Runtime Assignment of Reconfigurable Hardware Components for Image Processing Pipelines

Heather Quinn[1], L.A. Smith King[2],
[1]Dept. of Electrical and Computer Engineering
Northeastern University
Boston, MA 02115, USA
hquinn, mel, meleis@ece.neu.edu

Miriam Leeser[1], Waleed Meleis[1]
[2]Dept. of Mathematics and Computer Science
College of the Holy Cross
Worcester, MA 01610, USA
LA@cs.holycross.edu

Abstract

The combination of hardware acceleration and flexibility make FPGAs important to image processing applications. There is also a need for efficient, flexible hardware/software codesign environments that can balance the benefits and costs of using FPGAs. Image processing applications often consist of a pipeline of components where each component applies a different processing algorithm. Components can be implemented for FPGAs or software. Such systems enable an image analyst to work with either FPGA or software implementations of image processing algorithms for a given problem. The pipeline assignment problem chooses from alternative implementations of pipeline components to yield the fastest pipeline. Our codesign system solves the pipeline assignment problem to provide the most effective implementation automatically, so the image analyst can focus solely on choosing components which make up the pipeline. However, the pipeline assignment problem is NP complete. An efficient, dynamic solution to the pipeline assignment problem is a desirable enabler of codesign systems which use both FPGA and software implementations. This paper is concerned with solving pipeline assignment in this context. Consequently, we focus on optimal and heuristic methods for fast (fixed time limit) runtime pipeline assignment. Exhaustive search, integer linear programming and local search methods for pipeline assignment are investigated. We present experimental findings for pipelines of 20 or fewer components which show that in our environment, optimal runtime solutions are possible for smaller pipelines and nearly optimal heuristic solutions are possible for larger pipelines.

1 Introduction

With an increasing amount of image data accessible to image analysts, there is a need for efficient, flexible image processing systems. In our system, we have a library of image processing algorithms which are combined at runtime to create the image processing application [12]. This "lego block" approach has advantages over monolithic systems which implement one-off applications. This environment takes advantage of the hardware acceleration and programmability benefits of reconfigurable hardware devices, such as Field Programmable Gate Arrays (FPGAs), while taking into account the device's limitations. FPGA implementations of image processing algorithms (e.g. median filter or edge detection) can be two to three orders of magnitude faster than software implementations of the same algorithms, but also incur costs for initialization, data transfer, and reprogramming. Consider the image analyst sitting at a workstation with several images to process. With our system, for each image the analyst can combine existing algorithms dynamically into the best implementation, then process the image.

Image processing is often decomposed into a series of components called a *pipeline*, where each component in the pipeline implements a different algorithm and pipeline components can be implemented with FPGAs or software. Consequently, the image analyst is concerned with both the composition of the pipeline and choosing which implementation to use for the individual components. We want our system to choose the implementation automatically so the analyst can focus solely on choosing components which make up the pipeline.

Choosing from alternative implementations for each component to yield the fastest pipeline is called the *pipeline assignment problem*. Our system interposes a decision layer between the user input and the pipeline execution steps that solves the pipeline assignment problem to provide the most

effective implementation automatically. Given image properties, the desired pipeline, and a set of component implementations, the codesign system should decide the pipeline assignment. Secondly, the pipeline assignment problem needs to be solved dynamically. The pipeline composition and image properties are not static, and even the set of possible implementations may not be known in advance. For example, when FPGA devices are shared, a particular FPGA device may be already in use and be unavailable at image processing time, making software implementation the only choice. There is, also, no way to efficiently pre-construct all possible pipelines, therefore constructing pipelines dynamically in real time insures fast pipeline implementations with minimal designer involvement. An efficient, dynamic solution to the pipeline assignment problem is a desirable enabler of codesign systems which use both FPGA and software implementations. This research is concerned with solving pipeline assignment in this context.

Unfortunately, the pipeline assignment problem is NP-complete.[1] FPGA devices commonly incur overhead costs, such as initialization, extra processing steps for border cases, and costs of moving the image to and from the device. Because of overhead costs, there is a crossover point (CP) for each component where all smaller images are processed faster in software and all larger images are processed faster in hardware. Often there will be a range of image sizes near the CP where the difference between the hardware and software runtimes are insignificant.

Deciding between available hardware and software implementations for a single component processing is a simple function of image size. Choosing the composition for a pipeline is much more difficult. If each component has one hardware and one software implementation, a pipeline with N_s components will have 2^{N_s} implementations. Furthermore, the cost of using a particular component implementation may vary depending on composition of the rest of the pipeline. For example, there is a limit to how much logic can be programmed into an FPGA which limits the number of components an FPGA can implement simultaneously. If this limit is exceeded, using the FPGA incurs a reprogramming cost. Thus the same FPGA implementation's cost will vary between pipelines that incur reprogramming costs and those that don't. Furthermore, some algorithms benefit more from FPGA implementations than others. For a given pipeline, the optimal assignment might choose a slightly slower software implementation for one component in order to avoid reprogramming costs and keep another component which benefits more from hardware acceleration implemented with an FPGA device. The codesign system should handle this level of implementation detail so the image analyst can focus on pipeline composition instead.

[1] Proof available from the authors.

The remainder of the paper is organized as follows. Section 2 gives an example of the pipeline assignment problem. The pipeline assignment problem is defined more formally in Section 3, including associated constraints and restrictions. In Section 4, exhaustive search, integer linear programming, and local search methods are applied to solving the pipeline assignment problem defined in Section 3. Experimental results for pipelines of 1 to 20 components are also given and discussed in Section 5. Section 6 highlights the pipeline assignment problem in the context of our codesign system [12]. Related work is discussed in Section 7. Finally, Section 8 describes conclusions and directions for future work.

2 Pipeline Assignment Example

Consider a simple two component pipeline our image analyst might use consisting of Median Filter→Edge Detection as shown in Figure 1. Figure 1 shows the four combinations of implementations of the Median Filter→Edge Detection pipeline, and depicts costs associated with each permutation. The components are depicted with rectangles. Communication costs are incurred at software/hardware boundaries, denoted as octagonal boxes. Other costs, such as padding the image with an extra border or reprogramming the hardware, are shown as ovals and rectangles respectively.

Since both algorithms use window-based processing, the border cases need to be determined either before or during execution. For the components implemented with FPGA hardware, padding the image with an extra border before execution starts is more efficient than determining the border cases when the windows are being formed while processing. When needed, *image padding* is an extra processing step to either add, fix or remove the border cases when hardware implementations are used.

Experimental results can be used to define performance *profiles* for all implementations of a component as a function of image size. The performance profile can be used to predict the performance of a component implementation on an arbitrary image of a given size. Table 1 shows the software and hardware runtime profiles for both median filter and edge detection components as a function of image size (x). For this example, initialization and other costs are included in this table. The data are based on the average of several runs. The execution times between runs vary for many reasons, including differences in communication costs, and thread synchronization. Table 1 also shows the crossover point (CP). A component processes all images smaller than its CP faster in software, and all larger images can be processed faster in hardware. Also note that the equations in Table 1 show that Edge Detection benefits more from hardware acceleration than Median Filter, which

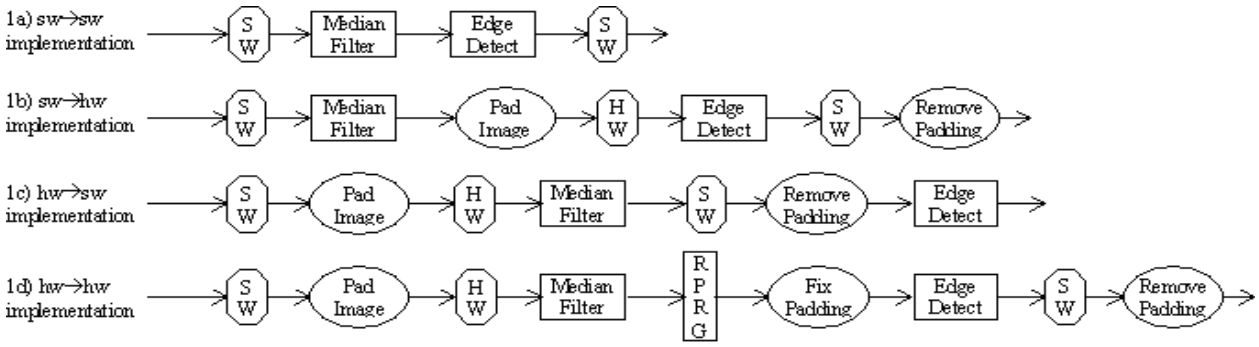

Figure 1. The Processing Steps for Each of the Four Hardware/Software Assignments of Median Filter→Edge Detection

gives Edge Detection a smaller crossover point.

The profiles for the components and the costs can be used to predict the profile of the pipeline. The profiles of all four permutations of the Median Filter→Edge Detection pipeline are shown in Table 2.[2] These equations combine component profiles with other costs incurred with the given pipeline permutation. The equations predict that an all hardware implementation will have the fastest runtime for images greater than 579 pixels. All smaller images run fastest if processed in an all software implementation. A more complicated pipeline would have ranges where mixed implementations with software and hardware component implementations would be optimal.

Algorithm	SW Runtime (ms)	HW Runtime (ms)	Crossover Point (pixels)
Median Filter	$0.08x + 169.72$	$0.003x + 533$	3600
Edge Detection	$0.15x + 65.06$	$0.003x + 358$	1680

Table 1. Software/Hardware Runtimes as a Function of the Number of Pixels (x)

The predicted performance of the pipeline assumes that component profiles and other costs are additive. This estimates the total runtime for the pipeline conservatively. Table 3 compares the predicted runtime with the actual runtime for two permutations of the Median Filter→Edge Detection pipeline for 13 images. In this case, the estimated values are higher by an average of 12% from the actual values. In general, the predicted pipeline performance is a worst case estimate.

Permutation	Runtime (ms)
SW→SW	$0.2332x + 234.78$
SW→HW	$0.0855x + 472.2$
HW→SW	$0.1527x + 367.91$
HW→HW	$0.0026x + 372.48$

Table 2. Runtime as a Function of the Number of Pixels (x)

pic	size	HW/SW	SW/HW
copy	576	15.5	3.1
cross	1400	2.4	15.7
bb0020	3600	11.7	33.8
logo5	6440	12.8	25.4
w0256	7900	17.5	24.9
i0015	10000	.06	2.4
w0020	11682	21.8	16.2
w0235	12772	20.3	24.2
w0371	17690	2.9	1.7
w0052	19320	1.3	3.1
bb0013	20736	0.2	2.1
o0057	22140	3.4	2.0
w0227	31488	2.3	1.8

Table 3. Absolute Value of the Percentage Difference of Estimated Values from Actual Values for Two Implementations of Median Filter → Edge Detection Pipeline

This simple Median Filter→Edge Detection pipeline illustrates some of the issues and complexity involved in pipeline assignment, and motivates the desirability of a

[2]Note that because we included other costs in Table 1, the equations in Table 2 were derived independently, i.e. one cannot just add pipeline costs to the equations of Table 1 to derive the pipeline profile equations in Table 2.

hardware/software codesign system which handles pipeline assignment automatically.

3 The Pipeline Assignment Problem

This section formally states the pipeline assignment problem. Because we are concerned with dynamic solution of pipeline assignment in the context of a codesign system [12], the assignment decision must be fast or decision time will dominate image processing time. Specifically, we have a strict time bound on the amount of processing time spent on pipeline assignment. We also assume realistic pipelines are unlikely to have more than a small, finite number of components. Consequently, we want to quickly find near-optimal solutions for small instances of the pipeline assignment problem. We focus on methods of doing pipeline assignment at runtime within a fixed time constraint of 500 ms for pipelines with 20 or fewer components.

3.1 General Problem Statement

This section summarizes the problem statement and current assumptions. The initial assumptions relevant to pipeline assignment are:

1. Communication costs are incurred at the beginning and end of each hardware subsequence (i.e. a series contiguous hardware implementations).

2. One hardware and one software implementation for a component exist in a library and components can be combined into pipelines at runtime.

3. Reprogramming cannot occur within a hardware subsequence.

4. Image padding issues (e.g. for border cases) are always addressed at component boundaries as specified in Table 4.

Case	Action	Implemented In
HW to HW	fix the image padding	HW
HW to SW	remove image padding	SW
SW to HW	pad the image	SW
SW to SW	no changes	N/A

Table 4. Different Cases for Image Padding

Although we restrict ourselves to at most one hardware (e.g. FPGA) and one software implementation in this paper, this restriction could be relaxed. It would just make the problem space larger, and require heuristic methods for smaller pipelines. Likewise, we restrict the amount of time allotted for the assignment decision to an arbitrary 500 ms to keep the ratio of decision time to runtime small. For small pipelines this value may be too large, as the pipeline runtime can be less than 50 ms. In the future we will explore making the decision time a variable that either the user supplies or which is based on the length of the pipeline.

We predict the performance of the pipeline using the performance profiles of the component implementations and profiles for any extra costs (e.g. reprogramming and initialization). We assume component runtimes and all extra costs are additive. These inputs to the pipeline assignment problem are a set (library) of components, a pipeline, a total area constraint for hardware subsequences, and image size. Inputs are defined more formally below.

3.2 Components

Each component represents an image processing algorithm. The set of all components is called the library and is denoted ***C***. All components in ***C*** have an interface consisting of data input and data output. Any two components in ***C*** can be ordered (connected) in sequence, and an ordering over a subset of ***C*** forms a pipeline.

There exists a set of implementations *I(c)* for each component *c* in ***C***. Implementations are classified as either hardware or software. A component has one or more implementations of each type such that the size of *I(c)* is greater than or equal to two for all *c* in ***C***.

The set of all the implementations for all the components in ***C*** is denoted ***UI*** (the union of all I's). For each implementation in ***UI*** there exists an area requirement, *A(I)*. All software implementations have $A(I) = 0$. For hardware implementations, *A(I)* is a positive, non-zero integer. For each implementation in ***UI*** there also exists a time cost *T(I)* which is an upper bound on the execution time of the implementation as a function of image size. For all implementations, *T(I)* is a positive, non-zero integer number of milliseconds and varies according to image size.

3.3 Pipeline

A pipeline is an ordered sequence of components (members of ***C***). Every pipeline starts with a *source* component and ends with a *sink* component. The *source* node represents all of the preprocessing steps for the pipeline, such as receiving the pipeline and image input from the user, and converting the image into a data array. The *sink* node represents all of the post processing steps, such as displaying the image. The total execution time from source to sink is called *latency*. The goal of the pipeline assignment problem is to minimize latency.

For all pairs of implementations *(i,j)*, where *i* and *j* are members of ***UI***, there exists a coupling cost, *X_cost*. *X_cost*

Case	i	j	X_cost
1	SW	SW	0
2	SW	HW	Communication and reprogramming costs
3	HW	SW	Communication costs
4	HW	HW	0

Table 5. Different Cases of X_cost

represents overhead costs associated with using these two implementations in series, such as communication or reprogramming costs. The different cases for a pair *(i, j)* are shown in Table 5. *X_cost* values can be used to implicitly order the pipeline. If the cost for all values where $j \neq i + 1$ is infinite, a pipeline implementation with a different ordering cannot be a solution to the problem.

Note that there is a special initialization cost associated with the first time a hardware implementation is used. Any pipeline that uses hardware implementations must include the first time initialization cost in the total latency. We assume the reconfigurable device is programmed before it is used.

3.4 Area Constraint

There are limits to how much logic can be programmed into an FPGA which limits how many components can be implemented in an FPGA simultaneously. This limitation is modeled as the total area constraint, *TAC*, which is a positive, non-negative integer. Every component in the pipeline is associated with an area cost. Each FPGA device has its own *TAC*, so the pipeline assignment solution for different FPGA devices may differ. The sum of all of the area costs for the components in a hardware subsequence must be less than the *TAC*. If the area of the device is exceeded, the device can be reprogrammed but this adds a reprogramming cost to the total cost of the pipeline. Reprogrammability allows for greater latitude of assigning components to hardware, which is one of the benefits of using reconfigurable technology in a hardware/software system. We restrict reprogramming steps to software/hardware boundaries, which is easier to model. In the future, allowing reprogramming steps in the middle of a hardware subsequence will be supported.

3.5 Decision Problem Statement

The decision problem statement for the pipeline assignment problem is:

> *Given a pipeline and positive integers k, and n, is there an implementation that has total area $\leq$ k, and has total execution time $\leq$ n ms?*

4 Optimal and Heuristic Pipeline Assignment

In solving the pipeline assignment problem defined in Section 3, two methods yielding optimal solutions are considered: exhaustive search and integer linear programming (ILP). Although optimal, these methods do not scale with the length of the pipeline. A heuristic method, such as local search, must be used when an optimal solution is not possible within the time constraint. In this section we present exhaustive search, integer linear programming and local search methods for solving pipeline assignment. In the next section we describe heuristics for selecting which solution to apply.

4.1 Exhaustive Search Algorithm

The exhaustive search algorithm for the pipeline assignment problem finds an optimal pipeline implementation with the minimum latency for the given area constraint. If the pipeline has N_s components and each component has one hardware and one software implementation, then there will be $N_p = 2^{N_s}$ implementations. This algorithm finds the optimal solution by computing the area and latency of every possible permutation of pipeline implementations. The search space can be pruned by rejecting permutations whose area does not meet the total area constraint. The best and worst case scenario for this algorithm is based on the number of permutations that pass the area test. The lower bound on execution time for the exhaustive search algorithm occurs when only one permutation passes the area test. The upper bound on execution time occurs when every permutation passes the area test. The time complexity of this algorithm is $O((N_s)2^{N_s})$. The algorithm is outlined in Figure 2.

```
ExhaustiveSearch (pipe, totalArea) {
  MinLatency = inf;
  For i = 0 to i < Np {
    Perm = generateNextPermutation(pipe, i);
    If (passesArea(perm, totalArea)) {
      latency = calcLatency(perm);
      If (latency < minLatency)
              MinLatency = latency;
    }
  }
  return minLatency;
}
```

Figure 2. Exhaustive Search Algorithm

4.2 Integer Linear Programming

Integer Linear Programming (ILP) also finds optimal solutions. Unfortunately, ILP is too computationally complex

to solve all problem sizes. The ILP formulation for pipeline assignment uses the Niemann/Marwedel [9] ILP formulation as a basis. Although their formulation handles many of the same factors as pipeline assignment, such as communication, execution time and area costs, there are some differences because area costs can be reset via reprogramming. The AMPL language [1] was used to describe the formulation and CPLEX [2] to solve the model.

Objective Function

The objective of the ILP formulation is to find an implementation that minimizes the sum of the execution, initialization, communication, and reprogramming costs while satisfying the area requirements.

Variables

There are three variables in the formulation. The variable *assignment* defines which implementation a component is assigned to and is an $I \times N$ matrix, where I is the number of implementations and N is the number of pipeline stages.

$$assignment[i,j] = \begin{cases} 1 & \text{component } j \text{ is mapped to implementation } i \\ 0 & \text{otherwise} \end{cases}$$

The variable *interface* defines what type of interface two components have based on *assignment* and is a $I^2 \times (N-1)$ matrix. The types of interface are: hardware/hardware, hardware/software, software/hardware, and software/software.

$$\textit{interface}[i,j] = \begin{cases} 1 & \text{component } j \text{ and } j+1 \text{ have interface type } i \\ 0 & \text{otherwise} \end{cases}$$

The variable *haveHW* indicates the presence of hardware assignments in *assignment*, which determines whether the first time initialization cost needs to be paid.

$$haveHW = \begin{cases} 1 & \text{at least one component assigned to HW} \\ 0 & \text{otherwise} \end{cases}$$

These variables are all assigned to integer values between 0 and 1 for ILP solutions.

Constraints

There are several constraints in this formulation:

- The sum of all assignments for a component is 1 and the sum of all interfaces for a component is 1.
- Source and sink components are only assigned to software.
- The sum of the entries for the *assignment* variable is equal to N.
- The sum of the entries for the *interface* variable is equal to $N-1$.
- The sum of the area costs for all hardware subsequences is less than or equal to the total area constraint.
- The entries in the *interface* matrix and the *haveHW* variable are consistent with the entries in the *assignment* matrix.
- None of the variables are negative.

There is one major drawback when using ILP. If the CPLEX solver cannot find a solution within the allotted time, it returns no solution. In cases where no solution is returned, the pipeline assignment problem will need to be solved a second time using another algorithm.

4.3 Local Search

Local Search algorithms take an initial solution and try to improve it in hopes of finding the optimal solution. There are many different local search algorithms; simulated annealing is the most common. Pseudocode for a generic local search algorithm is shown in Figure 3.

We consider three different approaches for our initial solution:

- A greedy algorithm which traverses the pipeline from source to sink taking the fastest implementation for each component that still meets the constraints at each step.
- A random algorithm which translates randomly generated numbers between 1 and 2^N into implementations and returns the first one that meets the constraints.
- An all software approach that implements all components in software.

An important feature of local search is that at any point in time after initialization, local search can return a solution, although the solution can be sub-optimal. This makes local search attractive when there is a time constraint, since local search can always return the best solution found when time expires. This feature is in contrast to ILP, which can sometimes complete without a solution.

The improvement step iterates, beginning with the initial solution, and uses either a 2-opt or 3-opt neighborhood and either steepest descent or a tabu list [3]. Steepest descent only considers solutions with smaller latency as an improvement, whereas tabu considers the implementation

```
LocalSearch(pipe, totArea, time) {
  int bestTime = inf;
  Perm bestSol = initSolution(pipe, totalArea);
  Start, end = currentTime;
  boolean done = false;
  while (!done && end - start < time){
    Perm imprvPerm = improve(bestSol, totArea);
    if (imprvPerm != null) {
      int newTime = calcTime(improvePerm);
      if (newTime < bestTime) {
bestTime = improveTime;
bestSol = imprvPerm;
      }
    }
    else
    done = true;
    end = currentTime;
  }
  return bestTime;
}
```

Figure 3. Local Search Algorithm

with the smallest latency that isn't on the tabu list an improvement. Therefore, tabu does not often naturally terminate. For small pipelines steepest descent will usually terminate in a local minima within 500 ms and return the implementation of the minima. For tabu and occurrences of steepest descent that do not finish computation within our time bound of 500 ms, the best solution found by that point is returned.

The time complexity of the local search algorithm is $\Theta(m * l)$, where *m* is the runtime of the neighborhood function and *l* is the number of times the while loop iterates. The 2-opt neighborhood has time complexity $\Theta(N_s^2)$ and 3-opt has $\Theta(N_s^3)$ to completely explore the neighborhood. The number of times the while loop iterates is not predictable. The while loop will continue to iterate if the improve function can find a solution and the time limit has not been exceeded.

5 Experimental Results

All the methods described in Section 4 can be used to solve the pipeline assignment problem. However, these algorithms perform differently as the number of components in the pipeline increases. A heuristic is needed for determining which method to use on a given pipeline.

Each algorithm's behavior can be studied by using it to solve a number of different pipeline assignment problems, and then comparing the observed behaviors of each algorithm. This requires a variety of pipeline assignment problems with different characteristics as input. Characteristics to be varied include: number of pipeline components, total area constraint, communication costs and component profiles for hardware and software implementations.

5.1 Testbench

We use a testbench designed to simulate different library components by using synthetic components that have hardware/software speedup and communication costs variables. The testbench covers many different scenarios. There are seven different execution models for generating hardware/software speedup data for each problem (pipeline) size and three different communication models for generating communication cost data which mimic transferring small, medium and large images. Each of the 21 combinations of execution model and communication costs is called a *test scenario*.

In this paper we present experimental results based on seven test scenarios: all seven execution models with the communication model for small images. For each test scenario, the pipeline size was varied from 1 to 20, and each pipeline size was associated with a unique total area constraint and other costs. The total area constraint was created by randomly generating a Gaussian value, multiplying it by 10,000, and rounding up to the nearest integer. The small image communication model generates randomly generated values between 1 and 50 milliseconds as the communication model.

5.2 Experiments

The seven test scenarios for pipeline sizes between 1 and 20 were solved with exhaustive, ILP, and local search algorithms. Local search was run multiple times to cover all combinations of 2-opt or 3-opt neighborhood with either tabu or steepest descent search. For the tabu version of local search, the tabu list tracks the last seven solutions found. Since local search can often run indefinitely, it was constrained to run for only 500 ms. The exhaustive and ILP algorithms were run to completion to yield the optimal solution for comparison with local search solutions.

To determine the quality of the solutions returned by local search, the results were compared with the optimal solution for each problem size. Table 6 shows the average difference for the seven test scenarios in latency between the derived and optimal solution. The differences are normalized to the optimal solution. We call this value the Average Solution Quality (ASQ). In this table, the larger the difference between ASQ and 1, the less optimal the solution.

For these tests, exhaustive search can find optimal solutions in 500 ms for pipelines of size 11 or smaller. ILP can find optimal solutions in 500 ms for pipelines of size 13 or smaller. Above problem size 13, ILP is unreliable as it does not always complete within 500 ms. When ILP fails to complete, it does not return any (even a sub-optimal) solution. For this reason, we are interested in heuristic solutions for pipelines larger than size 13 as seen in Table 6. The local

Local Search Method	Pipeline Length						
	14	15	16	17	18	19	20
Greedy, 2-Opt, Steepest	1.00	1.00	1.00	1.00	1.01	1.06	1.00
Greedy, 3-Opt, Steepest	1.00	1.00	1.00	1.00	1.03	1.08	1.02
Random, 2-Opt, Steepest	1.00	1.00	1.00	1.00	1.00	1.01	1.06
Random, 3-Opt, Steepest	1.00	1.06	1.06	1.40	1.27	1.42	2.09
All SW, 2-Opt, Steepest	1.00	1.00	1.00	1.00	1.00	1.05	1.06
All SW, 3-Opt, Steepest	1.00	1.02	1.04	1.10	1.11	1.30	1.24
Greedy, 2-Opt, Tabu	1.00	1.00	1.00	1.00	1.01	1.01	1.00
Greedy, 3-Opt, Tabu	1.00	1.00	1.00	1.00	1.03	1.08	1.02
Random, 2-Opt, Tabu	1.00	1.00	1.00	1.00	1.00	1.06	1.10
Random, 3-Opt, Tabu	1.10	1.20	1.18	1.25	1.27	1.21	2.15
All SW, 2-Opt, Tabu	1.00	1.00	1.00	1.00	1.00	1.05	1.06

Table 6. Average Solution Quality for Local Search Based on the Average of Seven Test Scenarios. The Left Column Is in the Form of "Initial Solution, K-Opt, Steepest/Tabu".

search algorithm using a greedy initial solution, a tabu list, and a 2-opt neighborhood is the best algorithm to use for all problem sizes larger than 13.

6 Hardware/Software Codesign System

We are developing a hardware/software codesign environment [12] in which pipeline assignment is solved dynamically at runtime. In this architecture, the pipeline assignment problem is handled by a decision layer, as shown in Figure 4. The decision layer consists of two steps. The first step decides how the pipeline should be implemented, and the second one implements the pipeline. The first process is called the Software/Hardware Runtime Procedural Partitioning (SHARPP) tool and the other is called Runtime Interfacing for Pipeline Synthesis (RIPS).

6.1 SHARPP

The software/hardware Runtime Procedural Partitioning (SHARPP) step actually solves pipeline assignment. As shown in Figure 4, SHARPP receives the pipeline and image size as specified by the image analyst. In addition, SHARPP has a library of available implementations (e.g. FPGA or software) for each component, the performance profiles for each component implementation, and the other cost information associated with combining different component information in a pipeline.

As described previously, SHARPP will use different algorithms to solve pipeline assignment depending on the problem size and will need to move seamlessly between these algorithms. These include the exhaustive search, integer linear programming and local search methods. SHARPP uses the results such as those described in Section 5 to determine when to switch algorithms.

The SHARPP tool applies the best search algorithm to the pipeline assignment problem to determine the pipeline implementation, and passes the solution to the RIPS tool.

6.2 RIPS

The RIPS tool translates the assignment received from the SHARPP into the pipeline implementation. First, the assignment is parsed to determine the implementation and ordering for all the components in the pipeline. The pipeline is then constructed by connecting the components' implementations together in the correct order. This last step also ensures data flows correctly from one component to the next. For example, image size can change as processing flows from one component to the next. We plan to use a rule-based methodology for handling these types of interfacing issues.

The pipelines are constructed at runtime because creating pre-constructed pipeline implementations for all the possible combinations of components is not feasible. Instead, implementations for the components are designed with well-defined interfaces such that image processing components can be classified by interface types with known rules for combining interface types. RIPS uses the image processing Basic Library of Components (ipBLOC) to construct pipelines from existing components.

We currently have a prototype of RIPS that uses pre-constructed pipeline implementations. This early system allows us to test our ability to parse SHARPP's output. Dynamic pipeline construction from ipBLOC components is future work.

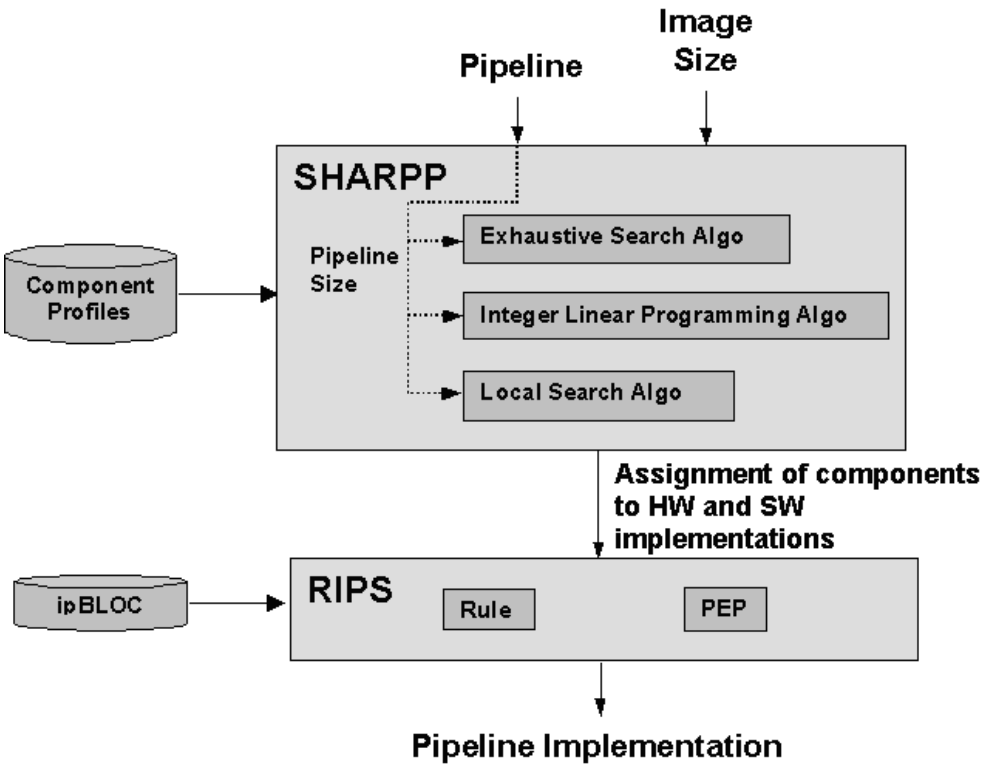

Figure 4. Decision Layer

7 Related Work

Many of the techniques used to solve the hardware/software partitioning problem can also be applied to the pipeline assignment problem. Neimann and Marwedel [9] use Integer Linear Programming to solve the partitioning problem. Their work is relevant to our work since they also consider interface costs. The COSYMA project [6] uses simulated annealing, which is a local search technique. The Ptolemy project [7] uses a process that is similar to our greedy algorithm, but their algorithm is applied iteratively while ours is not. The Hybrid System Architecture Model (HySAM) [4] uses dynamic programming to map applications to abstract reconfigurable platforms that represent different reconfigurable scenarios and focuses on optimizing loops.

Fundamental differences exist between pipeline assignment and hardware/software partitioning. First, the partitioning problem estimates execution time and area costs from a codesign system specification. Pipeline assignment uses component profiles, so the execution time and area costs are known at decision time. Second, overhead costs of the hardware/software interface usually do not enter into hardware/software partitioning whereas these costs, including reprogramming, are an essential part of the pipeline assignment problem. Finally, our focus on solving pipeline assignment dynamically at runtime is a major difference.

Interface synthesis is concerned with communication between two or more systems. This problem is very important in hardware/software systems, but is a problem that extends to pure software and pure hardware systems as well. The RIPS tool is a form of interface synthesis. The Chinook project [5] tries to minimize the amount of I/O pin overlap when using a coprocessor in a hardware/software system. The PIG project [11] interfaces two components with incompatible protocols. The LYCOS system [8] builds interfaces for one side of the interface but forces the interface of the component to comply to the interface protocol. A fundamental difference between these projects and our work is our emphasis on dynamic construction of interfaces at runtime.

Our project addresses applications similar to those handled by the CHAMPION project [10]. In terms of applications, the main difference is that our project maps image processing pipeline components to either hardware or software, while CHAMPION provides an environment for developing image processing pipelines in hardware only. Lastly, the two projects are complementary because

CHAMPION is a design-time tool, while our project emphasizes dynamic decisions and run-time execution.

8 Conclusion and Future Work

Effective hardware/software codesign systems for image processing applications enable image analysts to take advantage of FPGA hardware acceleration and reprogrammability. Image processing applications are often composed of pipelines of individual image processing components which can be implemented in hardware or software. Thus, the pipeline assignment problem is an important consideration for hardware/software codesign using FPGAs. Solutions to the pipeline assignment problem at runtime that are both fast and near-optimal are needed. We considered pipelines of 20 or fewer components with a fixed time bound (500 ms) for decision time, and found that exhaustive search and ILP techniques yield optimum solutions for pipelines of 13 or fewer stages, and near-optimum solutions for longer pipelines are possible using a local search method based on a greedy initial solution, with a 2-opt neighborhood and tabu list.

In the future, we plan to explore more general pipeline assignment problem spaces. One area is to relax the constraints on pipeline assignment and explore the impact on the solution space. For example, the decision time constraint can be made variable. Also, the number of implementations can be expanded by allowing more than two implementation alternatives (e.g. allowing several hardware implementations). In addition, we will also allow reprogramming within hardware subsequences (see Section 3.4). We will exploit more parallelism inherent in the pipeline. For example, the cost of reprogramming can be reduced by reprogramming a FPGA device in parallel with software image processing, in for example, a hardware→software→hardware pipeline where the hardware component is reprogrammed while the software component is processing.

We will fold the current results into the SHARPP component of our codesign system, and gather additional experimental results for the pipeline assignment problem. The SHARPP tool will be enhanced to use dynamic information about the availability of hardware devices; this will yield a robust system that can work in shared environments where FPGA hardware may be temporarily unavailable. The RIPS tool will be enhanced to enable full dynamic construction of pipelines. We also intend to use the system on realistic image processing applications and more complex reconfigurable devices. For example, pipeline assignment may be used to assign a pipeline to either reconfigurable logic or an embedded processor in new chips such as the Xilinx Virtex II Pro FPGA. This chip includes up to four Power PCs as well as reconfigurable logic on one chip.

References

[1] AMPL: A modeling language for mathematical programming. Available on the web at http://cm.bell-labs.com/cm/cs/what/ampl. Last visited in June 2002.

[2] ILOG CPLEX. Available on the web at http://www.ilog.com / products / cplex/. Last visited in June 2002.

[3] E. Aarts and J. Lenstra. *Local Search in Combinatorial Optimization*. Wiley, 1997.

[4] K. Bondalapati, G. Papavassilopoulos, and V. Prasanna. Mapping applications onto reconfigurable architectures using dynamic programming. In *Military and Aerospace Programmable Logic Device (MAPLD) International Conference*, 1999.

[5] P. Chou, R. Ortega, and G. Boriello. Synthesis of the hardware/software interface in microcontroller-based systems. In *Proceedings of the International Conference on Computer Aided Design*, 1992.

[6] J. Henkel, R. Ernst, U. Holtmann, and T. Benner. Adaptation of partitioning and high-level synthesis in hardware/software co-synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, 1994.

[7] A. Kalavade and E. Lee. A gobal criticality/local phase driven algorithm for the constrainted hardware/software partitioning problem. In *Proceedings Third International Workshop on Hardware/Software Co-Design (Codes/CASHE '94)*, pages 42–48, Grenoble, France, Sept 1994.

[8] J. Madsen and B. Hald. An approach to interface synthesis. In *the 8th International Symposium of System Synthesis (ISSS)*, 1995.

[9] R. Niemann and P. Marwedel. Hardware/software partitioning using Integer Programming. In *In Proccedings of the European Design and Test Conference*, pages 473–480, Paris, France, 1996. IEEE Computer Society Press (Los Alamitos, California).

[10] S.-W. Ong and et al. Automatic mapping of multiple applications to multiple adaptive computing systems. In *The Proceedings of Field-Programmable Custom Computing Machines (FCCM)*, 2001.

[11] R. Passerone, J. Rowson, and A. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Design Automation Conference*, pages 8–13, 1998.

[12] L. Smith King, H. Quinn, M. Leeser, D. Galatopoullos, and E. Manolakos. Runtime execution of reconfigurable hardware in a Java environment. In *Proccedings of International Conference on Computer Design*, pages 380–5, Austin, TX, 2001.