design space for a particular benchmark which is the permutation of all ASIP configurations

Features	BP1	BP2	BP3	BP4	BP5	BP6	BP7
Speed (MHz)	568	568	568	563	563	563	533
PIF	64	64	64	128	128	128	128
MAC16	√			√	√	√	
MUL16					√		
MUL32	√			√		√	√
Area (Gates)	88K	70K	68K	110K	88K	115K	87K

Table 7.1: Base Processors used in our experiments

generated for that benchmark, while column 9 shows the total time to generate the design space, simulate ASIP configurations, and record the timing and area information. As can be seen, simulating all ASIP configurations only once (instead of simulating all the possible ASIP configuration combinations in a pipelined system) helps the designer target large design spaces (in the order of 10^{12} design points). We used a granularity of 5000 gates and base processor BP4 for JESP, 4000 and BP5 for JEMP, 2000 and BP6 for MP3E, and 3000 and BP1, BP2 and BP3 for stage 1, stage 2 and stage 3 of JD respectively. Instruction and data cache sizes are changed from 1KB to 32KB. These values of overhead granularity parameter, base processors and instruction and data cache sizes were chosen to generate sufficiently large number of configurations to make up a large design space for each benchmark.

Bench.	S_1	S_2	S_3	S_4	S_5	S_6	Design Space	Time
JESP	144	144	396	144	252	144	4.2×10^{13}	19 hrs
JEMP	180	180	252, 252, 252	252	144	-	2.35×10^{16}	15 hrs
JD	288	288, 288, 288	252	-	-	-	1.73×10^{12}	13 hrs
MP3E	288	252, 252	324	324	-	-	1.92×10^{12}	24 hrs

Table 7.2: Design Space Generation Phase Results

Table 7.3 compares the effectiveness of the proposed heuristic to the ILP. Both the heuristic and ILP (with design space pruning) are executed with runtime constraints spanning the whole design space. Exploring the whole design space provides the Pareto front of the design space that can guide the designer, so that late changes can be made to the design. In Table 7.3, Column 3 shows the timing statistics of ILP and the heuristic, while columns 4-5 show the average area error (Av. AE) and maximum area error (Max. AE) respectively for the design points obtained via the heuristic (from the best obtained by the ILP). A timeout of 24 hours is used for the ILP in all the benchmarks. Average Area Error (column 4) is calculated on the basis of number of runtime constraints provided referred to as points in column 2. For example, 1261 different runtime constraints were used for ILP while exploring JESP design space. As explained in Section 5.3, there can be some runtime constraints for which the heuristic is unable to find a solution. This leads to different number of runtime constraints for ILP and the heuristic. The average area error for all the benchmarks is less than 2.5%, and the worst area error in all the benchmarks is 10.46%. Most importantly, the total time to explore the design space (column 3) differs by several magnitudes for ILP and the heuristic. For JEMP, 42 days were spent

Benchmark	Technique (points)	Total Time	Av. AE	Max. AE
JESP	ILP (1261)	4 hrs	-	-
	Heuristic (971)	17 mins	0.99%	2.72%
JEMP	ILP (1190)	42 days	-	-
	Heuristic (949)	16 mins	0.48%	2.22%
JD	ILP (7000)	28 days	-	-
	Heuristic (7000)	2 hrs	2.22%	10.46%
MP3E	ILP (10700)	18 days	-	-
	Heuristic (10100)	3 hrs	0.25%	5.03%

Table 7.3: Comparison of Heuristic with ILP

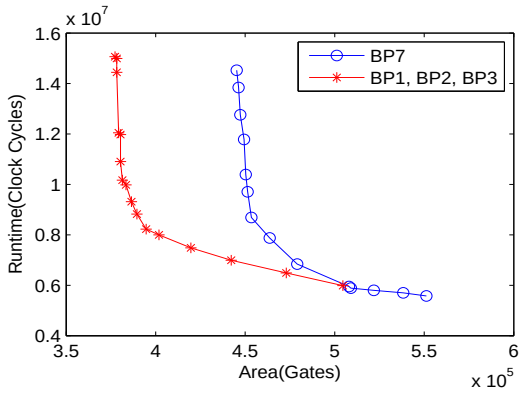

Figure 7.1: Near Pareto front of JPEG Decoder with same base processors and different base processors

in obtaining the Pareto front, while the heuristic revealed the near Pareto front in only 16 mins (with 2.22% maximum area error). Thus, the heuristic can be used for rapid design exploration as shown in Figure 5.1. We also explored design space using heuristic (without pruning) and the results show an increase in area error. This is because the maximum L^1s can be greater in the full design space when compared to the pruned space, reducing area error for heuristic with pruning. The number of points (runtime constraints) to be explored depends on the value of runtime step R_s which is specified by the designer. For our experiments, R_s = *1,000* is used for JESP, JEMP, and JD and R_s = *10,000* is used for MP3E so that there are enough points for average calculation, and thus a reasonable comparison can be done.

To show the effectiveness of our design methodology and how rapid exploration can be used to optimize the area of a pipelined system, we performed a case study on JD. The initial JD implementation was the same as shown in Figure 3.1 except that we used base processor BP7 with granularity of 3000 gates to generate and simulate a design space of 5.2×10^{12} design points in 20 hours. The near Pareto front of the design space is shown in Figure 7.1 marked as BP7 which was obtained in 2 hours using the rapid exploration phase. For sake of simplicity, not all the design points are shown in this graph. We observed that the change in area for runtime range of 1.4×10^7 to 0.9×10^7 clock cycles is miniscule. This means that the processor configurations in the final design are

being underutilized for runtimes greater than 0.9×10^7 clock cycles. Thus, we decided to use different base processors: BP1 for stage 1, BP2 for stage 2 and BP3 for stage 3 (first stage is reading and entropy decoding, second stage is dequantization and DCT, and third stage is color space conversion and writing back to file). A more complex processor is used for stage 1, but a simple processor is used for stage 3 which is not computationally intensive. The second graph annotated as 'BP1, BP2 and BP3' shows the near Pareto front of the design space with different base processors. As can been seen, there is a significant area reduction for most of the runtimes. This strategy can be repeated again and again until the designer is satisfied. It should be noted that it took only one day (including the simulation time) to generate and explore the new design space. The designer could well choose to only use the ILP solution with the provided runtime constraint, and if the optimal design point obtained is not within the area budget, a new design space can be generated. However, the solution time of the ILP cannot be guaranteed. Thus, for large design spaces the heuristic is more likely to reveal the near Pareto front quickly. Once the designer is satisfied with the design space (after Rapid Exploration in Figure 5.1), either ILP or the heuristic with the provided runtime constraint can be used to obtain the final optimal or near-optimal design point.

Bench.	R_c	ILP		Heuristic		Area
		R	A	R	A	Error
JESP	3620000	3601732	674336	3607761	679977	0.84%
JEMP	3620000	3777429	673330	3604671	659262	-2.09%
JD	11000000	10904701	380336	10904701	380336	0%
MP3E	300000000	296562778	587288	254315103	589336	0.35%

Table 7.4: Runtime and Area of final designs

Table 7.4 shows the final optimal and near optimal designs obtained using our methodology with reasonable runtime constraints. R and A stand for the runtime and area of the final design respectively. The results for JEMP are illuminating - the ILP timed out and was not able to find a solution within the provided runtime constraint. However, the heuristic not only found a solution, but a better solution as there is an area savings of 2.09%. Note that negative error points are not included in the calculation of average values for Table 7.3. Comparing the JEMP with JESP, for the same runtime constraint area for JEMP is 659,262 (column 6) while the area for JESP is 674,336 (column 4) which shows that multiple pipeline design is better for the JPEG encoder. This is because processing Y, Cb and Cr components of a macro block in parallel increases throughput. Therefore, simpler ASIP configurations can be used for JEMP to achieve the same system runtime but with reduced area footprint. Thus, different pipelined implementations of a single application can also be explored using the presented design flow. Another interesting result is the 0% area overhead for JD benchmark, which shows that in some cases the heuristic could well find the same solution.

We also implemented the ILP formulation presented in [17] for the purpose of comparison. Since the approach in [17] is only applicable to single pipeline systems, we compared the exploration time for JESP only. The worst solution time for the ILP formulation in this paper is 11 secs as opposed to 4 mins for the

formulation presented in [17]. This speed up is due to the use of fewer variables.

8 CONCLUSION

We presented a framework for efficient implementation of application specific heterogenous pipelined multiprocessor systems. A formal methodology consisting of a 0-1 ILP formulation and a heuristic for design space exploration is presented. A design space pruning algorithm is also used which can enable the use of ILP for large design spaces. Combining ILP and the heuristic in a single framework provides a flexible design flow to a designer to obtain optimal heterogeneous multiprocessor systems for different applications. Our experiments show that the average area error for all the benchmarks when using the heuristic is less than 2.5% and thus, the heuristic can be used for rapid design space exploration. Using the proposed design flow, we were able to explore and obtain optimal or near-optimal designs in one or two days for design spaces in order of 10^{12} design points. As future work, we will develop techniques for automatic application partitioning and will integrate it into the proposed methodology.

Bibliography

[1] Altera Nios Processor. Altera Corp. (http://www.altera.com).

[2] ARC the leader in configurable processor technology. ARC International (http://www.arc.com).

[3] Xtensa Processor. Tensilica Inc. (http://www.tensilica.com).

[4] S. L. Shee, A. Erdos, and S. Parameswaran. Heterogeneous multiprocessor implementations for jpeg:: a case study. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 217–222, New York, NY, USA, 2006. ACM.

[5] M. Strik, A. Timmer, J. van Meerbergen, and G.-J. van Rootselaar. Heterogeneous multiprocessor for the management of real-time video and graphics streams. *Solid-State Circuits, IEEE Journal of*, 35(11):1722–1731, Nov 2000.

[6] A. Beric, R. Sethuraman, C. Pinto, H. Peters, G. Veldman, P. van de Haar, and M. Duranton. Heterogeneous multiprocessor for high definition video. *Consumer Electronics, 2006. ICCE '06. 2006 Digest of Technical Papers. International Conference on*, pages 401–402, 7-11 Jan. 2006.

[7] T. Kodaka, K. Kimura, and H. Kasahara. Multigrain parallel processing for jpeg encoding on a single chip multiprocessor. In *IWIA '02: Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'02)*, page 57, Washington, DC, USA, 2002. IEEE Computer Society.

[8] S. Banerjee, T. Hamada, P. Chau, and R. Fellman. Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems. *Signal Processing, IEEE Transactions on*, 43(6):1468–1484, 1995.

[9] J. Jeon and K. Choi. Loop pipelining in hardware-software partitioning. In *Asia and South Pacific Design Automation Conference*, pages 361–366, 1998.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stien. *Introduction to Algorithms*. MIT Press and MCGraw-Hill, Second edition, 2001.

[11] J. DeSouza-Batista and A. Parker. Optimal synthesis of application specific heterogeneous pipelined multiprocessors. *Application Specific Array Processors, 1994. Proceedings., International Conference on*, pages 99–110, 22-24 Aug 1994.

[12] S.-R. Kuang, C.-Y. Chen, and R.-Z. Liao. Partitioning and pipelined scheduling of embedded system using integer linear programming. In *ICPADS '05: Proceedings of the 11th International Conference on Parallel and Distributed Systems - Workshops (ICPADS'05)*, pages 37–41, Washington, DC, USA, 2005. IEEE Computer Society.

[13] M. Schwiegershausen and P. Pirsch. A formal approach for the optimization of heterogeneous multiprocessors for complex image processing schemes. In *EURO-DAC '95/EURO-VHDL '95: Proceedings of the conference on European design automation*, pages 8–13, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[14] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha. Synthesis of application-specific heterogeneous multiprocessor architectures using extensible processors. In *VLSID '05: Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*, pages 551–556, Washington, DC, USA, 2005. IEEE Computer Society.

[15] J. Cong, G. Han, and W. Jiang. Synthesis of an application-specific soft multiprocessor system. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 99–107, New York, NY, USA, 2007. ACM.

[16] S. L. Shee and S. Parameswaran. Design methodology for pipelined heterogeneous multiprocessor system. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 811–816, New York, NY, USA, 2007. ACM.

[17] H. Javaid and S. Parameswaran. Synthesis of heterogeneous pipelined multiprocessor systems using ilp: jpeg case study. In *CODES/ISSS '08: Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 1–6, New York, NY, USA, 2008. ACM.

[18] Flix: Fast relief for performance-hungry embedded applications, 2005. Available at: http://www.tensilica.com/pdf/FLIX_White_Paper_v2.pdf.

[19] XPRES Generated Specialized Operations, 2005. Available at: http://tensilica.com/pdf/XPRES%201205.pdf.

[20] lp_solve. Available at: http://lpsolve.sourceforge.net/5.5/.