# Runtime Tools for Hardware/Software Systems with Reconfigurable Hardware

A Thesis Presented
by

**Heather Marie Quinn**

to
The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

**Doctor of Philosophy**

in Electrical Engineering

**Northeastern University**
**Boston, Massachusetts**

October 2004

# Abstract

Hardware/software codesign tools help designers create efficient systems that use hardware for speedup. These tools include co-specification, co-synthesis, and co-verification. Co-synthesis tools, the primary focus of this dissertation, translate a co-specification into a hardware/software system. This process involves solving many intractable subproblems, such as partitioning a design into hardware and software, synthesizing interfaces between components, and allocating resources to hardware functions.

This dissertation focuses on hardware/software codesign tools for Field Programmable Gate Arrays (FPGAs). Reconfigurable systems are inherently hardware/software systems, since the host computer controls the reconfigurable device. Reconfigurable devices can exhibit speedups of up to three orders of magnitude over software; however, overhead costs, such as hardware initialization, communication, and reprogramming, must be kept to a small proportion of the overall runtime. Codesign tools for reconfigurable systems help designers advantageously use FPGA technology provided they take into account overhead as well as computation.

A set of tools were designed to support runtime codesign. The Software HArdware Runtime Procedural Partitioning (SHARPP) tool applies linear optimization methods to solve a variation of the hardware/software codesign partitioning problem for FPGAs called the pipeline assignment problem. The model for this optimization problem takes into account the overhead costs associated with using FPGA hardware and executing components serially. These overhead costs, which include communication and reprogramming, can be as much as 90%

of a pipeline assignment's latency and are not accounted for in other FPGA-based co-synthesis environments. The Runtime Interfacing for Pipeline Synthesis (RIPS) tool translates the SHARPP output into an executable. The Packet Exchange Platform (PEP) layer supports the runtime execution of pipeline components through abstract interfaces.

These tools are applied to the image processing domain. The tools have access to a library called the image processing Basic Library of Components (ipBLOC) that was developed as part of this research. The ipBLOC components are hardware and software implementations of common image processing algorithms. Java is used as a unifying platform for both the hardware and software design processes. The hardware description is done in JHDL, a Java-based Hardware Description Language for FPGAs.

# Contents

# List of Tables

# List of Figures

# Acknowledgements

I would like to start off by thanking the people who made my research possible. I would like to thank my advisor, Dr. Miriam Leeser, who gave me the space to let me explore what I wanted to do with my research project, but never let me wander too far off the path without guidance. The members of the Rapid Prototyping and Biomedical Statistical Signal Processing Laboratories have been my compatriots in research and coffee trips. I would like to thank Dr. Laurie Smith King, who shared an office with me, wrote papers with me and listened to me for the last four years. I would like to thank Dr. Waleed Meleis, who helped me with my combinatorial optimization research and proofs. I would like to thank Dr. Elias Manolakos, who served on both my masters and doctoral committees. I would like to thank the Configurable Computing Group at Brigham Young University, who developed JHDL and helped me in the early years of my research while I was designing hardware components.

I would like to thank Goddard Space Flight Center at the National Aeronautics and Space Administration for funding me for the last three years on a Graduate Student Researchers Program fellowship. The flexibility in this funding has truly allowed me to explore my research topic without limitations.

Outside of work and school I have a wonderful network of friends and family. My family has watched me through all my educational endeavors, even though they wondered when I am going to finish. I have been truly blessed with good friends. I would like to thank Beth and Kelly who gave me cookies and listened to me. I would like to thank Dean and FJ who helped me translate my thoughts into words over pots of tea at their house. I would like to thank Maigann,

Noreen, and Amanda for seeing me through so many of life's bumps. I would like to thank the dinner crowd for Sunday night dinners. I would like to thank Amy and Alan for being on the other side of the instant messenger dialog, the back up Tivo, and dance partners. I would like to thank Seth for helping me write when I was stuck and his wife Inna for being patient with both of us. I would like to thank my dissertation group for supporting my during the final years of my writing and research. I would like to thank the Jae H. Kim Tae Kwon Do Institute for giving me a place to be and the competition team for the motivation and the purpose to be there. I will miss you all during my travels.

I am forever grateful to my late grandmother, Ann Rowe, who was a force, a miracle and a heroine of an extraordinary life. She read many of my high school and college essays, and was greatly missed as I wrote this book. I dedicate this dissertation to her memory.

# Chapter 1

# Introduction

Hardware/software codesign tools can help designers create efficient systems that use reconfigurable devices for hardware acceleration. Designers can use these tools to efficiently implement algorithms and to achieve overall system speedup. This dissertation studies codesign tools that allow designers to combine already designed components at runtime to create simple applications.

## 1.1 Overview

Hardware/software systems are realized through a combination of hardware and software components. Hardware/software systems are becoming more common, as embedded technology becomes more widespread. Embedded technology can be found in automobiles, household appliances, cell phones, and personal digital assistants. In these systems both hardware and software implementations are needed due to timing requirements, computational complexity, or area constraints in the system specification. Some hardware devices, such as reconfigurable devices, are software controlled, which makes applications using these devices inherently hardware/software systems as the entire system cannot be realized in hardware. This dissertation addresses runtime tools that simplify the task of creating hardware/software systems that use reconfigurable technology.

This chapter introduces basic concepts for hardware/software codesign systems, and tools to help build codesign systems.

### 1.1.1 Hardware/Software Systems

Hardware/software systems are becoming more common, since there are many advantages to using both domains. When pure software systems do not meet timing requirements, a custom hardware design of one or more subsystems might provide enough speedup to meet the specification. The need for hardware acceleration has always made codesign systems popular for real time systems which need to quickly respond to inputs. On the other hand, software can simplify pure hardware systems. Software can do computations that are not well matched to the type of hardware being used. In addition, moving the finite state machine to software is often useful. Therefore, using hardware and software processing together can make efficient systems. This section introduces hardware/software systems. Common hardware/software architectures and tools to create hardware/software systems are covered.

There are many possible codesign architectures. The basic hardware/software system has at least one general purpose processor (GPP) and at least one hardware coprocessor. The hardware coprocessor can be implemented with any number of hardware technologies, including Application Specific Integrated Circuits (ASIC), Application Specific Instruction Processors (ASIP), and reconfigurable hardware. Our research focuses on an architecture of one GPP with a reconfigurable device called a Field Programmable Gate Array (FPGA). Over the last several years FPGAs have started to be used as ASIC replacements. This type of hardware is more flexible than ASICs, as the device can be reprogrammed to run different or improved algorithms. When algorithms need regular data access or have fine-grained parallelism, the FPGA implementation of the algorithm is often superior to an ASIC implementation.

How the hardware and software components are interfaced is an important aspect of the codesign architecture. In many codesign tools, the components

are modeled as being connected through a *channel*, such as the memory or PCI bus. Data is formatted according to a *protocol*. Many times either a software or hardware machine outside of the interfacing components controls the timing of the data using the channel. Since communication costs incurred when crossing hardware and software domains can be a large portion of a system's latency, the channel and protocol need to be designed to minimize latency. When using commercial off-the-shelf (COTS) products, where optimizing the channel might not be possible, the hardware/software system must be designed to mitigate communication costs.

The codesign field has been hampered by the complicated design process of creating a system in two processing domains. Codesign systems are more challenging to build than pure software or pure hardware systems. The designer needs to understand which aspects of the system are better suited for hardware than software. Efficient channels and protocols are needed so that switching between domains is possible without system degradation. These are a few of the many challenges that face designers of hardware/software systems: combining two design processes, partitioning a design between hardware and software, interfacing hardware and software components, and supporting multiple target architectures. These problems are discussed below. In the next section, tools that address these problems are introduced.

## Two Design Processes

The design processes for software and hardware can differ greatly. Often software and hardware development are done by different teams in different implementation languages with different design styles. When different teams are designing components for different domains, there is always the chance that the components do not work together properly. The teams need to work together and all the behaviors of the subsystems must be well documented, since interfacing software and hardware components is not always easy. Full system debugging

may not be possible. The use of different implementation languages further separates design work in both domains. Recently many research projects have created hardware description languages (HDLs) based on programming languages. These HDLs attempt to bring the hardware design process closer to the software design process, which might not be the best way to approach quality hardware design. Finally, the design teams use different design methodologies. We address the problem of managing two separate design processes by doing both the hardware and software design in Java.

## Design Partitioning

The next challenge is to determine how a hardware/software system must be partitioned across the two domains. Often designers try to minimize overall area or latency for the system. Recently, partitioning to minimize power consumption has become popular. How the design is allocated between the hardware and software domain has a first order effect on these attributes. Finding the optimal partition for a hardware/software system is difficult because it is dependent on several factors, such as interfaces, architectures and design constraints. When partitioning a system specification, designers might over-focus on minimizing area, latency or power. The smallest system might not meet timing requirements. Likewise, the fastest system might take too much space or power. A good partition is needed, otherwise system degradation is likely. Our variation of the codesign partitioning problem for FPGA-based systems is called the pipeline assignment problem. We use a realistic mathematical model of our target architecture and our pre-designed implementations of image processing algorithms so that we can find partitions that minimize the latency and meet area constraints using linear optimization techniques.

## Interfaces

A third challenge is deciding how to interface hardware and software. The designer needs to be aware of how data flows through the system with an emphasis

on when and where to move data. The channel and protocol need to be as fast and lightweight as possible. Communication bottlenecks cause system degradation, and frequent data movement should be avoided. We address this problem through the accurate modeling of our target architecture, so that pipeline assignments that minimize communication delays can be found.

### Target Architectures

The final challenge is choosing the correct target architecture for a design. There is no single architecture that is suitable for all codesign problems. The target architecture also affects how the codesign tool decides to partition a design or how to interface components. Our target architectures include both GPPs and FPGAs. FPGAs are useful for algorithms that exhibit massive fine-grained parallelism, which is the case for image processing. Systems that do not have much parallelism to exploit are not very good candidates for FPGAs. FPGAs also introduce several overhead costs, such as device initialization, reprogramming and communication. Not all of these costs are present in ASIC-based systems. In our related research, we found no other codesign tools for FPGAs that take all of these costs into account when doing co-synthesis. Without these costs the co-synthesis tasks have a greatly reduced complexity, and wildly inaccurate results.

## 1.2 Codesign Tools: Co-synthesis, Partitioning, and Interface Synthesis

Quality codesign tools can help designers build hardware/software systems that meet timing and area requirements. A codesign environment may include a number of tools for co-specification, co-synthesis, co-simulation, or co-verification. We are most interested in co-synthesis, which transforms a specification into a hardware/software implementation. There are several intractable subproblems to the co-synthesis process. Among the subproblems are:

- **Partitioning:** assigning parts of the specification to hardware and parts to software.
- **Scheduling:** deciding the timing of operations.
- **Resource Allocation:** assigning the hardware components to functional units.
- **Interface Synthesis:** designing communication channels between modules so that data can be moved through the system.

Of these co-synthesis subproblems, we consider partitioning and interface synthesis. Co-synthesis, partitioning, and interface synthesis are discussed in greater detail below.

### 1.2.1 Co-Synthesis Environments

In co-synthesis, the high-level description of a hardware/software system is translated into hardware and software components. The input specification can be a Control/Data Flow Graph (CDFG) or a system description in a co-specification language. There are several co-specification languages available, such as Streams-C [26, 34]. Many of these languages are derivative of either traditional programming languages (C/C++ or Java) or HDLs (VHDL or Verilog). The output for the software components is usually compiled code. The output for the hardware components can be either a HDL implementation of the components, synthesized components, or CDFGs. Many co-synthesis systems have one target architecture that they design systems for. Ideally, codesign tools would explore all possible target architectures to find the one best suited for the specified system. In reality, co-synthesis tools have to solve many intractable subproblems which are dependent on the target architecture, so fixing the target architecture eliminates some of the complexity in these tools. This dissertation has a target architecture of one CPU and one reconfigurable device.

There are many similarities between our research and co-synthesis tools. We provide an environment to translate pipeline specifications to hardware/software

executables. Our runtime environment assigns components in the pipeline specification to hardware and software implementations so overall latency is minimized and area is constrained to the device size. There are also tools to translate the assignments into an executable pipeline using interface synthesis.

Our environment has several significant differences to traditional co-synthesis tools. In particular, our environment was designed to solve several co-synthesis problems at runtime, while also providing an execution subsystem for the executable pipelines. Solving codesign problems at runtime is more difficult than at design-time because the processing time is severely constrained. Co-synthesis tools must solve several NP-Complete problems and optimal solutions at runtime may not be possible. There are also very few co-synthesis environments that target FPGA boards and ones that target FPGAs do not take advantage of runtime reconfiguration. Our environment uses runtime reconfigurability to put more components of a pipeline specification into hardware.

### 1.2.2 Codesign Partitioning Tools

In this dissertation, we solve the pipeline assignment problem (PA) as discussed in detail in Chapter 4. PA is a variation of the hardware/software codesign partitioning problem for reconfigurable systems. Codesign partitioning is a doubly constrained problem that minimizes latency and area requirements by assigning the co-specification to hardware and software domains. Some co-synthesis environments automatically partition the co-specification, but most system designers still manually partition their designs. There are several complications to the partitioning problem, such as interfaces, architectures and design constraints. Interfaces are important because communication costs are incurred when moving between hardware and software. Communication costs are often a large portion of the overall latency of hardware/software systems. The target architecture affects both the partition and the interface, which differ for each target architecture. Finally, design constraints must be considered when partitioning because the fastest designs might be too large for the given reconfigurable

device and the smallest designs might be too slow to meet timing requirements in the system specification. PA has many of these same problems as the above discussed problems with codesign partitioning.

PA has several key differences with the traditional codesign partitioning problem. These differences include using pre-compiled designs for software and hardware, and solving the problem at runtime. Most partitioning tools use a model to estimate a hardware/software system's latency and area from estimates of the hardware and software components in the partitioned design. In the pipeline assignment problem, we use a model that estimates a pipeline assignment's latency from measured hardware or software implementations' latencies as well as from measured overhead costs. The model accurately combines components as they sequentially execute so that all costs are accounted for. An accurate execution model and measured latencies should give more accurate system area and latency estimates. The other main difference between the PA and traditional partitioning models is how area is calculated. Normally, there is some metric for measuring the area of software implementations. In our system, area is measured by the amount of space the implementation requires on the hardware device, so software implementations take no area. Besides having no area, using software implementations allows a reprogramming cycle to occur, so that the next hardware implementation can use the entire hardware area. Another key difference is the amount of time allotted to finding a solution. Because partitioning in most systems is a design-time decision and not a runtime decision, partitioning systems can take several days to determine the solution. In runtime partitioning the computation is highly time constrained.

### 1.2.3 Interface Synthesis

Interface synthesis is the other subproblem of co-synthesis we are interested in. Interface synthesis determines how to interface two or more modules so that they can work in conjunction. Interface synthesis is very important in hardware/software systems, but is a problem that extends to pure software and pure

hardware systems as well. Interface synthesis can facilitate hierarchical design of complex systems using already designed components (e.g., intellectual property components) by helping designers find methods for handling data transfer between modules. Interface synthesis tools generate a channel to pass data between two or more modules, a protocol that determines the format the data takes, and a hardware or software machine to control the channel so that the data passes properly through the channel with the right timing. In cases where pre-designed components are used, the machine might also be responsible for repackaging data into different formats to resolve incompatible protocols. The generated interface between two components should have the minimal amount of glue logic to combine the components.

Our interface synthesis techniques have many similarities to traditional interface synthesis methods, such as finding methods to connect components so they execute in the correct order and pass the correct data. Our interface synthesis methods translate the pipeline assignment into an executable pipeline. This process involves creating a Java source file that executes all of the correct implementations and extra processing methods in the correct order. The input and output from the implementations and methods must also be connected properly so all of the data needed for each component arrives at the correct time and in the correct format.

Despite doing much of the same work as a traditional interface synthesis tool, our tool solves a highly constrained version of interface synthesis. In particular, all of the component implementations are designed before runtime. All of the component interfaces fall under one of several standard interfaces, which are used by the runtime environment to simplify both the model for PA and the interface synthesis work. Our tool also does not deal with intellectual property components. These constraints make interface synthesis easier to solve at runtime.

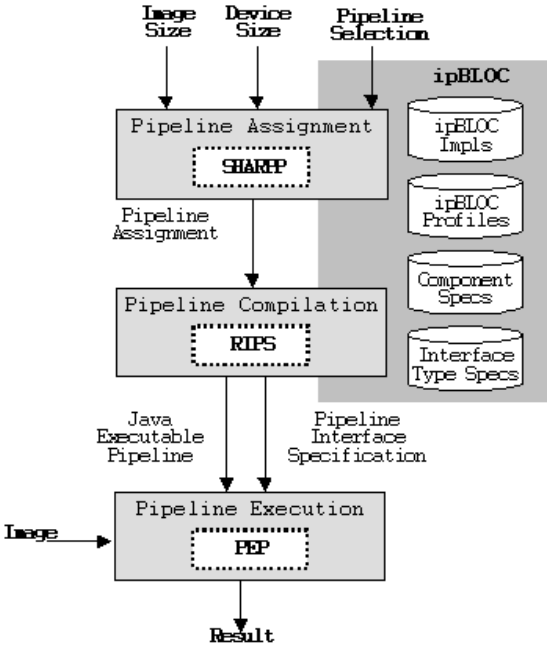


Figure 1.1: System Overview

## 1.3 Our Codesign Environment

Our codesign environment uses pre-defined components to build simple applications at runtime. This environment exploits the ability of FPGAs to reconfigure at runtime to build dynamic applications. Using these pre-designed components we can construct executable pipelines with runtime partitioning and interface synthesis on a specified series of components (*pipeline*). The user chooses and orders components from the image processing Basic Library of Components (ipBLOC) library to define a pipeline specification to process an image. Pre-defined software and hardware implementations of image processing components reside in the ipBLOC library. Our target architecture is one GPP and one reconfigurable board. The hardware implementations are designed for the Annapolis Microsystems Wildcard device that has a Xilinx XCV300E FPGA device and

4 Mb of RAM. In this section, our codesign environment is presented including the implementation language, the ipBLOC, and runtime codesign tools.

As stated earlier, managing the two design processes for hardware and software design is challenging. The best way to combat the challenges of two separate design processes is to bring the two processes together as much as possible. One way to reduce the challenge of coordinating two design processes is to unify the implementation languages, so that hardware and software designers share the same toolset. Using one language for the entire system also facilitates end-to-end debugging. Our codesign environment uses Java for designing both the hardware and software implementations, which allows us to design entire systems in the same Java Integrated Development Environment (IDE). The hardware description is done in Brigham Young University's JHDL [47, 6], a Java-based HDL for FPGAs. The runtime environment and the other codesign tools are also written in Java, so the entire runtime environment, including executable pipelines, can be debugged in one tool.

Part of the experimental work included designing the ipBLOC, which are hardware and software implementations of common image processing algorithms. The ipBLOC includes all of the necessary documents to use an implementation, as the runtime environment is document-driven. There are three document libraries in the ipBLOC: one for implementation information, one for component information, and one for interface information. The implementation documents include the hardware and software latency and area profiles for the given implementation. This library is used by the runtime environment in several ways. The Software/HArdware Runtime Procedural Partitioning (SHARPP) tool uses the implementation and component documents to estimate a pipeline assignment's area and latency characteristics. The Runtime Interfacing for Pipeline Synthesis (RIPS) tool uses the interface documents to connect component implementations to create executable pipelines. The ipBLOC library is described in Chapter 3.

The runtime environment coordinates user input, partitioning pipeline specifications, builds executable pipelines, and executes pipelines. The runtime environment was designed using component-oriented software engineering techniques, which allows all of the tasks to be separated from each other in modules. An abstract communication layer separates the interfaces between modules from the functionality of the modules so that any changes made to the interface do not change the entire system. In this way, pipelines are separated from the runtime environment, the components from the pipelines, and the implementations from the components, so that all three can be changed each time the runtime environment executes a pipeline. The Packet Exchange Platform (PEP) implements the abstract communication layer and is discussed in Chapter 5. Two important subsystems of the runtime environment, runtime partitioning and runtime interface synthesis, are discussed in the paragraphs below.

Our runtime environment includes tools for runtime partitioning of image processing pipelines. PA is a variation of the hardware/software codesign partitioning problem for reconfigurable devices. In this variant, latency is minimized and area is constrained to the size of the reconfigurable device. Since the device can be reprogrammed multiple times, there is no need to minimize the area. PA is NP-Complete, but exhibits the optimal substructure property which makes dynamic programming formulations possible. To accurately partition the components of a pipeline between hardware and software, modeling the interactions of components and their implementations when executed in series is required. A mathematical model was designed so that different pipeline assignments can be tested. The SHARPP tool uses this model to explore the problem space when trying to find a pipeline assignment. As part of our research, several algorithms for solving pipeline assignment through optimal and heuristic techniques were designed. These algorithms are used in the SHARPP tool. Since solving an NP-Complete problems becomes increasingly computationally intensive as the problem size grows, SHARPP uses a variety of these algorithms so that the time to solve PA does not impact system performance. PA is discussed in Chapter 4 and the SHARPP tool is discussed in Chapter 5.

The other important aspect of the runtime environment is that executable pipelines are built and run. The compilation process ensures that the pipeline assignment from the SHARPP is translated accurately into an executable Java source file. Plus, the executable pipeline is designed to make certain the user's data moves through the pipeline correctly. The RIPS tool is responsible for building the executable pipeline and is discussed in Chapter 5.

The tools are combined to create a testbench called Dynamo. Figure 1.1 shows the system diagram of Dynamo. There are three distinct phases in Dynamo execution: pipeline assignment, pipeline compilation, and pipeline execution. The only input the user needs to provide is an image, a pipeline specification and the name of the FPGA board in the target architecture. From the inputs, Dynamo extracts the image and device size that are used with the pipeline to solve PA in the SHARPP. The pipeline assignment is used by the RIPS tool to build the executable pipeline. The PEP attaches the executable pipeline to the execution subsystem so that the input image can be processed. The result of executing the pipeline is displayed to the user once processing finishes.

## 1.4 Contributions

The contributions of this research are:

1. The formulation of the pipeline assignment problem and the proofs of the NP-complete and optimal substructure properties.

2. The design of a mathematical model of how components and their implementations interact when executed in series on a target architecture of one FPGA and one GPP so that the pipeline assignment problem can accurately predict the latency and area characteristics for a pipeline assignment. In our related research, we found no other co-synthesis environment for FPGA-based systems that includes all of the overhead costs associated with reconfigurable technology.

3. The application of six algorithms to solve the pipeline assignment problem optimally and near-optimally. Algorithms were analyzed for two time limits over a range of pipeline and device sizes to determine which algorithm to use for a given pipeline size.

4. The design of an abstract communication layer that separates the executable pipeline from the runtime environment's execution subsystem to allow the pipeline to be easily changed at runtime.

5. The design of several software and FPGA hardware implementations of common image processing algorithms for the ipBLOC. Each implementation was profiled for a set of image sizes, so that a pipeline assignment's latency can be calculated from the implementations' runtimes.

6. The design of a runtime environment that coordinates user input, solves the pipeline assignment problem, builds executable pipeline and executes pipelines using component-oriented software engineering techniques.

## 1.5 Dissertation Structure

The related work is presented in Chapter 2. Background information about the model, the hardware and the implementations is covered in Chapter 3. Chapter 4 defines the pipeline assignment problem and introduces several methods used to solve the problem. Chapter 5 presents the Dynamo runtime environment. The dissertation concludes with discussion about future work in Chapter 6.

# Chapter 2

# Related Work

This chapter covers a variety of topics related to our research. Several projects from the general codesign field as well as the partitioning and interface synthesis subfields are discussed as many of those projects influenced our work. There are also several codesign environments for image processing that we studied. Since PA is closely related to other NP-Complete problems, such as graph coloring and parallel computer scheduling, projects focusing on combinatorial optimization solutions are also discussed.

## 2.1 Related Co-Synthesis Environments

There are many co-synthesis research projects which we studied. For brevity not all of them can be covered here. This section will cover a few of the most important projects: Ptolemy, Hybrid Architectures with a Single Transformable Executable (HASTE) and a system developed by Vahid. Tables 2.1 and 2.2 highlight several projects that are not discussed in detail. As discussed in the previous chapter, our work with codesign environments greatly differs from these projects. First of all, we allow the user to create simple applications using predefined components that have been implemented in software and in reconfigurable technology. This setup allows us to focus on building codesign tools for

runtime execution, as opposed to the more traditional design-time execution, which are highlighted in this section. Recently, two other dynamic codesign projects (HASTE and a project led by Frank Vahid) have cropped up. Both projects focus on transferring one software loop to hardware, whereas our project focuses on putting entire algorithms into hardware. Our project has potentially more speedup by focusing on functional level partitioning. Finally, in contrast with other codesign environments, ours specializes in reconfigurable technology. Designing for this type of hardware is very different than designing for ASICs, so specialized codesign environments are needed to fully exploit the advantages.

### 2.1.1 Ptolemy

Ptolemy [50, 51, 17] uses models of computation to do modeling, simulation and design of embedded systems, such as analog/digital systems. The partitioner uses a basic block granularity and optimizes over multiple hardware or software platforms. Ptolemy has an iterative partitioning algorithm that uses the concept of "Global Criticality/Local Phase" to balance resource consumption with time constraints. Their partitioning tool constrains time and minimizes area, which is the opposite of the pipeline assignment problem (PA) objectives. The objective for Ptolemy's partitioning algorithm adapts itself during execution to balance when to conserve latency and when to minimize area.

### 2.1.2 HASTE

The HASTE project [56] captures the instruction stream of executing object code to do dynamic partitioning so that the system designer only needs to create a software system. The project focuses on loops that run a fixed number of iterations. While the loop processes, the Hardware Compilation Unit (HCU) converts the software executable into a graphical representation of the loop's process. The graphical representation is synthesized into a hardware implementation for the reconfigurable fabric. Once the hardware implementation is ready, processing is transferred from the processor to the hardware device. When the

| Project | Target Architecture | Spec Language | Outputs | Partitioner | Notes |
|---|---|---|---|---|---|
| LYCOS [60, 54, 55] | CPU, ASIC | C or VHDL | Assembly and VHDL | dynamic programming | The PACE partitioning tool is discussed in the Partitioning section. |
| COSYMA [44, 43, 41] | SPARC processor, RAM, ASIP | $C^x$ | C and CDFGs | simulated annealing | The CDFG is synthesized by the Braunschweig Synthesis System to produce the hardware components. Their partitioner supports dynamically determined granularity |
| Vulcan [35] | CPU, ASICs | Output from Hercules Tool in *hardwareC* | synthesized software code; interface and hardware models in SIF | resource allocation and scheduling | The SIF models are synthesized in the Hebes/Ceres hardware synthesis environment |

Table 2.1: Related Co-Synthesis Environments, Part 1

| Project | Target Architecture | Spec Language | Outputs | Partitioner | Notes |
|---|---|---|---|---|---|
| Ptolemy [50, 51, 17] | embedded systems | multiple GUI front ends to describe system | C and VHDL | iterative | |
| HASTE [56] | CPU, reconfigurable fabric | executing software | synthesized hardware | none | Dynamic co-synthesis on loops |
| Vahid [76] [59] [77] | CPU, custom reconfigurable device | executing software | synthesized hardware | dynamic partitioning module | Dynamic co-synthesis on loops |

Table 2.2: Related Co-Synthesis Environments, Part 2

end of the loop is reached, processing is returned to the processor for completion. This project is very new and no performance results have been published.

### 2.1.3 Vahid

Frank Vahid [76, 59, 77] is leading a project similar to HASTE for dynamic partitioning of loops. In this system, the microprocessor is coupled with a dynamic partitioning module and a reconfigurable device. The dynamic partitioning module solves a very stripped down hardware/software partitioning problem in hardware. Since this project is focused on transferring a processing loop to hardware dynamically, the partitioning system has a more limited scope than traditional partitioners. Therefore, a hardware implementation of a partitioning tool is feasible. This project is significantly more advanced than HASTE and the published numbers give this project 1-5% speedup on the inner loops. While there are advantages to doing dynamic partitioning in this manner, 1-5% speedup is simply not enough. Our custom-made hardware modules have significantly more speedup, which is why custom hardware design is worthwhile.

### 2.1.4 Co-synthesis Summary

Co-synthesis is the main body of related research for this dissertation. These environments take system descriptions and translate them into hardware/software implementations. Our system combines already designed software and FPGA hardware implementations into an application. The functional level partitioning in our codesign environment gives vastly superior speedup than the two dynamic codesign environments presented here, which only accelerate loops.

## 2.2 Related Partitioning Systems

Partitioning assigns parts of the high level specification of a system to either hardware or software. Since this subproblem of co-synthesis environments is

very complicated, there are several projects that focus just on partitioning. Partitioning has been solved in many different ways from traditional methods, such as integer linear programming, to less traditional methods, such as fuzzy logic [8]. A variety of these projects are covered below; Tables 2.3 - 2.5 summarize several partitioning projects. These partitioning tools differ from our research in PA for many of the same reasons that our research differs from other general codesign environments. Our tool specifically partitions pipelines at runtime for reconfigurable devices.

### 2.2.1 Granularity

Partitioning algorithms differ depending on the unit of the code (*granularity*) the partitioning system works on. A partitioning tool with a fixed granularity will determine how best to partition all of the *grains* of a system specification. Many of the original partitioning systems worked at an instruction level, so each line of a system description potentially could be assigned to a different processing domain. With communication delays, that type of assignment is highly unlikely. Grains of sequentially executing code without control code (*Basic Blocks*) have become more common in recent years. Recent projects [42] have experimented with dynamically determined granularity. In this chapter, granularity is one of the characteristics that distinguish partitioning tools. In our approach, PA has a fixed granularity, since the components are neither divisible nor mergeable.

### 2.2.2 Niemann/Marwedel

Niemann/Marwedel from the University of Dortmund have done integer linear programming (ILP) [67] and mixed integer linear programming (MILP) [68] formulations of the partitioning problem. The designer specifies the system description, the target architecture and design constraints. The output is a VHDL specification of the hardware/software system. Their ILP formulation was the basis of ours, but theirs differs in how area requirements are constrained. Their formulation takes a standard approach to area estimation: both hardware

| Project | Target Architecture | Algorithm | Granularity | Inputs | Outputs | Notes |
|---|---|---|---|---|---|---|
| Niemann Marwedel [67, 68] | designer chooses | ILP/MILP | VHDL entities | VHDL, design constraints, target architecture | C, DFL, or VHDL | |
| SpecSyn [28, 27, 81, 80, 40, 45] | CPUs, custom hw processors, memories and buses | functional and greedy | system level | SpecCharts | VHDL | Partitioner supports designer interaction. |
| PACE [53] | CPU, ASIC | dynamic programming | basic block | C or VHDL | Assembly and VHDL | dynamic programming formulation based on knapsack formulation |
| AKKA [70, 48, 49] | RISC core, caches, and dedicated high-speed HW | dynamic programming | function/ loop level | C/C++ | object code and VHDL | VHDL is synthesized in their SYNT tool. Communication overhead is not accounted for when switching between hardware and software domains. Memory allocation, which has a first order effect on system performance, is optimized instead. Dynamic programming formulation based on knapsack formulation |

Table 2.3: Related Partitioning Tools, Part 1

| Project | Target Architecture | Algorithm | Granularity | Inputs | Outputs | Notes |
|---|---|---|---|---|---|---|
| ASSET [74] | CPUs, FPGAs | dynamic programming | function level | C, constraints | C, VHDL | dynamic programming formulation based on single processor scheduling |
| Codex-dp [9, 10] | CPUs, DSPs, ASICs and some programmable components | dynamic programming | course-grained | task graph | task graph | The mapping and partitioning phases occur simultaneously. The communication model supports communication concurrent to processing between modules, which is rarely modeled in coarse-grained partitioning models. |
| IMPACCT [57] | voltage scaling CPUs in a NOW with multi-speed communication interfaces | dynamic programming | function level | custom language | architectural templates | Partitions for energy consumption. The partitioner has three different dynamic programming formulations: optimal partitioning, optimal communication speed selection and a combined optimal partitioning and communication speed selection. |

Table 2.4: Related Partitioning Tools, Part 2

| Project | Target Architecture | Algorithm | Granularity | Inputs | Outputs | Notes |
|---|---|---|---|---|---|---|
| Eles [24] | real time systems | simulated annealing and tabu search | block, loops, subprograms, and processes | extended VHDL with message passing mechanism | synthesized hardware and compiled software components | Software and hardware components can execute concurrently. Algorithm based on graph partitioning. |
| Wiangtong et al. [82] | CPU, ASIC/ FPGA | genetic algorithm, simulated annealing, tabu search | functional | task graph | task graph | Uses a dynamic penalty, so that more "promising" regions can be explored |
| Hu Greenwald [46] | CPUs, ASICs | evolutionary | system level | not specified | not specified | The partitioner works with a multiple target objective function and uses imprecisely specified, multiple attribute utility theory (ISMAUT) to define the range that each target must be within in the optimal solution. A set of optimal solutions that meet the targets is likely and a ranking system determines which optimal solution is best. |
| Catania [8] | embedded systems | Fuzzy logic, genetic algorithms, nueral networks | functional modules | SpeedChart Extended Finite State Machine | C and VHDL | |

Table 2.5: Related Partitioning Tools, Part 3

and software components figure into the total. In our system, software has no area and reprogramming resets the area cost.

### 2.2.3 PACE

The PACE partitioner [53] is part of the LYCOS project [60]. Their dynamic programming formulation is based on the formulation for the knapsack problem and has a time complexity of $O(n^2 \cdot A)$, where $n$ is the number of code fragments being partitioned and $A$ is the total area of the hardware device. Their partitioner uses a basic block granularity built on the CDFG system model. Each basic block is annotated with the execution time and area requirements for both software and hardware, as well as the variables needed for input/output communication. The software execution time is estimated by translating the CDFG into simple programs. The programs' execution times are estimated using a "technology file" for a given processor. The hardware execution time is estimated by scheduling the operations, and the schedule is used to estimate latency. Hardware area also depends on the schedule generated for the execution time estimate. Finally, communication latency is estimated based on the time to do memory mapped I/O for each variable. All of these variables are used to estimate the system latency. PACE uses this model with its partitioning tool to find optimal partitions.

### 2.2.4 Wiangtong

Wiangtong et al. [82] developed a partitioning method using tabu search. Tabu searches "remember" frequently visited solutions using some sort of memory system in order to steer away from them. The simplest memory system maintains a list of the last $T$ solutions, so that the algorithm does not pick those solutions again. The Wiangtong project maintains both a tabu list and frequency statistics to track which regions are heavily searched. Frequently searched regions are penalized, forcing the algorithm into less explored areas. Wiangtong et al. explore a dynamic penalty, so that more "promising" regions can be explored.

Their project is more successful than standard tabu search. The PA tabu search algorithm maintains only a tabu list, but meets the requirement of hill-climbing out of local minima. In the future, we may try exploring the concepts of a penalty/reward system to leave heavily explored regions.

### 2.2.5 Partitioning Summary

Many of the projects discussed in this section provided direction for solving the pipeline assignment problem. There are, however, a few key differences. The area calculations in most partitioning tools usually include a metric for software components. Not only do software components not figure into the area calculation in the pipeline assignment problem, but the area estimate may be reset as the FPGA device can be reprogrammed on a software/hardware boundary. The final key difference is that many of the partitioning tools use algorithms too complex to implement for runtime use.

## 2.3 Interface Synthesis

Interface synthesis tools automatically create hardware or software methods to connect two or more modules so that data passes through the system correctly. Interface synthesis is an intractable problem and is applicable to pure hardware, pure software, and codesign systems. The runtime environment does interface synthesis at runtime on the pipeline assignment solution. We have simplified the interface synthesis problem as much as possible to build executable pipelines at runtime. This section covers two projects (PIG and LYCOS) that do design-time interface synthesis. Table 2.6 highlights several projects not covered in detail.

### 2.3.1 PIG

The PIG project [73] is a joint effort between Berkeley and Cadence, and does interface synthesis for intellectual property blocks with incompatible protocols. When modules have incompatible protocols the format of the data must be

| Project | Target Architecture | Inputs | Outputs | Notes |
|---|---|---|---|---|
| Chinook [12] | micro-controllers with external devices | behavioral description of the system, external device descriptions, and the microcontroller description | netlist, customized driver, and interface components | |
| Poseidon [36] | ASICs and programmable device | hardwareC | | Part of the Vulcan codesign environment. |
| PIG [73] | intellectual property building blocks | description of the protocol used by each module | Verilog | |
| LYCOS [61] | CPU, ASIC | C or VHDL | Assembly and VHDL | The term client/server is defined such that the client is the hardware component and the server is the software component. |

Table 2.6: Related Interface Synthesis Projects

repackaged with a different format while it passes through the data channel between modules. A Verilog-specified machine is designed to convert between the different signal conventions. The communication model between modules is similar to our abstract communication layer. There are abstracted producers and consumers of data which communicate through a synthesized interface. Unlike my research, the system designer has no control over a block's output format. Any two blocks may have different data formats, and the machine is responsible for translation between formats. In our research, any data format translation is done by the component.

### 2.3.2 LYCOS

The LYCOS project implements "one-sided" interface synthesis [61]. The hardware/software systems targeted in this project are applications where only the part that does not meet timing requirements in software is moved to hardware. Therefore, the hardware component is considered a client to the software system's server. The interface is synthesized from the server subsystem and the client must meet the interface requirements of the server interface. This project fits into an overall codesign environment that takes a system specification as input and returns the server implementation, channel implementation and the client specification as the output.

### 2.3.3 Interface Synthesis Summary

The projects described in this section have many related problems to our work into abstract communication layers and runtime interface synthesis. The interface synthesis problem in the general form is very complex to solve, and our research greatly simplifies the problem so that runtime solutions are possible.

## 2.4 Image Processing Systems

Besides general codesign environments, there are many image processing specific codesign environments. Since codesign involves solving many intractable problems, restricting the range of implementable applications, like restricting the target architecture, allows researchers to design good quality application-specific tools instead of poorer quality general tools. There are many research and commercial tools for building image processing hardware from a variety of hardware description languages (HDLs). This section covers three projects: Cameron, MATCH, and CHAMPION.

### 2.4.1 Cameron

Colorado State University's Cameron project [39] created a C-variant HDL, called Sassy (Single-Assignment C), for building image processing systems. Sassy compiles into VHDL code. The Sassy programming language [38] is considered "closely related" to other C-based parallel languages (C\\and CT++). This project attempts to create codesign system designers from software programmers, which is a dubious goal. Programming in a single assignment C-variant language is not the same as programming in C. Pointers and recursion are not allowed in a single assignment language, so the designer must adapt their algorithms to work in the single assignment structure. The programmer must also specify the bit width of each variable. Efficient FPGA designs often use the minimum bit width possible, which the designer is often responsible for determining. Bit width analysis is not a typical skill of programmers. Besides the difference in programming, there are differences in how to implement algorithms in hardware versus software. Efficient hardware implementation of an algorithm might differ greatly from the software implementation and compilation tools may not find the best hardware implementation using a software implementation. For example, our most efficient median filter hardware implementation uses a highly optimized nine element sorting network and the most efficient software implementation uses bubblesort.

### 2.4.2 MATCH

The Northwestern University Electrical Engineering department has created MATCH (Matlab Compiler for distributed Heterogeneous computing systems) [5] that compiles Matlab code to RTL VHDL and C/MPI. The first part of the project works with native Matlab functions. These functions were translated to RTL VHDL to create a library of components. MATCH can also transfer user-defined functions into RTL VHDL. Once the VHDL code is created, the circuit can be automatically mapped to hardware using the SYMPHONY mapping tool developed at Northwestern. This project is very promising and their results are out-performing computation on the host machines and embedded DSP boards. This project focuses on creating image processing FPGA-based designs, whereas our research uses both software and FPGA technology for image processing.

### 2.4.3 CHAMPION

The CHAMPION project [69] also tries to optimize image processing pipelines. CHAMPION spreads computation across a multiple FPGA environment, whereas our research spreads computation across FPGAs and software. Since our project addresses runtime execution while CHAMPION is a design time tool, the two projects are complementary.

### 2.4.4 Image Processing System Summary

The systems described in this section help designers create image processing systems. Most of these projects are focused on building hardware implementations of image processing algorithms. Our codesign environment assumes this phase of the project is already completed and that the hardware implementations already reside in a library for use. Thus, much of this work is complementary to our own.

## 2.5 Graph Coloring

The graph coloring problem assigns colors to the vertices of an undirected graph such that the end points for each edge in the graph do not have the same color. Since the problem is NP-Complete, optimal solutions exist only for small instances. Graph coloring has many practical applications, such as scheduling, so there are many heuristics and metaheuristics for quick, near-optimal solutions. Many heuristics for solving graph coloring, especially Generic Tabu Search (GTS), guided my work with local search. This section discusses GTS and summarizes several other graph coloring algorithms.

### 2.5.1 GTS

The GTS algorithm developed by Jin-Kao Hao [18] explores the problem space by accepting solutions with conflicting colors (i.e., connected vertices assigned to the same color). Given a $k$-color solution, GTS tries to eliminate the color $k$ by assigning all $k$-colored vertices to a new color in the range $[0, k-1]$ such that the number of conflicts is minimized. The algorithm attempts to resolve the conflicts using local search. If a proper coloring is found, the algorithm attempts to eliminate the color $k-1$. If a proper coloring is not found, the algorithm halts with the $k$ coloring solution.

We modify GTS for the pipeline assignment problem. Since not all implementations of a component work with all implementations of other components in the ipBLOC library, assigning a component to an implementation that combines with only a few implementations might cause conflicts in the solution. Therefore, many of the heuristics only assign components to implementations that work with all of the other possible implementations, which restricts the problem space. Using conflict resolution techniques, components are assigned to new implementations without concern of conflicting with the adjacent components' implementations. If the conflicts can be resolved, the new assignment is kept. Conflict resolution allows local search to explore areas of the problem space inaccessible by other heuristics.

| Project | Algorithm | Notes |
|---|---|---|
| Generic tabu search [18] | tabu search | First graph coloring algorithm to use conflict resolution |
| Hybrid Coloring Algorithm [30, 19] | hybrid metaheuristic of tabu search and genetic algorithms | Considered the best graph coloring solver |
| Scatter search [37] | population-based method related to tabu search | This algorithm is meant to be more deterministic than Hao's research with hybrid evolutionary algorithms, but still uses some of the work from his other projects, such as the GPX crossover operator |
| Variable Neighborhood Search [4] | local search technique that uses variable sized neighborhoods | |
| Iterated Local Search [11, 72] | local search | |
| Evolutionary Annealing [25] | hybrid genetic algorithm with simulated annealing | The simulated annealing parameters are tuned to control whether the algorithm is making aggressive moves to leave a local minima or fine tuning an already good solution. |

Table 2.7: Related Graph Coloring Projects

### 2.5.2 Miscellaneous Graph Coloring Projects

There are many variations on the local search algorithm for graph coloring. Some of these variations are standard local search algorithms, but there are also many projects that combine local search with other heuristic techniques. Two projects are the the Variable Neighborhood Size (VNS) and Iterated Local Search (ILS) projects. Avanthay et al. [4] developed a standard local search technique that uses variable sized neighborhoods. VNS uses several different types of neighborhoods in hopes that at least one neighborhood in the set would avoid local minima. Eleven different neighborhoods were designed, and their algorithm uses the six best neighborhoods. Their method of searching the neighborhood randomly picks a few solutions to try, unlike our method which methodically searches the entire neighborhood. The ILS algorithm [11, 72] attempts to do random walks around local minima to jar the solution out of the minima. Once the solution is out of the local minima, local search is applied again until the solution is stuck in another local minima. This method for perturbing is not as aggressive as a random restart. ILS performs better than Hybrid Color Algorithm (HCA), which is widely considered the best graph coloring algorithm for structured graphs, but not random graphs. These techniques might be interesting to modify for our local search algorithm in cases when the local search algorithm gets stuck in a local minima before the time limit expires.

There are many local search hybrids, such as HCA, Evolutionary Annealing (EVA) and Scatter search. Many of these algorithms combine genetic algorithms and local search. HCA and EVA evolve a population of solutions. Each solution in a population is improved using local search. Then superior solutions from the population are picked to create the solutions for the next generation of the population. Child solutions are made from combining two or more parent solutions using crossover or mutation algorithms. Genetic algorithms may not be well suited to PA on many different levels. Distinguishing what makes a good parent PA solution is not clear. Combining two parent solutions may create a highly conflicted child solution. The conflict resolution process on the child may

eliminate the parents' good characteristics. Finally, these hybrid algorithms expect to run the local search algorithm many times on a large population of solutions. With our time constraints, there is barely enough time to perform local search on one solution of a pipeline, so maintaining a large population would not be practical.

### 2.5.3 Graph Coloring Summary

There are many interesting heuristics and metaheuristics that have been developed for graph coloring. The techniques employed by GTS for allowing and resolving conflict have greatly influenced the local search techniques for PA. Other techniques may be useful in cases where the algorithm gets stuck in a local minimum before the time limit expires.

## 2.6 Partitioning and Scheduling for Parallel Computing

Parallel computing research addresses many of the same problems as hardware/software codesign environments. Both translate the specification of a system into an "appropriate" implementation. In parallel computing, the appropriate implementation is a parallel version of the sequential program. In McCreary [63], the process of transformation from a sequential to parallel program is stated as "four non-trivial problems:

- the identification, though program dependence analysis, of code fragments which can be executed in parallel;
- the elimination of non-essential dependences to maximize the number of parallelizable code fragments;
- the aggregation of such fragments into processes that optimize overall performance (in terms of both communication and processor time);

- the assignment of processes to processors (scheduling)."

This process is the same as codesign partitioning. As with codesign, these problems are NP-Complete. McCreary [63] states that there are three main methods for doing scheduling: critical path heuristics, list scheduling heuristics and graph decomposition. In our research we found several other methods in use: local search, greedy, and dynamic programming. In this section, I cover two relevant problems: Clans and Management System for Heterogeneous Networks (MSHN). Table 2.8 summarizes several projects not discussed in detail. Before these projects are discussed, the similarities and differences between our research and parallel computing are presented.

PA has many similarities to the scheduling problem in parallel computing. Unlike codesign environments, parallel computing can decide the partition and schedule statically (before runtime) or dynamically (at runtime). PA is most similar to static scheduling, but much can be learned by also looking at runtime scheduling algorithms. In the scheduling problem [23] a set of tasks need to be distributed across several machines. The target architecture for the pipeline assignment problem (one machine, one FPGA) can be seen as a tightly coupled heterogeneous network of workstations, and the pipeline can be seen as a very simple application with a very rigid schedule.

There are also several differences between PA and parallel computing scheduling. Granularity, as discussed below, is handled dynamically in parallel computing scheduling, whereas it is fixed in PA. Parallel computing scheduling has much more simplified coupling costs than in PA, as only communication costs would need to be taken into account. Without hardware initialization, reprogramming and extra processing to account for, the problem is greatly simplified, though still NP-Complete.

### 2.6.1 Granularity

In parallel computing granularity is defined differently than in codesign partitioning. In Khan et al. [52] granularity is defined as the "measure of the

| Project | Algorithm | Notes |
|---|---|---|
| DSC [85, 84, 32] | list scheduling | This project is a precursor to many other scheduling heuristics. DSC finds the optimal clustering of tasks over a set of processors. |
| Graph Decomposition (Clans) [52, 63, 64] | clan-based graph decomposition | |
| TASK [83] | local search | Target architecture is a message passing system |
| Porto/Ribeiro [15, 16] | tabu search | The initial solution is determined through a greedy algorithm called DES+MFT [65, 14, 66]. The algorithm maintains a matrix of the last $n$ moves, where each solution is a vector represented in the matrix. |
| Management System for Heterogeneous Networks [2, 33, 7] | greedy, dynamic programming, and mixed-integer programming | |

Table 2.8: Related Parallel Computing Projects

ratio of execution time to communication time." Khan also points out that when granularity is low, communication costs outweigh the usefulness of parallelization. Conversely, when granularity is high, communication costs are trivial compared to execution time. Another difference is that in codesign partitioning the granularity is usually fixed. In contrast, McCreary [63] states that the role of the partitioner in automatic parallelization of serial code is to find the optimal granularity size.

### 2.6.2 Graph Decomposition (Clans)

Scheduling and partitioning can be approached through clan-based graph decomposition (CGD) [52, 63, 64], a technique which builds on 2-structure theory [22]. The first step in this method is to convert the program to a CDFG called a Program Dependence Graph (PDG). The PDG is a Directed Acyclic Graph (DAG) or Directed Acyclic Hypergraph (DAH). Under this context, CGD is defined as "a parse of the PDG into a hierarchy of subgraphs."

Clans can be independent, linear or primitive. The clan structure of a PDG shows the areas of the program that must execute sequentially (linear clans) or can execute in parallel (independent clans). The clan structure is also independent of a target architecture. Once the cost metric is added to the PDG then partitioning for a target architecture can occur. When a partition and schedule of a PDG cannot be found that is faster than the sequential program, then the sequential implementation is used. In other systems, the parallel implementation of a program might be significantly slower than the sequential version. In the future, I plan to investigate more complex pipelines that include control structure. Using graph decomposition to cluster the clans of the pipeline may be useful.

### 2.6.3 MSHN

Viktor Prasanna's group at the University of Southern California has been working on a resource allocation project in heterogeneous computing (HC) [2, 33, 7]

called the MSHN. This project schedules heterogeneous network of workstations (NOWs) with real-time components (e.g., sensors). MSHN does two-level scheduling for applications. An initial static schedule is determined for spreading the tasks across the HC. As the load on the system changes, the environment monitors the changes and dynamically reschedules tasks to do load balancing. This approach requires a good initial solution to the scheduling of the application that can then be altered to meet the demands of increased data. They have solved this problem using many different approaches including greedy, dynamic programming, and mixed-integer programming. This project is different from PA because in PA the pipelines are expected to change frequently but knowledge of previous pipeline assignments is not always useful when solving PA repeatedly.

### 2.6.4 Parallel Computing Summary

When developing algorithms for solving PA, parallel computing algorithms guided our research. In the future, PA will be extended to include partitioning across a network of workstations. Many of the algorithms presented here can be modified to solve PA for the NOW environment.

## 2.7 Conclusions

In this chapter, related research has been presented. Our research is related to the fields of codesign environments, partitioning tools, and interface synthesis tools. Our work with combinatorial optimization algorithms to solve PA was guided by work on other NP-Complete projects. Several projects from all of these fields were examined to show how these other projects influenced our own research. PA differs from previous research in that FPGA-based systems are specifically targeted. FPGA systems have different overhead costs than other hardware devices, and codesign environments do not account for these costs. When the FPGA-specific overhead costs are not accounted for the system latency is usually underestimated. In the next chapter, the models, hardware

and libraries for the pipeline assignment model are introduced.

# Chapter 3

# Models for Pipeline Assignment

In this section, we introduce basic information about the building blocks of our research. First, we introduce our model for how components are combined into executable pipelines for our target architecture of one Field Programmable Gate Array (FPGA) board and one CPU. Then we detail the specifics of our hardware setup. Finally, we present the image processing Basic Library of Components (ipBLOC) and describe how implementations fit into the model.

## 3.1 Pipeline Assignment Modeling

In our system, the image analyst chooses and orders components from the ipBLOC to form a pipeline to be executed on an image (or images). There are many hardware and software implementations for each component in the ipBLOC. Each of these implementations has its own latency and area properties. We are interested in finding a way to assign the components of a pipeline to implementations (*pipeline assignment*) so the overall latency is minimized and the area is constrained to the FPGA size.

Assigning components to implementations is not a simple task, as not all assignments can be translated into real, executable pipelines. In these cases, invalid pipeline assignments either will not work correctly with the FPGA board

(i.e., the bitstream does not fit on the device, the pipeline assignment does not include communicating the image to the board, etc.) or illegally combine implementations. To automate pipeline assignment, a model of how the hardware board can be used and how components can be combined has been developed.

We apply linear optimization techniques to the model to find a pipeline assignment that meets our objectives. This task is not straightforward either. Besides the latency from the implementations, the model must account for all of the overhead latency caused by using the hardware device and extra processing from combining implementations in the series. The hardware device itself adds three types of overhead costs: powering on the device (*hardware initialization*), programming the device with the hardware implementation (*reprogramming*), and moving data on or off the board (*communication*). The extra processing methods are needed to connect components in series and their latency must also be modeled. In our related research we found no FPGA-based codesign environments that model all of these costs. Removing the overhead reduces the model's complexity and the model's accuracy. Therefore, we expect our results to more accurately reflect the overall system latency. This section provides an overview of our model; Chapter 4 details the formulation of this model.

There are many area and latency considerations when assigning the components of a pipeline to implementations. The FPGA board must be accurately modeled so the assignments to hardware reflect how reconfigurable hardware can be realistically used. Specifically, the FPGA board is finite, so our model must reject pipeline assignments that do not adhere to the area constraint. Combining components into pipelines is the other important task of that needs to be modeled. Besides the assigned implementations' latencies, there are other overhead costs associated with executing components in series. The FPGA board incurs overhead costs, and there are also component-specific overhead costs. Not all implementations can be combined with all other implementations, so the model must reject bad combinations. All of these costs and constraints must be taken into account when assigning components to hardware and software implementations. In the following paragraphs we examine these concepts in greater detail.

First, there are area constraints for the hardware. Area for the assignments are constrained to the size of the FPGA. Hardware implementations have a positive non-zero integer area property; software implementations have zero area. Many times there will be a contiguous set of components in the pipeline assigned to hardware; we call these *hardware subsequences*. A hardware subsequence can be as small as one component and as large as the entire pipeline. When there are multiple components in a hardware subsequence, the individual bitstreams are merged into one bitstream so that they can be executed in series without reprogramming. Hardware subsequences have an area equal to the sum of all of the individual areas for the implementations in the subsequence. Hardware subsequences cannot exceed the size of the FPGA. We assume the device is reprogrammed on all software/hardware boundaries so each hardware subsequence can use the entire device space.

Next, there are latency costs when combining components into executable pipelines and additional costs when using hardware. We assume we can estimate the overall latency of a pipeline assignment by summing the latencies of all of the individual latencies for the implementations and overhead costs. All of the implementations in the ipBLOC are profiled before the image analyst uses them, so the average runtime for each implementation over a set of image sizes is known. Linear interpolation is used to determine the latencies for image sizes not profiled. Another source of latency is the cost of combining two components the *coupling cost*. The coupling cost is related either to using the FPGA board or combining components.

First, we present the overhead costs associated with using the FPGA board. If assignments are made to hardware implementations, then a one time hardware initialization cost is paid to turn on the device. At the beginning of a hardware subsequence, the input data is written to the hardware board's memory and the device is programmed with the merged bitstream. At the end of a hardware subsequence, the output data is read from the hardware board's memory. We assume that within a hardware subsequence there is no data communication with the host PC. All of these costs have a significant impact on the overall latency

of an executable pipeline. The hardware initialization cost for our hardware board is so large that assigning components to hardware implementations does not happen until there are enough components in the pipeline to amortize the cost over several hardware implementations.

We also need to model how implementations can be combined to execute in series. Extra processing methods are needed to combine components together to properly condition the image data during processing. The extra processing costs are a function of both the implementation and the component. There are two types of extra processing costs that we model: image padding and packing. Image padding allows the hardware to process the image boundaries during window-based processing. Image packing allows some of the components to pack multiple pixels into one data word to decrease the communication cost of removing the image from the board's memory. Both are described in Section 3.3. Extra processing methods are an overhead cost that is included in the model.

In summary, the model of pipeline assignment formulates the rules for how FPGA hardware can be used with software so that only valid pipeline assignments are implemented. The model also quantifies the properties of valid pipeline assignments so that they can be compared accurately. In Chapter 4 this model is transformed into a mathematical formalization that is used with combinatorial optimization techniques to find optimal or near-optimal pipeline assignments. In the next few sections, we discuss the details of the FPGA board we use and the components implemented in the ipBLOC.

## 3.2 Wildcard

All of the hardware implementations in this dissertation were designed for the Annapolis Micro Systems Wildcard board. A block diagram of the board is shown in Figure 3.1. The Wildcard has one FPGA device, a Xilinx Virtex XCV300E chip and two banks of two Mb SRAM. The SRAM is single port memory, but each bank is connected to the FPGA through separate buses. This means that both banks can be accessed simultaneously for one memory

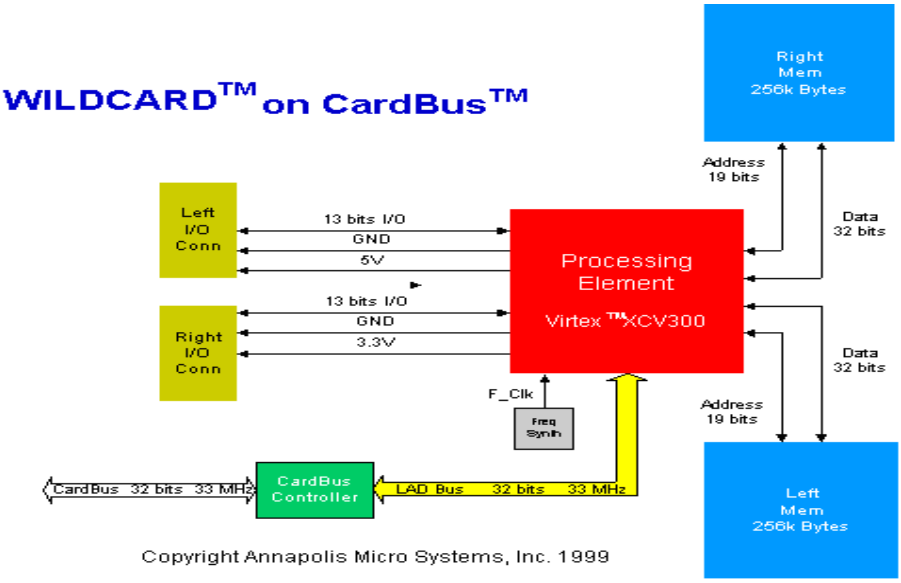


Figure 3.1: Block Diagram of the Wildcard

transaction per clock cycle. The Wildcard board is a type II PCMCIA Card and is connected to a machine through the 32-bit CardBus interface. This section details how our designs use the Wildcard's memory architecture as well as how the specific Wildcard hardware costs are modeled.

The memory architecture and interfaces are an important concern in our designs. In our hardware designs the bottleneck is the memory bus since we can only read or write one 32-bit pixel at a time to each memory bank. Since our components need to store both the input and output data in memory at the same time, one memory bank stores the input while the other is used for output. This allows us to be able to read input and write output in the same clock cycle.

The latency calculation for a pipeline assignment that assigns some components to hardware includes hardware initialization, reprogramming and communication costs. We measured all of these costs in the Eclipsecolorer profiling tool [21] for the Eclipse [20] Java integrated development environment (IDE). The hardware initialization cost for the Wildcard is a fixed value of 2365 ms.

The reprogramming cost for the Wildcard is negligible, but a larger device may have more significant reprogramming costs. The communication costs for the FPGA board are much more complicated than the hardware initialization and reprogramming costs, and are discussed in the next few paragraphs.

Communication costs pertain to moving data between hardware and software. This cost can make up to 90% of the cost of executing a hardware implementation. There are two classes of data transactions: memory and register. Hardware subsequences start with writing the image data to memory, and end with reading the result (image or histogram) from memory. Most of the hardware implementations also have several user controlled registers for supplying data about the image, such as the image's dimensions. Hardware implementations also use start and done registers on the FPGA to do handshaking. A software process monitors the done register so that the next task in the executable pipeline can be started.

In the profiling tool, the register transactions have negligible execution time, unlike the memory transactions. The execution times for memory transactions are a function of the amount of data being transferred and the hardware implementation's clock speed. How much data is transferred depends on the implementation. The windowing components may transfer a raw image or a padded image (which is larger than the raw image) for input and output. A histogram implementation requires that an image be transcribed to the device, and histogram data be returned. The size of the histogram data is proportional to the number of channels and bins for output.

## 3.3 ipBLOC

Pipelines are constructed from components in the ipBLOC. The components represent basic image processing algorithms that have been implemented in both software and FPGA hardware. Currently supported components are listed in Table 3.1. There are two edge map and eight histogram components that represent completely different algorithms. The two edge map components take the

| ipBLOC Component Name |
|---|
| Median Filter |
| Minimum Filter |
| Maximum Filter |
| Sobel Edge Detector |
| Edge Map: And |
| Edge Map: Or |
| Histogram: 4 channels (TRGB), all 8 bits per channel, 256 bins |
| Histogram: 4 channels (TRGB), all 8 bits per channel, 16 bins |
| Histogram: 4 channels (TRGB), upper 4 bits per channel, 16 bins |
| Histogram: 3 channels (RGB), all 8 bits per channel, 256 bins |
| Histogram: 3 channels (RGB), all 8 bits per channel, 16 bins |
| Histogram: 3 channels (RGB), upper 4 bits per channel, 16 bins |
| Histogram: 1 channel, upper 3 bits for RGB channels merged, 256 bins |
| Histogram: 1 channel, upper 3 bits for RGB channels merged, 16 bins |

Table 3.1: ipBLOC Components

threshold of the Red, Green, and Blue color channels separately and then combine the results into a one bit result. The different approaches to the edge map component are two different methods for combining the threshold results for the three channels: one *and*s and the other *or*s. The histogram components represent eight ways to define which channels to use, how many bits of a channel to use and how many bins per channel.

All of the implementations were profiled for latency by using the Eclipsecolorer profiling tool for the Eclipse Java IDE. For each implementation, multiple images were profiled. All of the data for an image size for an implementation is averaged to estimate the latency of the implementation for that given image size. This averaged data is stored in the implementation specification documents in the ipBLOC. In cases where the runtime environment is processing an image with an image size not covered in the profiles, linear interpolation is used to determine the latency.

Each component has a component specification file, which is stored in the

component specification library as part of the ipBLOC. The component specifications define these values:

- Component name (from Table 3.1)
- Command line arguments with types to execute the component
- Interface type (from Table 3.2)
- Bitstream information: name of the bitstream, input/output memory location (right or left) for input/output data, area in slices
- Profiling data: all measured values for the component's hardware and software implementations

### 3.3.1 Image Padding

The median filter, maximum filter, minimum filter, and edge detection components all do window-based computations. Window-based processing uses the neighbors of a given pixel to process the pixel. Currently, all window-based components use $3 \times 3$ windows. Pixels on the boundary of an image do not have all the neighboring pixels needed to form a window. There are many techniques for determining neighbors for pixels at image boundaries [58]. Our components use the value of the nearest neighboring pixel inside the image to determine the value of the pixel outside the image.

The software and hardware implementations create windows for each pixel being processed. In software, the boundary cases can easily be determined on the fly. Hardware works fastest without special cases, so a one pixel-wide border is added to the image before processing. This process is called *image padding*. There are three methods to support image padding: pad, unpad and fix. Before using a windowing hardware implementation, image padding is added to the image. If the windowing hardware implementation is followed by another windowing hardware implementation, the image padding is fixed between implementations. The image padding needs to be fixed, because the values of

the image boundary pixels might have changed during processing. There are many different ways to fix the image padding, as discussed in Chapter 4. If a windowing implementation is followed by either a software implementation or a non-windowing hardware implementation, the image padding is removed between computations.

### 3.3.2 Image Packing

When a pixel is edge mapped, the result is one bit wide, which easily allows for image packing. Edge mapped images are packed so that 32 pixels are represented with one data word. In cases where a hardware implementation of an edge map component is followed by a software implementation, packing the image greatly reduces the communication cost of transferring the image back to the host processor. While the hardware/software communication cost is reduced, the image needs to be unpacked before it is used by the software implementation. In cases where the hardware implementation of an edge map algorithm is followed by another hardware implementation the image is never packed.

### 3.3.3 Bitstream Merging

Bitstream merging allows two or more synthesized hardware implementations (*bitstreams*) to be combined together into one implementation. Bitstream merging allows us to create many small hardware implementations and then combine them together after they have been synthesized. This process allows us to put multiple hardware implementations onto an FPGA device at one time without reprogramming.

This operation is currently not implemented in our runtime environment, but will be added in the future. In the meantime, we created multiple hardware implementations of components some with extra processing methods. Median filter, maximum filter, minimum filter, and edge detection need the image padding methods of pad, unpad, and fix. The two edge map components need the image packing methods of pack and unpack. All of the image padding and

packing methods are small and easily implemented in hardware. Therefore, both of the edge map components have hardware implementations with and without the packing method. Similarly, the windowing components have six implementations: no padding methods; pad only; unpad only; fix only; pad then process then unpad; and pad then process then fix. Not all combinations are realizable for the particular device we are using. For example, the median filter hardware implementation is significantly larger than the other three windowing components, so cannot implement the last two variations with processing as well as two image padding methods. The algorithms for solving the pipeline assignment problem treat these different variations as separate hardware implementations.

## 3.4 Interfaces

Just as components are a generalization of one algorithm's implementations, several components can be generalized with one interface. Many components have the same input/output data and do a similar type of processing. Any components that share the same input/output data and processing type are represented by one interface type. For example, median filter and edge detector are two types of window-based image processing algorithms that take the image and the image dimensions as input and return an image as output, so they can be generalized under a window interface. Interfaces greatly simplify the mathematical model, since there are fewer rules for how to combine components serially.

Each ipBLOC component is classified by *interface type*. Components with the same interface type have similar coupling costs. The interface types for the components are shown in Table 3.2. Interface types are classified by processing type, input variables, and output variables. Each interface type has a corresponding specification which is stored in the ipBLOC. The interface type specification stores the common data for all components with that interface type. The interface type specifications define these values:

| Interface Types | ipBLOC Component Names | Processing Type | Input Interface | Output Interface |
|---|---|---|---|---|
| Threshold | Edge Map And, Edge Map Or | per pixel | image, width, height, threshold | image |
| Windowing | Median Filter, Minimum Filter, Maximum Filter, Sobel Edge Detector | per window | image, width, height | image |
| Histogram | Histogram: all eight | per pixel | image, width, height | one array of histogram data per channel |
| Simple | Source, Sink | per pixel | image | image |

Table 3.2: Interface Types for ipBLOC Components

- Interface type name: from Table 3.2
- Input interface data: defines input data (and their types) for a component of this interface type
- Output data: defines output data (and their types) that is produced by a component of this interface type
- Display type: output image or histogram bargraph

## 3.5 Coupling Cost

Central to the model is determining the processing needed to execute components in series. The *coupling cost* represents these overhead costs in the model. These costs are communication, extra processing and reprogramming costs. The coupling cost varies both by the implementation types (hardware or software) and by the interface types of the two components and are outlined in Table 3.3.

| i | j | Coupling Cost for Pair (i,j) |
|---|---|---|
| SW | SW | 0 |
| SW | HW | Communication, reprogramming, extra processing |
| HW | SW | Communication, extra processing |
| HW | HW | Extra processing |

Table 3.3: Coupling Costs for Each Boundary Type

The implementation types of the two components determine whether communication costs are needed; they also determine whether reprogramming costs are needed. For example, a software/hardware transition requires the input data to be communicated to the board and the FPGA on the board to be programmed with the bitstream, but a software/software transition requires neither of these costs. The extra processing costs, as discussed in this chapter, are a function of both the implementation and the interface. For example, hardware implementations of window components have image padding that needs to be added before execution and either removed or adjusted after execution contingent on the following components' implementations and interface types. In Chapter 4 the coupling costs are defined for all of the possible combinations of two interface types and two implementation types.

## 3.6 Conclusions

In summary, this chapter presented the assumptions, hardware and libraries used to create the models for pipeline assignment. The Wildcard board is modeled so pipeline assignments that work with the device are found and the overhead cost of using the board are accounted for. In the next chapter we take these concepts and formalize them into a model so linear optimization methods can be used to find solutions that meet our objectives.

# Chapter 4

# The Pipeline Assignment Problem

## 4.1 Introduction

A *pipeline assignment*, $K_N$, maps the components of a pipeline $P_N$ to hardware and software implementations. There are two primary characteristics of a pipeline assignment: latency and area. The latency of a pipeline assignment is the total execution time of all of the implementations in the pipeline assignment plus all of the overhead costs. The area of the pipeline assignment is determined by sequentially executing hardware implementations (*hardware subsequences*). The area of a pipeline assignment is the largest hardware subsequence area. The pipeline assignment problem (PA) is the optimization problem of finding a pipeline assignment $K_N$ for a pipeline $P_N$ and image size $z$ with the minimum latency, such that the area for $K_N$ is less than or equal to the area of the hardware device. A *valid* pipeline assignment (or valid solution) has an area that meets the area requirements of the FPGA. An *optimal* pipeline assignment (or an optimal solution) is a valid solution with minimal latency. PA is a variation of the hardware/software codesign partitioning problem formulated for FPGAs. Both problems share many similarities, but have a few key differences. The

comparison is presented below.

First, we present the similarities. Both PA and the codesign partitioning problem are NP-Complete, which makes them difficult to solve as the problem size grows. Many codesign partitioning algorithms can be modified to solve PA, which allowed us to use decades worth of combinatorial optimization work on the codesign partitioning problem when we designed our algorithms. Both PA and hardware/software codesign partitioning have the same objective: to balance the use of hardware and software for better overall performance in embedded systems. Partitioning a design specification, as with assigning components, has a first order effect on system performance. Moving complex calculations from software to hardware can make the system orders of magnitudes faster as long as the overhead costs are not significant. Finally, both problems are affected by the interplay of area and latency. If the design is optimized for one attribute without regard for the other, deleterious effects on performance can happen, such as a system that is either too slow or too large.

The pipeline assignment problem differs from the partitioning problem on several key points. PA was designed to be solved at runtime and to target FPGAs. We will first discuss solving PA at runtime. PA is solved in a special runtime environment after the pipeline is specified and before the image is processed. In order to not dominate processing time, PA is solved within a set time limit which forced us to use techniques to solve the problem quickly. This approach assumes the components' hardware and software implementations already exist in a library so they can be combined at runtime to create executable hardware/software pipelines. Since all implementations are pre-built, the latency and area profiles are determined before PA is solved, which leads to a more accurate assessment of latency. Traditionally, partitioning is determined at design time using estimated latency and area values. Since the pipeline assignment is determined at runtime, the user may change the image, pipeline, and FPGA device each time PA is solved. Giving the user the ability to change the pipeline at runtime allows the user to experiment with several pipelines for

one image. In contrast, traditional codesign partitioning is done before the components are designed and does not respond quickly to changes in the application specification.

The second difference has to do with the target architecture. The traditional codesign partitioning problem is formulated for target architectures with Application Specific Integrated Circuits (ASICs), whereas PA uses Field Programmable Gate Arrays (FPGAs) and use a different approach to area and latency calculations than ASICs. Using FPGAs in hardware/software codesign systems fundamentally changes the role of area in the partitioning problem, since FPGAs have finite area and reprogrammability. Reprogrammability allows more components of a pipeline specification to be assigned to hardware than the limited device size allows, as the device can be reprogrammed. In addition, the overhead in ASIC-based designs differs from the overhead in FPGA-based designs. Both have the cost of transferring data to and from the target hardware, but FPGAs also have board initialization and reprogramming costs to account for in the latency. The pipeline assignment problem takes all of these factors into account, so the FPGA is used advantageously.

This chapter begins with a detailed description of the pipeline assignment problem. Section 4.3 describes optimal methods for solving PA, while Section 4.4 describes heuristic techniques. This chapter finishes with a discussion of performance results for these techniques. Terminology and equations are provided in the next section which are then used through out this chapter and the rest of the dissertation. These definitions can also be found in the glossary in Appendix A.

## 4.2 The Pipeline Assignment Problem Definition

A pipeline assignment maps each component in an $N$-stage pipeline to an implementation. The pipeline assignment problem is the problem of finding the

pipeline assignment for an $N$-stage pipeline and image size $z$ that has minimal latency, and area less than or equal to the hardware device. The area of the hardware device is called the *Total Area Constraint (TAC)*. PA is NP-Complete; the proof of this is in Appendix B.

An $N$-stage pipeline is defined as an ordered sequence of $N$ components, denoted $P_N = (c_1, c_2, ..., c_N)$. Each component represents an operation. An $N$-stage pipeline has a pipeline size of $N$. Some pipelines include *source* and *sink* nodes. The source and sink components are $c_0$ and $c_{N+1}$ respectively. Component $c_0$ is the pipeline source that handles any pre-processing, such as converting from an image format to a data array. Likewise, component $c_{N+1}$ is the pipeline sink that handles any post-processing, such as displaying the results. These two components are only implemented in software and do not change the optimization methods used to solve PA.

### 4.2.1 Components

A component is a way to represent an image processing algorithm in the pipeline without defining the specific implementation. Abstracting the communication boundaries for each implementation, as is discussed in Chapter 5, allows the implementations to be dynamically chosen at runtime for each component. As long as all implementations of a component have the same input/output interface, they can all be represented by one component in a pipeline.

There are two implementation types: hardware and software. Given a component $c_i$, an implementation of type $x$ is denoted $c_{ix}$. Given a component $c_i$, $I_i$ is the set of all implementations of $c_i$, $s_i$ is the set of all software implementations for $c_i$ and $h_i$ is the set of all hardware implementations for $c_i$. For each implementation $c_{ix} \in I_i$, $c_{ix}$ is a member of $s_i$ or $h_i$, but not both. In this dissertation, many components have multiple implementations, as discussed in Chapter 3. If for implementation type $x$, the set of all implementations with that type for component $c_i$, denoted $x_i$, has a size $|x_i| > 1$, then the the yth member of $x_i$ is denoted $c_{ix_y}$. The notation $c_{ix}$ is used when there is no need to

specify a specific instance.

Each implementation $c_{ix}$ has two associated costs for an image size $z$: latency $l(c_{ix}, z)$ and area $a(c_{ix})$. An implementation's latency is simply its execution time for the given image size and does not include the execution time of any overhead costs, such as communication. An implementation's latency is profiled for many image sizes. The latency for image sizes that do not have profiles are estimated through linear interpolation. The second cost, area, measures the amount of an FPGA device used when programmed with the implementation and is not a function of image size. A hardware implementation's area is a constant that represents the size of the synthesized hardware implementation. Software implementations have zero area, since they do not use the hardware device. The area of a synthesized component cannot be larger than the device's TAC to be implementable.

### 4.2.2 Pipeline Assignments

A pipeline assignment is a function $K_N$ that maps the elements of an $N$-stage pipeline $P_N = (c_1, ..., c_N)$ onto $\mathbf{I}$, where $\mathbf{I}$ is the set of implementations for all components. The assignment for a component $c_i$ is defined as $K_N(c_i) = c_{ix}$ where $x$ is the implementation type and $c_{ix} \in I_i$. A pipeline assignment is also called a solution. The solution $K_N$ is a path through all possible implementations of the components in the pipeline. For example, given a four-stage pipeline $P_4 = (c_1, c_2, c_3, c_4)$ with source $(c_0)$ and sink $(c_5)$ a pipeline assignment is $K_4 = \{(c_0, c_{0s}), (c_1, c_{1s}), (c_2, c_{2h}), (c_3, c_{3h}), (c_4, c_{4s}), (c_5, c_{5s})\}$. Figure 4.1 shows the solution $K_4$ for this example. In this figure all of the implementations for each component are listed. For example, component $c_1$ has five implementations: $c_{1s_1}$, $c_{1s_2}$, $c_{1h_1}$, $c_{1h_2}$, and $c_{1h_3}$. Likewise, component $c_2$ has five implementations: $c_{1s_1}$, $c_{1s_2}$, $c_{1s_3}$, $c_{1h_1}$, and $c_{1h_2}$. The dark nodes in Figure 4.1 are an example path for the solution $K_4$. Note that not all implementations combine with all other implementations, as is the case of $c_{1h_3}$ which only combines with $c_{2h_2}$. Given a pipeline $P_N$, and a *TAC*, a *valid* solution to PA is a pipeline assignment

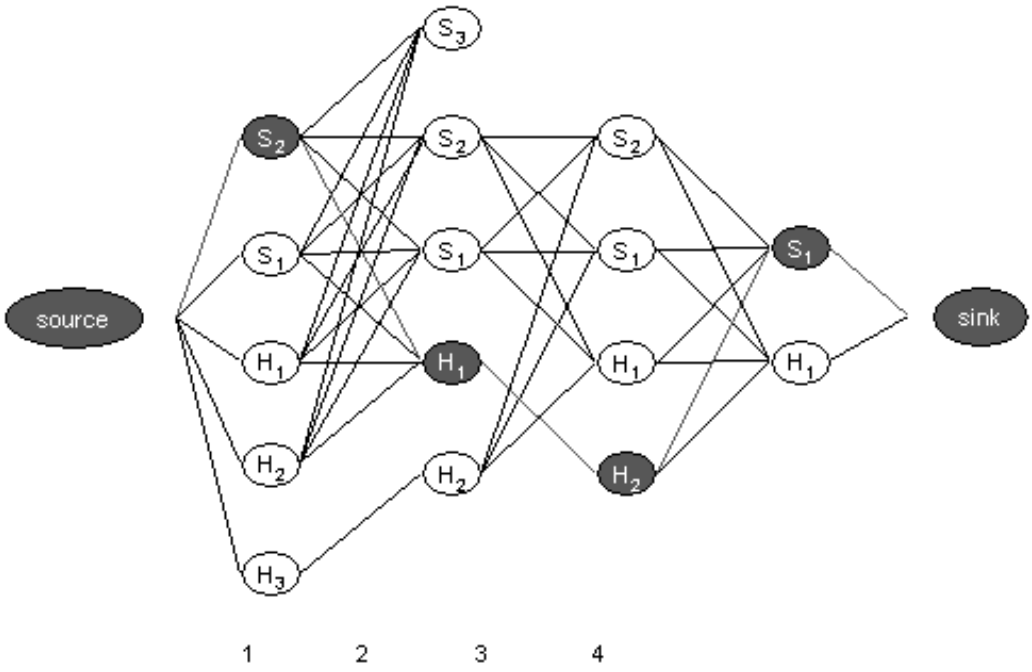


Figure 4.1: Path Through All Possible Implementations of Each Component to Construct $K_4$

$K_N$ that maps all components of $P_N$ to hardware and software implementations such that the area constraint is satisfied. An *optimal* solution $K_N^*$ is a valid solution with minimal latency for a given image size $z$. This section covers finding pipeline assignments through transformations, the definition of subsequence, and characteristics of a pipeline assignment.

### Transforming Pipeline Assignments

Constructing pipelines by adding or removing components from a base pipeline can be useful. In these cases, a solution for the base pipeline can be transformed into a solution for the constructed pipeline. The pipeline assignment transformations of interest are either growing (*augmenting*) or shrinking (*restricting*) solutions. Many $N$-stage solutions are built constructively by augmenting the

solutions of smaller pipelines. Restricted solutions have interesting properties. Augmenting and restricting solutions are defined below.

The solution $K_N$ for a pipeline $P_N$ may be the basis for a solution for a superpipeline of $P_N$. Given the pipeline $P_N$ and some $i \in \mathbb{Z}^+$, the *superpipeline* $P_{N+i}$ is defined as $P_{N+i} = (c_1, ..., c_{N+i})$ where the first $N$ components are the pipeline $P_N$. The solution $K_N$ is *augmented* to create a solution $K_{N+i}$ such that $K_{N+i}(c_j) = K_N(c_j)$ for $j \in [1, N]$ and all the values of $K_{N+i}(c_j)$ for $j \in [N+1, N+i]$ are defined. For example, the five-stage pipeline $P_5 = (c_1, c_2, c_3, c_4, c_5)$ is created from the four-stage pipeline $P_4 = (c_1, c_2, c_3, c_4)$ and the solution $K_4 = \{(c_1, c_{1s}), (c_2, c_{2h}), (c_3, c_{3h}), (c_4, c_{4s})\}$ is augmented to create the solution $K_5 = \{(c_1, c_{1s}), (c_2, c_{2h}), (c_3, c_{3h}), (c_4, c_{4s}), (c_5, c_{5s})\}$ by defining $K_5(c_5) = c_{5s}$. If solution $K_N$ is valid for some pipeline $P_N$ and device $d$, there is not enough information to determine if the augmented solution $K_{N+i}$ is valid for device $d$. The validity of $K_{N+i}$ depends on how the components $c_i$ for $i \in [N+1, N+i]$ are assigned. If solution $K_N$ is not valid for device $d$, then solution $K_{N+i}$ is also not valid for device $d$. Many of the algorithms implemented for solving PA depend on building $N$-stage pipeline assignments through the repeated augmentation of smaller solutions.

Similarly, the solution $K_N$ for a pipeline $P_N$ may be the basis of the solution for a subpipeline of $P_N$. Given a pipeline $P_N$ and some $i \in [1, N-1]$, a *subpipeline* is a pipeline $P_i = (c_1, c_2, ..., c_i)$ of the first $i$ components of $P_N$. The solution $K_N$ for $P_N$ is *restricted* to create a solution $K_i$ for $P_i$ such that $K_i(c_j) = K_N(c_j)$ for all $j \in [1, i]$. Using the previous four-stage pipeline example, a restricted solution for subpipeline $P_2 = (c_1, c_2)$ is $K_2 = \{(c_1, c_{1s}), (c_2, c_{2h})\}$. If a solution $K_N$ is valid for some pipeline $P_N$ and device $d$, then for all $i \in [1, N-1]$ the restricted solutions $K_i$ are also valid for device $d$. If a solution $K_N$ is not valid for a given pipeline $P_N$ and device $d$, then there is not enough information to determine if any of the restricted solutions $K_i$ for $i \in [1, N-1]$ are valid for device $d$. In this case, some of the restricted solutions may be valid, but there is no guarantee that any are valid. Restricted solutions are used to prove that pipeline assignments have the optimal substructure property (see Appendix B)

which is needed for the dynamic programming formulation in Section 4.3.3.

### Subsequences

Given a solution $K_N$ to a pipeline $P_N$, and $i$ and $j$ such that $1 \leq i \leq j \leq N$, a *subsequence* of $K_N$ is defined to be a sequence of implementations $K_N(c_m)$ for $m \in [i, j]$ such that all implementations in the sequence have the same type. A subsequence of implementation type $x$ is maximal if the implementation type for the implementations $K_N(i-1)$ and $K_N(j+1)$ both differ from $x$. In this dissertation, only maximal subsequences are important, and the term subsequence refers to maximal subsequence unless noted. Using the definition of maximal subsequence, every implementation in the image of $K_N$ is in exactly one subsequence and subsequences are are not divisible into smaller subsequences. Let $H$ be the set of all subsequences with implementation type hardware and $S$ be the set of all subsequences with implementation type software for a given $K_N$. As an example, if $K_4 = \{(c_1, c_{1s}), (c_2, c_{2h}), (c_3, c_{3h}), (c_4, c_{4s})\}$, then $H = \{(c_{2h}, c_{3h})\}$ and $S = \{(c_{1s}), (c_{4s})\}$. These two sets are defined such that each subsequence is in either $H$ or $S$ but not both. Given a pipeline $P_N = (c_1, ..., c_N)$ and a solution $K_N$, the *final subsequence* contains implementation $K_N(c_N)$. Final subsequences are important in the dynamic programming formulation in Section 4.3.3.

### Pipeline Assignment Properties

| | |
|---|---|
| **comm** | Cost to transfer data between the hardware and software domains. |
| **unpack** | Cost to unpack thresholded image. |
| **fix** | Cost to recompute a one pixel border on the image. |
| **none** | There are no coupling costs. |
| **N/A** | This interface type pair is invalid. |
| **pad** | Cost to add a one pixel wide border to the image. |
| **reprog** | Cost to program the hardware. |
| **unpad** | Cost to remove a one pixel wide border from the image. |

Table 4.1: Definitions of Abbreviations for Coupling Cost Tables

| From: Software | To: Software | | | |
|---|---|---|---|---|
| | Thresh | Window | Hist | Simple |
| Thresh | none | none | none | none |
| Window | none | none | none | none |
| Hist | N/A | N/A | N/A | none |
| Simple | none | none | none | none |

Table 4.2: Coupling Costs for Software/Software Boundaries

Area and latency are costs associated with pipeline assignments. These costs are calculated differently for a pipeline assignment than they are for components. Between every pair of contiguous components in a pipeline there is a boundary that changes how area and latency are calculated. There are four types of boundaries which represent the four ways two components are assigned to hardware and software implementations: software/software, software/hardware, hardware/software, and hardware/hardware. There are four interface types as specified in Table 3.2 in Chapter 3: Threshold, Windowing, Histogram, and Simple. A pipeline assignment's area and latency and the consideration of boundary costs are discussed in further detail in the following paragraphs.

The area of a pipeline assignment is determined by its hardware subsequences. The area of a software subsequence is zero. The area of a hardware subsequence, $hss$, that starts at index $m$ and ends at index $n$ is the sum of the area of the components' implementations:

$$A(hss) = \sum_{i=m}^{n} a(c_{ih}). \tag{4.1}$$

In PA, $K_N$ is a valid solution if all of the hardware subsequences have area less than or equal to the $TAC$. That is:

$$\forall hss \in H,\ A(hss) \leq TAC. \tag{4.2}$$

The area of a pipeline assignment, $A(K_N)$, is defined as the maximum hardware

| From: Software | To: Hardware | | | |
|---|---|---|---|---|
| | Thresh | Window | Hist | Simple |
| Thresh | comm + reprog | (pad:sw + comm + reprog) \| (pad:hw + comm + reprog) | comm + reprog | N/A |
| Window | comm + reprog | (pad:sw + comm + reprog) \| (pad:hw + comm + reprog) | comm + reprog | N/A |
| Hist | N/A | N/A | N/A | N/A |
| Simple | comm + reprog | (pad:sw + comm + reprog) \| (pad:hw + comm + reprog) | comm + reprog | N/A |

Table 4.3: Coupling Costs for Software/Hardware Boundaries

subsequence area. Given a solution $K_N$, $AFS(K_N)$ is the final subsequence's area, an important concept in the dynamic programming formulation discussed in Section 4.3.3. Reprogramming is done on all software/hardware boundaries, which means that each hardware subsequence can use the entire available hardware space (i.e., $TAC$). In practice, reprogramming can also occur on hardware/hardware boundaries; this will be implemented in the future. Allowing hardware/hardware boundary reprogramming broadens the definition of a valid solution.

The latency of a pipeline assignment includes the latencies of all individual implementations, as well any costs incurred crossing execution boundaries and initializing the hardware. Given implementations $c_{(i-1)x}$ and $c_{iy}$, the coupling cost of executing them in sequence is defined to be $X(c_{(i-1)x}, c_{iy}, z)$, where $z$ is the image size. The different coupling costs must be taken into account when making pipeline assignment choices. Table 4.2 to 4.5 describe the costs for each interface type pair for each of the four combinations of two implementation types (software/software, software/hardware, hardware/software, and hardware/hardware). The terms in Table 4.1 define the abbreviations used in these tables.

| From: Hardware | To: Software | | | |
|---|---|---|---|---|
| | Thresh | Window | Hist | Simple |
| Thresh | (unpack:hw) \| (unpack:sw + 2×comm + reprog) | (unpack:hw + pad:hw) \| (unpack:sw + pad:sw + 2×comm + reprog) \| (unpack:hw + pad:sw + 2×comm + reprog) \| (unpack:sw + pad:hw + 2×comm + reprog) | (unpack:hw) \| (unpack:sw + 2×comm + reprog) | (unpack:hw) \| (unpack:sw + 2×comm + reprog |
| Window | (unpad:sw + comm) \| (unpad:hw + comm) | (unpad:sw + comm) \| (unpad:hw + comm) | (unpad:sw + comm) \| (unpad:hw + comm) | (unpad:sw + comm) \| (unpad:hw + comm) |
| Hist | N/A | N/A | N/A | comm |
| Simple | N/A | N/A | N/A | N/A |

Table 4.4: Coupling Costs for Hardware/Software Boundaries

| From: Hardware | To: Hardware | | | |
|---|---|---|---|---|
| | Thresh | Window | Hist | Simple |
| Thresh | (unpack:hw) \| (unpack:sw + 2×comm + reprog) | (unpack:hw + pad:hw) \| (unpack:sw + pad:sw + 2×comm + reprog) \| (unpack:hw + pad:sw + 2×comm + reprog) \| (unpack:sw + pad:hw + 2×comm + reprog) | (unpack:hw) \| (unpack:sw + 2×comm + reprog) | N/A |
| Window | (unpad:hw) \| (unpad:sw + 2×comm + reprog) | (unpad:hw + pad:hw) \| (unpad:sw + pad:sw + 2×comm + reprog) \| (unpad:hw + pad:sw + 2×comm + reprog) \| (unpad:sw + pad:hw + 2×comm + reprog) \| (fix:hw) \| (fix:sw + 2×comm + reprog) | (unpad:hw) \| (unpad:sw + 2×comm + reprog) | N/A |
| Hist | N/A | N/A | N/A | N/A |
| Simple | N/A | N/A | N/A | N/A |

Table 4.5: Coupling Costs for Hardware/Hardware

Communication cost is the time required to transfer data to or from memory on a board. This cost is a function of the hardware implementation and the image size, so the boundary costs are unique to each pair of implementations. Extra processing steps, such as padding or decompressing an image, are steps that are necessary to execute two implementations serially. Extra processing steps are a function of the two components' interface types and their assigned implementations, and are discussed in detail in Chapter 3. Finally, there is a one-time cost for device initialization to use hardware implementations, which may be a significant portion of the latency in small pipeline instances. Hardware initialization is used only when implementations are assigned to hardware. This cost is represented with the function $hwInitCostFn(H)$, which is zero if $H$ is empty and the constant *hwInitCost* otherwise. The constant $hwInitCost$ is unique to each device. Given a pipeline $P_N$ with a solution $K_N$ and an image size of $z$ the latency is:

$$\begin{aligned} L(K_N, s) = \quad & hwInitCostFn(H) + \sum_{i=1}^{N} l(K(c_i), s) + \\ & \sum_{i=2}^{N} X(K(c_{(i-1)}), K(c_i), s). \end{aligned} \tag{4.3}$$

If there is a source and sink component, then the range of the second summation is changed to $1 \leq i \leq N + 1$.

## 4.3 Optimal Pipeline Assignments

Optimal pipeline assignments have the minimum latency for a given pipeline. NP-Complete problems are widely believed to be unsolvable in polynomial time for all instances of the problem, which makes optimal solutions prohibitively expensive to calculate as the pipeline size grows. Therefore, optimal solutions are only feasible for small pipeline instances. This section covers three options for solving PA optimally: exhaustive search, integer linear programming, and dynamic programming.

```
exhaustiveSearch (pipe, totalArea) {
  prevDS = new dataStructure();
  for (int i = 1; i <= N; i++) {
    dataStructure currDS = new dataStructure();
    for each solution in prevDS {
      for each implmentation of component i {
         potlSoln = generateNextPotlSoln(soln, impl);
         if (area < totalArea ) {
           add(currDS, potlsoln);
         }
      }
    }
    prevDS = currDS;
  }
  {bestTime, bestSoln} = findBestSoln(prevDS);
  return {bestTime, bestSoln};
}
```

Figure 4.2: Exhaustive Search Algorithm

### 4.3.1 Exhaustive Search

The exhaustive search algorithm for the pipeline assignment problem finds an optimal pipeline assignment for the given pipeline, image, and TAC. Given an $N$-stage pipeline $P_N = (c_1, c_2, ..., c_N)$, where each component has $|h|$ hardware implementations and $|s|$ software implementations for a total of $|I|$ implementations, there are $|I|^N$ solutions in the problem space. This algorithm finds the optimal solution by determining the area and latency values for each member of the problem space. The pipeline assignments are built constructively through the repeated augmentation of subpipelines from component $c_1$ to $c_N$. For example, in iteration $i$ all of the pipeline assignments from the previous iteration are augmented with all of the implementations of component $c_i$. The algorithm's runtime is reduced by calculating area and latency together and pruning any invalid pipeline assignments immediately from the data structure storing solutions for the next iteration. If either the latency exceeds the current best minimal latency or the area violates the area constraint, the calculation is terminated immediately. The best and worst case scenarios for this algorithm are based on the number of valid solutions in the problem space. The lower bound on execution time for the exhaustive search algorithm occurs when only

one member of the problem space is a valid solution and all invalid solutions are easily determined. This situation happens when the TAC is smaller than all hardware implementations. The upper bound on execution time occurs when every member of the problem space is a valid solution. This situation occurs when the TAC is large and hardware/hardware reprogramming is allowed. The time complexity of this algorithm is $O(N|I|^N)$. The algorithm is outlined in Figure 4.2.

### 4.3.2 Integer Linear Programming

Integer Linear Programming, like exhaustive search, finds optimal solutions. The integer linear programming formulation for PA is based on the Niemann/Marwedel [67] integer linear programming formulation for partitioning. Their formulation handles many of the same factors as PA, such as communication, execution time and area costs. Area costs differ in PA from partitioning, as discussed earlier. The AMPL language [3] was used to describe my formulation and CPLEX [13] to solve it. The full formulation is in Appendix C.

#### Objective Function

The objective of the integer linear programming formulation is to find an optimal solution to PA for a given pipeline, image size and device size.

#### Variables

There are three variables in the formulation. The variable *assignment* defines the mapping of each component in the pipeline to one of its implementations and is an $I \times N$ matrix, where $I$ is the number of implementations and $N$ is the pipeline size.

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

The variable *interface* defines what interface type two components have based on *assignment* and is an $I^2 \times (N-1)$ matrix, where $I$ is the number of implementations and $N$ is the pipeline size.. Interface type is defined by the implementation of the two components, and is a function of the implementations found in the ipBLOC.

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

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

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

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

## Constraints

There are several constraints in this formulation:

- The sum of all assignments for every component in the pipeline is 1.
- The sum of all interfaces for every pair of adjacent components is 1.
- Source and sink components are always 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.
- All of the variables are positive integers between zero and one.

There is a drawback when using CPLEX to solve the integer linear programming formulation. If the CPLEX solver does not find a solution within the allotted time, no solution is returned. In cases where no solution is returned, the pipeline assignment problem is solved a second time using another algorithm.

### 4.3.3 Dynamic Programming

The third approach for solving PA optimally is dynamic programming. To use dynamic programming, the optimization problem must have the *optimal substructure* property, which means that the structure of an optimal solution for a pipeline $P_N$ must contain the optimal solution for each subpipeline $P_i$ for $i \in [1, N-1]$. Pipelines are built constructively in the dynamic programming formulation by augmenting solutions. The implementation of the PA dynamic programming formulation iterates through the components of the pipeline so that iteration $i$ of the algorithm computes the optimal solutions for the $P_i$ subpipeline. In iteration $i$ of an N-stage calculation there is no way to know which of the implementations in $I_i$ are in the optimal $K_N$ solution of the $P_N$ pipeline. Therefore, optimal solutions for each implementation in $I_i$ need to be calculated. There is also no knowledge in iteration $i$ what the area of the optimal solution is. The dynamic programming formulation computes the best pipeline assignment to each implementation for each possible area size in [0, TAC]. In this sense, the dynamic programming formulation is two dimensional and there are $|I_i| \times TAC$ solutions for each iteration $i \in [1, N]$. This section covers the dynamic programming formulation, a discussion of algorithmic complexity, and examples of dynamic programming execution. This section relies heavily on the definitions from Section 4.2, which can be found in the glossary in Appendix A.

## Dynamic Programming Formulation

Given a pipeline $P_N$ an implementation $x$ of component $c_N$ and an area $y \in [0, TAC]$, $f_N^*(x, y)$ is the minimum latency of an optimal solution $K_N^*$ for $P_N$, if one exists, such that the last component's implementation $K_N^*(c_N)$ equals $x$ and the final subsequence's area $AFS(K_N^*)$ equals $y$.[1] If there does not exist a $K_N^*$ that satisfies the given $x$ and $y$, $f_N^*(x, y) = \infty$. Given the pipeline $P_N$ and some $i \in [1, N]$ the optimal latency for the subpipeline $P_i$ for an implementation $x = K_i(c_i)$ and area of the final subsequence $y = AFS(K_i)$ is defined recursively as:

$$f_i^*(x,y) = \begin{cases} l(x,s) & \text{if } i = 1 \\ \min\limits_{\substack{z \in I_{i-1} \\ a \in B(x,y)}} \{f_{i-1}^*(z,a) + X(z,x,s) + l(x,s)\} & \text{otherwise} \end{cases}$$

where component $c_i$ has implementation $x$, the image size is $z$, the final subsequence has area $y$, $I_{i-1}$ is the set of all implementations for component $c_{i-1}$ and $B(x, y)$ is defined as:

$$B(x,y) = \begin{cases} \{0, ..., TAC\} & \text{if } x \text{ is implemented in software} \\ \{y - A(x)\} & \text{if } x \text{ is implemented in hardware.} \end{cases} \tag{4.4}$$

The function $B(x, y)$ defines the set of possible values $AFS(K_{N-1})$ can have to satisfy $K_N^*$. For example, given $f_i^*(x, y)$ where $x$ is a hardware implementation of component $c_i$, the only $K_{i-1}$ solutions that are possible subsolutions to $K_i^*$ have $AFS(K_{N-1}) = y - a(x)$. If $x$ is a software implementation, reprogramming has occurred and any valid $K_{i-1}$ solution is a candidate subsolution for $K_i^*$. For $f_i^*(x, y)$ the range of values for $x$ is $I_i$ and $y$ is $B(x, y)$. The values for $f_i^*(x, y)$ form the matrix $I_i \times |B(x, y)|$. The size of the set is $1 \leq |B(x, y)| \leq TAC$. In reality, the $f_i^*(x, y)$ matrix is very sparse, where most of the values

[1] The latency and solution functions known to be optimal are denoted with asterisks.

are set to infinity. Hardware initialization adds a non-linearity to the model, and it is temporarily removed from the latency calculation. The non-linearity is discussed further in the implementation section. The proof that PA has optimal substructure is given in Appendix B.

Solutions for a pipeline $P_N$ are found by iteratively augmenting the solutions to subpipelines starting with $P_1$. When pipelines have a source component, the $P_1$ solution includes the source component. The initial pool of solutions for iteration $i$ is created by augmenting each non-infinite solution in the matrix $f^*_{i-1}(x, y)$ with all implementations of component $c_i$, which means that the size of the initial pool of solutions is $|I_i||f^*_{i-1}|$, where $I_i$ is the set of implementations for component $c_i$. The augmented solution for iteration $i$ that is created by defining $K_i(c_i) = x$ where $x \in h_i$ is called a *solution ending in a hardware assignment (SEHA)*. Similarly, augmenting by defining $K_i(c_i) = x$ where $x \in s_i$ is called a *solution ending in a software assignment (SESA)*.

The dynamic programming formulation executes faster than the exhaustive search technique due to problem space pruning. Any time an invalid solution $K_N$ is created, it is immediately pruned from future calculations by setting $f^*_{i-1}(K_N(c_N), AFS(K_N)) = \infty$. Pruning valid solutions is also possible, if they are proven to be non-optimal. If two or more $K_i$ solutions exist with component $c_i$ assigned to implementation $x$ and a final subsequence area of $y$, then the solution with the smallest latency is saved for the next iteration and all other solutions are pruned. If two or more $K_i$ solutions with the same assignment for component $c_i$ but different final subsequence area of $y$, then all such solutions are saved. In this situation, there is no way to predict in the $i$th stage of computation which solution will be minimal in the $N$th stage of computation. Since the SESAs all have zero final subsequence area, most of the SESAs are pruned. In fact, the only SESAs saved for the next iteration are the minimal solution for each software implementations for a total of $|s_i|$ solutions. Given a software implementation $x$ of component $c_i$, all SESAs defined with $K(c_i) = x$ have the same area, which means that only one SESA per software implementation is saved for each iteration. SEHAs are pruned when either there exists a solution

with the same final component implementation and area of final subsequence, or the area constraint is not met. As a worst case estimate, the number of solutions saved in iteration $i$ is $|h|^i + |s| \sum_{k=0}^{i-1} |h|^k$ (this formula is derived in Appendix B). The maximum number of solutions saved with dynamic programming is still significantly smaller than with exhaustive search, which saves $(|h| + |s|)^i$ solutions per iteration.

**Example**

As an illustration of solving PA with dynamic programming, an artificial four-stage pipeline is solved. For this example, components are limited to only one hardware and one software implementation each. Table 4.6 shows the hardware and software execution times for each components for a given image size. The zeroth component is the source and does not have a hardware implementation. We make the following assumptions:

1. The same amount of data is being transfered to and from the board each time there is a communication, and this cost is 25 ms for a one way trip.
2. Reprogramming costs 70 ms.
3. Hardware initialization costs 100 ms.
4. The hardware implementations all have equal area.
5. The *TAC* is large enough to fit four hardware implementations in one hardware subsequence.

Note that SEHAs are not pruned since each SEHA has a unique area for the final subsequence. Pruning occurs only for SESAs.

Figure 4.3 represents the tree of solutions obtained when applying dynamic programming to this example. Each node represents a component implementation and level $i$ represents the subsolution to PA after $i$ components have been assigned. Intermediate results for the latency computation are shown in milliseconds. For example, the latency value of 682 ms (300 ms + 100 ms + 25 ms

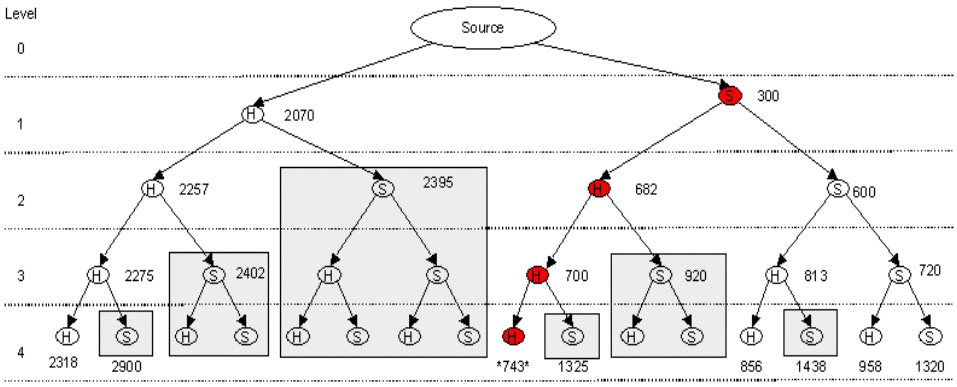


Figure 4.3: Computation Tree for Dynamic Programming that Shows Intermediate Results and Tree Pruning

| Component | Hardware Runtime (ms) | Software Runtime (ms) |
|---|---|---|
| 0 | $\infty$ | 0 |
| 1 | 1875 | 300 |
| 2 | 187 | 300 |
| 3 | 18 | 120 |
| 4 | 18 | 600 |

Table 4.6: Software/Hardware Runtimes for a Source and Four Components

+ 70 ms + 187 ms = 682 ms) in the second level of the tree includes the execution costs for the first two components (300 ms + 187 ms = 487 ms), as well as hardware initialization (100 ms), reprogramming (70 ms) and communication costs (25 ms) necessary for this solution. The grey boxes in the figure indicate branches that are pruned from computation. Nodes without latency numbers next to them are never visited. The optimal solution is marked with asterisks. Note that the optimal solution is not eliminated and that several non-optimal solutions are safely eliminated from computation.

This figure shows solutions that have been pruned safely from computation. In the second level there are two SESAs: $K_2^A = \{(c_0, c_{0s}), (c_1, c_{1h}), (c_2, c_{2s})\}$ and $K_2^B = \{(c_0, c_{0s}), (c_1, c_{1s}), (c_2, c_{2s})\}$. Because the latency of $K_2^A$ (2395 ms) is larger

than the latency of $K_2^B$ (600 ms) there is no need to expand the subtree below $K_2^A$. What follows is an explanation of why this subtree can be pruned.

In the second iteration there are two SESAs: $K_2^A$ and $K_2^B$. We know that the subtree below both solutions is the same and the solution to the subtree is $K_{2..4}$. Since $K_2^A$ and $K_2^B$ end with the same implementation, the cost for joining $K_{2..4}$ to them is the same. This last statement has both area and latency implications. Since the area of $K_2^A$ and $K_2^B$ are the same, then the solutions created by augmenting these solutions with $K_{2..4}$ also have the same area. Therefore, there is no area reason to keep both solutions. Also, since both solutions are being augmented with the same values from the subtree solution and have the same assignment for $c_2$, then the same amount of latency is added to both solutions by augmenting them with the assignments from $K_{2..4}$. Therefore, since $K_2^B$ has smaller latency than $K_2^A$, the $K_4^B$ solution will continue to be faster than the $K_4^A$ solution. Since the solution that is fastest in the second iteration remains the fastest at end of computation, there is no reason to keep the larger latency solution after the second iteration. The $K_2^A$ solution is eliminated from computation during the algorithm's second iteration without removing a potentially optimal solution.

Figure 4.3 also shows several partial solutions that are not pruned. In the second level there are two SEHAs: $K_2^C = \{(c_0, c_{0s}), (c_1, c_{1h}), (c_2, c_{2h})\}$ and $K_2^D = \{(c_0, c_{0s}), (c_1, c_{1s}), (c_2, c_{2h})\}$. These two SEHAs have two different areas: $A(K_2^C) = a(c_{1h}) + a(c_{2h})$ and $A(K_2^D) = a(c_{2h})$. Since they have unique areas, these two partial pipeline assignments are saved. When determining the partial assignments for component $c_2$, there is no way to determine whether $K_2^C$ or $K_2^D$ are subsolutions to the optimal solution. Subsolutions are not pruned until they are absolutely determined to be not part of the optimal solution.

## Dynamic Programming Implementation

In this section, we discuss the translation of the dynamic programming formulation into software. This section focuses on how new solutions are created, and

how the non-linearity of the hardware initialization cost is resolved.

Figure 4.4 shows the pseudocode for the dynamic programming formulation. The algorithm returns the optimal solution and the optimal latency. When computing the latency of a solution $K_i$ for an image of size $z$, the latency is calculated constructively:

$$\begin{aligned} L(K_i, s) = \quad & L(K_{i-1}, s) + X(K_i(c_{i-1}), K_i(c_i), s) + l(K_i(c_i)) + \\ & hwInit(K_{i-1}, K_i). \end{aligned} \tag{4.5}$$

where the value $L(K_{i-1}, s)$ is calculated in the previous iteration, the value $X(K_i(c_{i-1}), K_i(c_i), s)$ determines the coupling cost of $K_i(c_{i-1})$ and $K_i(c_i)$, and $hwInit(K_{i-1}, K_i)$ determines whether the hardware initialization cost needs to be included. If the hardware initialization cost is included in $K_{i-1}$, then $hwInit(K_{i-1}, K_i) = 0$. If the hardware initialization cost is not included in $K_{i-1}$ and implementation $K_i(c_i)$ is assigned to a hardware implementation, then $hwInit(K_{i-1}, K_i)$ equals the hardware initialization cost, $hwInitCost$. If the hardware initialization cost is not included in $K_{i-1}$ and implementation $K_i(c_i)$ is assigned to a software implementation, then $hwInit(K_{i-1}, K_i) = 0$.

There are two data structures in use during the algorithm's execution: one with the previous iteration's solutions and one with the current iteration's solutions. Solutions for iteration $i$ are created by augmenting all solutions stored in the previous iteration's data structure with all the implementations for component $c_i$. A new solution $K_i$ is saved to the current data structure in the *addOrNot* method if these conditions are met:

1. $K_i$ is a valid solution.

2. If there does not exist $K_i^{'}$ in the current iteration's data structure such that $AFS(K_i) = AFS(K_i^{'})$ and $K_i(c_i) = K_i^{'}(c_i)$, or

3. If there exists $K_i^{'}$ in the current iteration's data structure such that $AFS(K_i) = AFS(K_i^{'})$ and $K_i(c_i) = K_i^{'}(c_i)$ , and $L(K_i) < L(K_i^{'})$.

```
dynamicProgramming (pipe, totalArea) {
  prevDS = new dataStructure();
  for (int i = 1; i <= N; i++) {
    dataStructure currDS = new dataStructure();
    for each solution in prevDS {
      for each implmentation of component i {
         potlSoln = generateNextPotlSoln(soln, impl);
         addOrNot(currDS, potlsoln);
      }
    }
    prevDS = currDS;
  }
  {bestTime, bestSoln} = findBestSoln(prevDS);
  return {bestTime, bestSoln};
}
```

Figure 4.4: Dynamic Programming Implementation

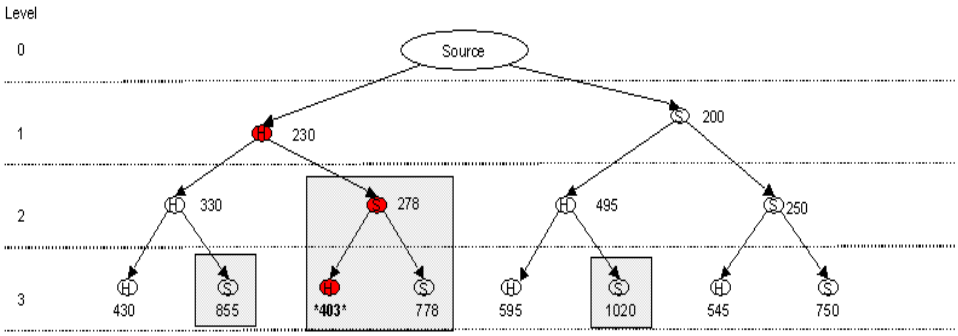


Figure 4.5: Hardware Initialization Cost Non-linearity

In the third case, the solution $K_i'$ is removed when the $K_i$ solution is added to the data structure. A solution is saved with its latency and area properties to avoid recomputing these values in the next iteration.

The dynamic programming implementation must correct the non-linearity caused by the hardware initialization cost. The remainder of this section addresses the conditions which cause the non-linearity to allow eliminate the optimal solution and how the implementation avoids eliminating it. An example of how the non-linearity comes into play is shown in Figure 4.5. This example is based on the component implementation runtimes found in Table 4.7. As with the previous example, we assume one way communication cost is 25 ms,

| Component | Hardware Runtime (ms) | Software Runtime (ms) |
|---|---|---|
| 0 | $\infty$ | 0 |
| 1 | 35 | 200 |
| 2 | 100 | 50 |
| 3 | 75 | 500 |

Table 4.7: Software/Hardware Runtimes for Four Components

hardware initialization is 100 ms, and reprogramming is 70 ms. The zeroth component is the source, and does not have a hardware implementation. All hardware implementations are the same size and the sum of all three hardware implementations is less than the *TAC*. In this example, the optimal solution $K_3^* = \{(c_0, c_{0s}), (c_1, c_{1h}), (c_2, c_{2s}), (c_3, c_{3h})\}$ is eliminated in the algorithm's second iteration. In cases where the optimal solution $K_N^*$ for pipeline $P_N$ has a mixture of hardware and software implementations, the optimal solution is pruned from the computation in iteration $i$ if all of these three conditioners are met:

1. The SESA for the software implementation $x$ in iteration $i$ is one software subsequence, which means that the hardware initialization cost has not been paid.

2. The optimal solution has already assigned at least one component to a hardware implementation by the $ith$ iteration and will assign the $ith$ component to the software implementation $x$. This solution has paid the hardware initialization cost.

3. The difference in latency between these two solutions is less than the hardware initialization cost.

The problem persists until all SESAs include at least one assignment to a hardware implementation. Special entries in the data structure track what the SESA would be if the hardware initialization cost was included in the latency.

The optimal solution is preserved with this modification, even when the three conditions above occur.

## 4.4 Heuristic Solutions

While optimal solutions are ideal, the nature of the pipeline assignment problem makes optimal solutions infeasible for large pipeline sizes. Heuristic techniques find solutions within the decision making time frame. In this section, these heuristic methods are covered: greedy, random, and local search.

### 4.4.1 Greedy

The greedy algorithm is a heuristic that finds a solution to a decision problem by making locally optimal decisions. For the pipeline assignment problem, instead of making a decision for the entire pipeline, a decision is made for each component. For each component, the decision to map to hardware or software is based on the area and speed requirements. If the component's hardware implementation is too large to be considered, then the component is assigned to the software implementation. Otherwise, the implementation of the current component with the smallest execution time is chosen. The pseudocode for the algorithm is shown in Figure 4.6. The time complexity of this algorithm assuming one hardware and one software implementation is $\Theta(N)$.

Since PA does not have the greedy property where globally optimal solutions are found through locally optimal decisions, not all pipeline instances can be solved optimally using the greedy algorithm. The greedy algorithm can find good solutions contingent on several factors. First of all, the order of the pipeline components has a large effect on the solution. If the same components are arranged into two different pipelines, greedy might generate a near-optimal solution for one pipeline and a pathologically bad solution for the other. Second, in cases where speedup is always gained by assigning components to hardware,

```
GreedyAlgorithm(pipeline, totArea) {
  Area = 0;
  N = pipeline.size();
    For i = 0 to i < N {
      comp = pipeline.getComponent(i);
      HWTime = calcHWRuntime(pipeline, i);
      SWTime = calcSWRuntime(pipeline, i);
      if ((Area+calcArea(pipeline,i)) < totArea){
        if(HWTime > SWTime) {
          pipeline.assignComponent(i,SW);
          Area = 0;
        } else {
          pipeline.assignComponent(i,HW);
          Area += calcArea(pipeline, i);
        }
      } else {
        pipeline.assignComponent(i,SW);
        Area = 0;
      }
    }
}
```

Figure 4.6: Greedy Algorithm

the area costs of the hardware implementations determine how components get assigned to software. Greedily placing components with trivial speedup in hardware might cause components with large speedup to be forced into software due to lack of space. Third, hardware implementations that contain special edge processing are not used, since there is no way to know whether a suitable implementation for the next component will be found that satisfies both the coupling rules and the area requirements for a solution. The only implementations used are the ones that can be combined with any other component implementations. These hardware implementations give the most limited speed up available. There are many ways the greedy algorithm could be rewritten to gain better quality solutions, but fundamentally the greedy algorithm makes decisions too simply to find optimal solutions.

### 4.4.2 Random

The random algorithm is similar to the greedy algorithm. At each stage one of the possible implementations is chosen for the given pipeline, based on a randomly generated value. If the chosen component implementation does not

pass the area constraint, the algorithm randomly chooses another implementation. If after ten attempts a suitable implementation is not found, the software implementation is chosen as it always passes the area constraint. Like greedy, the hardware implementations with extra processing are not allowed in these solutions, as there is no way to guarantee whether the next component has a compatible implementation. The random algorithm can be used as a method for finding valid solutions, but is more useful when combined with the local search heuristic.

### 4.4.3 Local Search

Local Search algorithms take an initial solution generated by a heuristic and tries to improve it. There are many different local search algorithms, such as simulated annealing and tabu search. Pseudocode for a generic local search algorithm that solves the pipeline assignment problem is shown in Figure 4.7. The *improve* function looks at all the related solutions (*neighborhood*) to a given solution and returns a "better" solution, called the *point*. The process is then repeated using the point in place of the initial solution. In this manner, the algorithm may be able to gradually move from the initial solution to a better solution. There are four parameters to a local search algorithm: initial solution, neighborhood, acceptance criterion, and stopping criterion.

A direct path through the solution space from any solution to the optimal solution may not exist, so the local search algorithm explores many areas of the problem space in an attempt to find paths to optimal solution. Using the four parameters of the local search algorithm, different variations can be created in an attempt to find the best way to find near-optimal solutions using local search. Lenstra and Aarts [1] call local search a "homemade voodoo optimization technique," since the algorithm is customized to find good solutions based on knowledge about the problem instead of using knowledge about heuristics. For the pipeline assignment problem, results show that most large pipelines are

```
LocalSearch(pipeline, totArea, time) {
  bestLatency = inf;
  point = initSolution(pipeline, totalArea);
  Start, end = System.getCurrentTime();
  boolean done = false;
  while (!done && end - start < time){
    point = improve(point, totArea);
    if (point != null) {
      pointLatency = calcTime(point);
      if (pointLatency < bestLatency) {
        bestLatency = pointLatency;
        bestSol = point;
      }
    }
    else
    done = true;
    end = System.getCurrentTime();
  }
  return bestTime;
}
```

Figure 4.7: Local Search Algorithm

assigned mainly to hardware implementations, so an all hardware initial solution is expected to find good quality solutions for large pipelines. We have implemented the local search algorithm with four initial solutions, three neighborhood sizes and two acceptance criteria for a total of 24 different variations on local search. A time limit on decision time provides a natural termination mechanism. The other three aspects are discussed below.

### Initial Solutions

Since there is a limit on decision time, quick heuristics are needed to find the initial solution. The greedy and random algorithms return a solution quickly for all problem sizes. Two other initial solutions, which take virtually no time to create, assign either all components to a software implementation or all components to a hardware implementation. The all hardware solution supports reprogramming of the hardware device by inserting a hardware/software and a software/hardware boundary between each component. If the only hardware implementation for a given component is larger than the $TAC$, then the component is assigned to software. Sometimes a near-optimal solution is less likely to have a good path to the optimal solution, so good and poor initial solutions

were experimented with. The all software and greedy initial solutions provide us with good initial solutions for small pipelines and poor initial solutions for large pipelines. The random and all hardware initial solutions provide poor initial solutions for small pipelines and good initial solutions for large pipelines.

### Neighborhood

Given a solution $K_N$ to a pipeline $P_N$, the neighborhood $\aleph_N$ defines the set of solutions such that for all solutions $K_N^{'} \in \aleph_N$, $K_N^{'}$ is considered "close" to $K_N$. A *k-opt* or a *k-exchange* neighborhood [1, 71] is used, and we implemented algorithms for $k = 1$, $k = 2$ and $k = 3$. The 1-opt neighborhood includes all solutions made by changing one component's implementation. The 2-opt neighborhood includes all solutions made by changing two or fewer components' implementations. The 3-opt neighborhood includes all solutions made by changing three or fewer components' implementations. We say that an implementation is *swapped* if we change its implementation.

The neighborhood is searched in the *improve* function, which is in the inner loop of the local search algorithm. Increasing the value of $k$ generally improves the quality of the local search solutions found, but increases runtime, as the complexity of the *improve* function is $O(N^{2k})$. Increasing $k$ may not help find better solutions due to the limited overall runtime.

The local search algorithm has the same problem with using implementations with extra processing that the greedy and random algorithms had when making locally optimal decisions. When choosing a new implementation for swapping, there is not enough knowledge about which implementation is compatible with the neighboring implementations to determine if extra processing can be used. The simplest way to solve this problem is to restrict the implementation swaps to only the implementations that do not conflict with any possible neighboring implementations. This restriction is used with greedy and random algorithms. This approach is feasible but limits the solutions to the problem space where none of the components use the implementations with extra processing. Since most optimal solutions do as much edge processing in hardware as possible, the

solutions from this approach are poor.

Another approach was devised using the concept of conflict resolution from graph coloring. This neighborhood is based on the *k-opt* neighborhood, but allows all implementations to be considered for swaps. Component $c_i$ can be swapped for an implementation $x \in I_i$ as long as the conflicts are resolvable with the adjoining components $c_{i-1}$ and $c_{i+1}$. For example, if between components $c_i$ and $c_{i+1}$ there is some extra processing done in software that local search swaps into hardware in the component $c_i$ implementation, then the extra processing needs to be removed from the coupling between components $c_i$ and $c_{i+1}$. Components $c_{i-1}$ and $c_{i+1}$ are examined to see if changes in edge processing or implementation will resolve the conflict. When the component implementations for $c_{i-1}$ and $c_{i+1}$ are examined, the code tries to find an implementation that performs with minimal runtime. Once components $c_{i-1}$ and $c_{i+1}$ are altered to accommodate the $c_i$ change, conflicts may exist between components $c_{i-1}$ and $c_{i-2}$ and between components $c_{i+1}$ and $c_{i+2}$. The conflict resolution process continues with the $c_{i+1}$ and $c_{i-1}$ components. The conflict resolution process continues moving the right and the left of the $c_i$ component until no component implementations in either direction are changed. In some cases a conflict cannot be resolved and the $c_i$ implementation change is rejected. This approach allows implementations with internal edge processing to be gradually worked into the solutions. Because the entire pipeline may be traversed to fix all conflicts, the *improve* runtime is $O(N^{2k})$, instead of $O(N^k)$.[2]

### Acceptance Criteria

The neighborhood codifies the area of the problem space the next point lies in, but does not determine which solution to choose as the point. The acceptance criterion determines how the point is chosen. Two acceptance criteria are used: steepest descent and tabu search. Steepest descent accepts only the solution in the neighborhood with the smallest latency as the point. Given the point $K_N$

[2]This approach could be used with the Greedy and Random algorithms. We choose not to so that the runtimes of these heuristics are fast.

with neighborhood $\aleph_N$, if $K_N$ is the best solution in the neighborhood, then the solution is a local minimum. Steepest descent only accepts solutions as the next point, if the latency for the next point is smaller than the latency of the current point. Since steepest descent only accepts "improving" solutions as points, local minima cause the local search algorithm to halt when this acceptance criterion is used. The problem space for PA has many local minimum, so this acceptance criterion may cause local search to halt before the time limit elapses and miss better solutions.

Metaheuristics, such as tabu search, escape local minima by accepting non-improving solutions as the point. Accepting non-improving solutions may cause the algorithm to backtrack through old points. To avoid retracing through old points or getting caught in cycles where a number of old points are revisited, a tabu list is used. The tabu list $T$ keeps track of the last $|T|$ points, and no solution already on the tabu list is accepted as a point. The tabu list gets rid of any cycles involving $|T|$ or fewer points. Since non-improving solutions are allowed as points, tabu search can hill-climb out of local minima. In cases where a solution that has a smaller latency than the current point exists in the neighborhood, this solution is always accepted (*aspiration criterion*). Aspiration allows steepest descent moves in a tabu search. The length of the tabu list is also definable. While long tabu lists could decrease the likelihood of getting caught in cycles, storing many solutions may cripple the memory space. These two issues need to be balanced.

### Time Complexity

The time complexity of the local search algorithm is $O(N^{2k}l)$, where $k$ is the number of implementations swapped in the neighborhood and $l$ is the number of iterations. The number of iterations is not predictable, since our termination criterion is a fixed time limit.

## 4.5 Results

In the previous section, we presented many optimal and heuristic techniques for solving the pipeline assignment problem. In this section, the algorithms are analyzed using two different time limits. One time limit is a constant of 500 ms for all pipeline sizes, and the other time limit is 100 ms per pipeline stage. We call the first time limit the *fixed time limit* and the second one the *adaptive time limit*. The fixed time limit is small enough that decision time does not affect overall processing time. The adaptive time limit allows the time limit to change with pipeline size, as the image analyst may want to spend more time solving PA for large pipeline instances than for small pipeline instances. We want to determine which algorithms to use for each pipeline size and time limit so that PA can be solved in the runtime environment optimally or near-optimally. In this section, algorithms are analyzed in terms of their speed and their ability to return good solutions. This section closes with recommendations for algorithms to use in the runtime environment. Before starting the analysis, our experimental setup and metrics are defined.

We are interested in providing good algorithms for solving pipeline sizes 1 through 20. Thirty random pipelines were created for each pipeline length for a total of 600 pipelines. All algorithms (exhaustive search, dynamic programming, greedy, random and all 24 variations of local search) were tested with all 600 pipelines. The random pipelines were generated using the algorithm in Figure 5.4 in Chapter 5. In the case of exhaustive search, large pipelines do not run to completion so the test does not cover the entire range of test pipelines. Each algorithm was tested with three TACs: 0 (*TAC1*), the size of the Wildcard device (*TAC2*), and the maximum integer size (*TAC3*). The TAC often affects the algorithm's runtime and ability to find good solutions. The different TAC sizes allow experimentation with three cases: no hardware or hardware is busy; realistic hardware size; and a hardware device larger than all hardware implementation areas. The algorithms were run for all combinations of TAC and time limit (i.e., TAC1 and fixed time limit, TAC1 and adaptive time limit, TAC2 and

fixed time limit, etc). All tests were run on a machine with a Pentium 3 650 Mhz processor and 512 Mb of RAM. The results for all tests are in Appendix D.

The two metrics we are interested in are execution time and solution quality. The first metric of execution time is measured for each algorithm by comparing the machine's clock before and after the algorithm executes. An error of $\pm 50$ ms affects this measurement; this is only significant for the smallest pipelines for most algorithms. The second metric is the solution quality, which is defined as the latency of a solution divided by the latency of the optimal solution. Optimal solutions to all 600 pipelines were found using dynamic programming. Solution quality is used to judge the quality of solutions found by the heuristic techniques. For the heuristic techniques, the solution quality is greater than or equal to one.

### 4.5.1 Execution Time

We first examine the execution time of each algorithm. Except for local search, which can easily be constrained to return its best result after a particular time limit, the algorithms are run until they reach completion. The algorithms are analyzed for the effect of pipeline size and TAC size on the execution time. This section covers both optimal and heuristic techniques. Once the algorithms are analyzed, recommendations for each algorithm are made.

#### Optimal Techniques

The execution time for optimal techniques are a function of the device size, pipeline size, and the amount of pruning done by the algorithm. First, we present an analysis of the device size's effect on execution time. We found that increasing the TAC greatly increased the execution time for the implementation of the optimization methods. Figure 4.8 shows the average execution times for dynamic programming for all three TACs for 1- to 20-stage pipelines. This graph shows that the average execution time for dynamic programming has a distinct trend for each TAC size, and the larger the TAC the longer the average

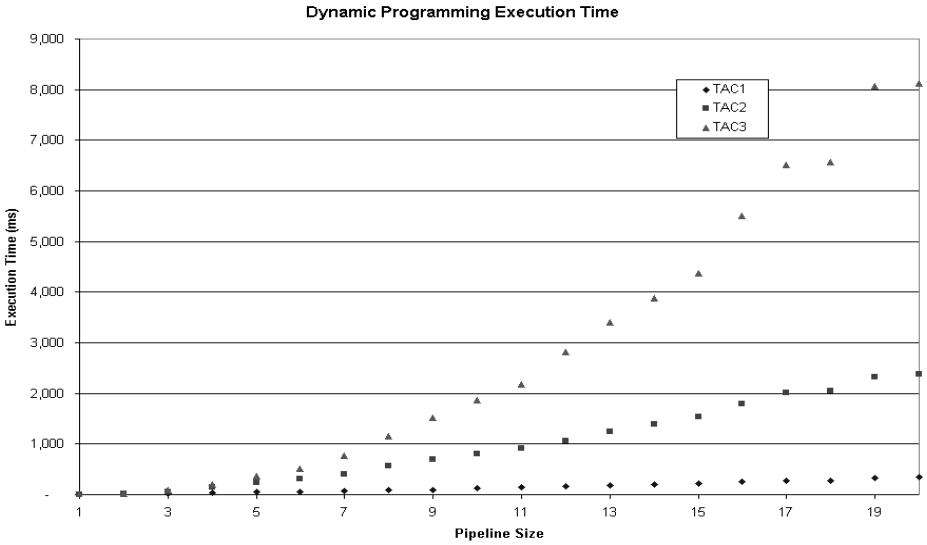


Figure 4.8: Dynamic Programming Execution Times

execution time. For dynamic programming, the execution time for a 20-stage pipeline takes an average of 887 ms for TAC1, 10421 ms for TAC2, and 30954 ms for TAC3. Figure 4.9 shows the average execution times for exhaustive search for all three TACs for 1- to 5-stage pipelines. Exhaustive search does not run all pipelines with 6 or more stages to completion, so this graph focuses on the area it does solve. The disparity between the runtimes between TAC1 and TAC3 are great enough that the scale on Figure 4.9 is logarithmic. The average runtime for TAC3 is approximately two orders of magnitude longer than TAC1. For exhaustive search, the execution time for a 5-stage pipeline takes an average of 56 ms for TAC1, 5527 ms for TAC2, and 18853 ms for TAC3. The runtimes of all the optimal techniques are dependent on the number of valid solutions in the problem space. Since valid solutions by definition must meet the area requirement of the hardware device, the number of valid solutions in a problem space is a function of the device size. Therefore, increasing the size of the TAC increases the number of valid solutions in the problem space which in turn

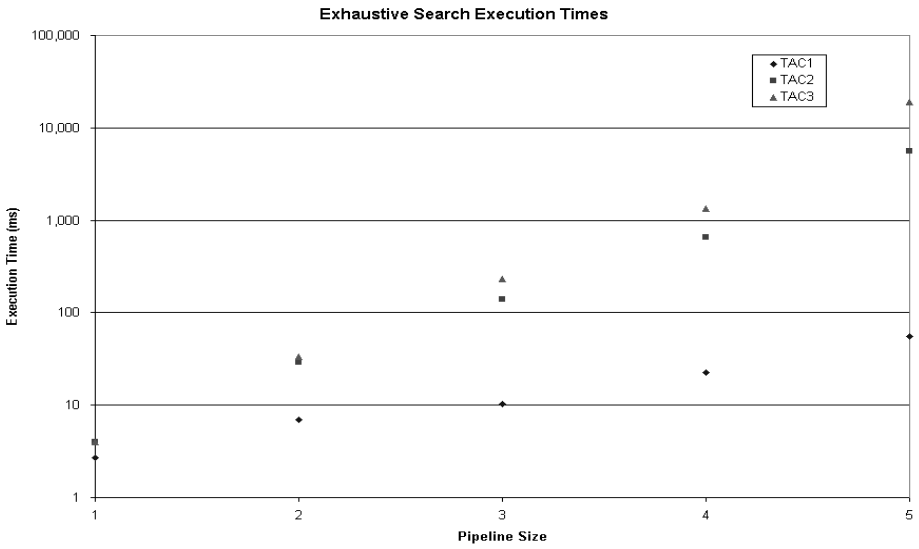


Figure 4.9: Exhaustive Search Execution Times

increases the execution time.

There is a large disparity between the runtimes of dynamic programming and exhaustive search. Figure 4.10 shows a comparison of the exhaustive search and dynamic programming execution times for TAC2 for 1- to 5-stage pipelines. This graph is done with a logarithmic scale, as the runtime for exhaustive search is much larger than the runtime for dynamic programming. From this figure we can see the average execution time for a 5-stage pipeline is 230 ms for dynamic programming and 5527 ms for exhaustive search. This difference is caused by problem space pruning. The dynamic programming implementation immediately prunes any invalid solutions and any valid solutions that are known to be not optimal, which has a direct effect on execution time. In dynamic programming, each pruned solution in iteration $i$ eliminates the construction and calculation of $|I_{i+1}|$ solutions in iteration $i + 1$. Exhaustive search only prunes invalid solutions, which means that exhaustive search stores significantly more solutions per iteration than dynamic programming. For iteration $i$, the upper

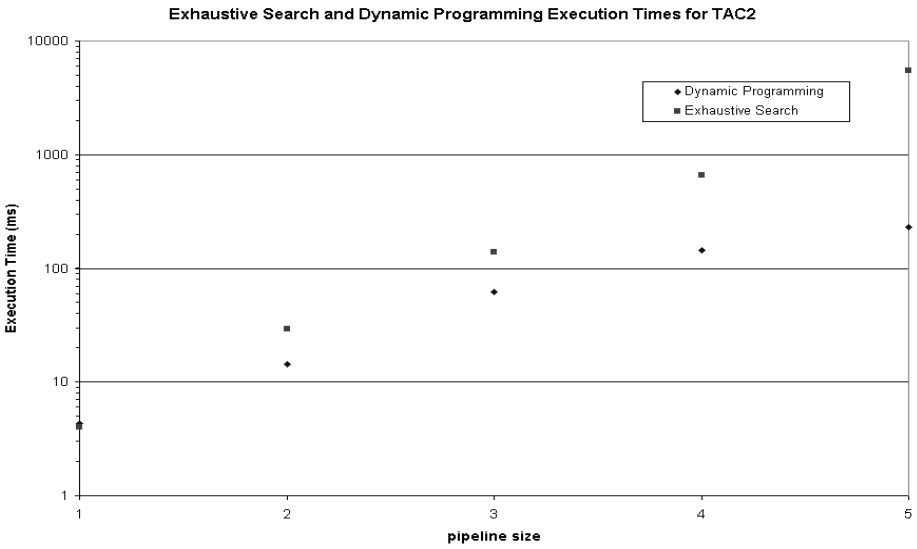


Figure 4.10: Comparison of Exhaustive Search and Dynamic Programming Execution Times (TAC2)

limit for the number of solutions saved are $|I_i|^i$ solutions for exhaustive search and $|h|^i + |s| \sum_{k=0}^{i-1} |h|^k$ solutions for dynamic programming. As an example, we compare the number of solutions stored for a few iterations of exhaustive search and dynamic programming when all components have seven implementations. In iteration three, exhaustive search stores 385 solutions, and dynamic programming stores 265 solutions. In iteration four, exhaustive search has to build and calculate the area and latency for 2695 solutions, but dynamic programming does the same calculations on 1855 solutions. Therefore, dynamic programming is doing significantly less work than exhaustive search, which is why there is such a disparity between their runtimes.

Finally, the runtime for the optimal techniques are also affected by pipeline size. As the pipeline size increases, so does the problem space and the number of valid solutions. Figure 4.9 shows that the average execution time for exhaustive search grows exponentially with problem size for all three TACs. Figure 4.8 shows that the average execution time for dynamic programming grows as a

function of device and pipeline size. The breakdown of the runtimes for dynamic programming for the three TAC sizes are:

- **TAC1:** proportional to the pipeline size,
- **TAC2:** proportional to the square of the pipeline size, and
- **TAC3:** proportional to the cube of the pipeline size.

Note that no results for integer linear programming are presented in this section. All of the algorithms were originally designed for the problem space where each component has one hardware and one software implementation. In that problem space, integer linear programming rarely performed better than exhaustive search. When we adapted exhaustive search for problem spaces with more component implementations, we found it was only useful for very small pipeline instances and decided that adapting the integer linear programming formulation was not worthwhile.

### Heuristic Techniques

With the exception of local search, execution times for the heuristic techniques are a function of pipeline size. Figures 4.11 and 4.12 show that the greedy and random algorithms' average runtimes grow linearly with pipeline size. As can be seen in Tables D.1.2 and D.1.2, the average execution time for creating either all hardware or all software pipeline assignments is barely measurable and takes approximately 1 ms per pipeline stage.

The local search algorithms' execution times are a function of the stopping and acceptance criteria; device size plays no role. Steepest descent variations of the local search algorithm halt either in a local minimum or when the time limit is reached, which ever happens first. In our analysis, this variation of local search finds local minima very quickly, which indicates that our initial solutions are close to or in local minima. The tabu search variations of local search usually halt when the time limit is exceeded. Tabu search sometimes

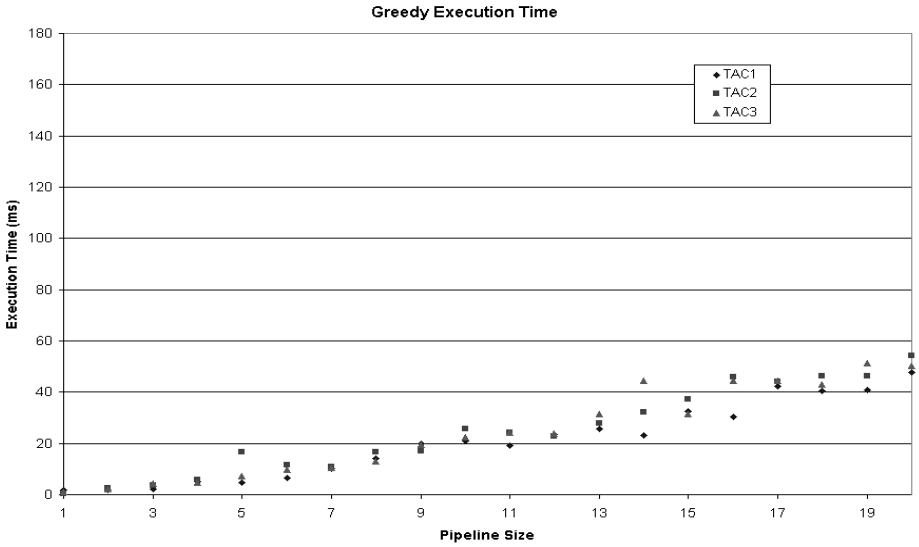


Figure 4.11: Greedy Algorithm Execution Times

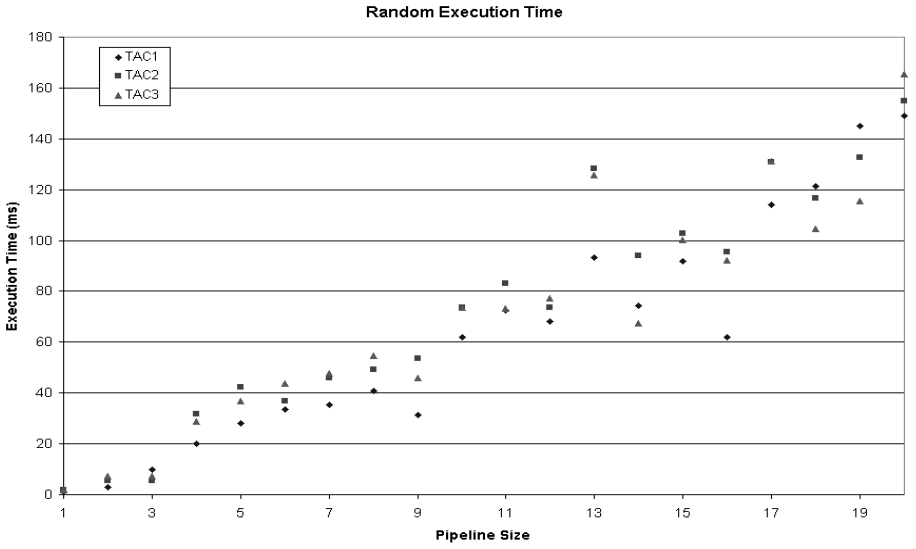


Figure 4.12: Random Algorithm Execution Times

halts prematurely, in situations where the only solutions in the neighborhood that would be accepted as points already exist on the tabu list.

### Recommended Algorithms

These results lead to our predictions of expected algorithm runtimes based on the TAC and pipeline size. First, the predictions based on the TAC are examined. The ratio of the average hardware implementation size to the device size indicates how the execution time will track the TAC1, TAC2, and TAC3 results. If the average hardware implementation size is very small compared to the size of the TAC, the execution time tracks the TAC3 results. If the average hardware implementation size is very large compared to the size of the TAC, the execution time tracks the TAC1 results. If the average hardware implementation size is roughly the same size as the TAC, then the execution time tracks the TAC2 results. Some care needs to be taken when using this ratio. Since the ratio is based on average hardware implementation size, a large standard deviation means the execution time might not track these predicted results very well. Therefore, this prediction works best when the average hardware implementation size has a small standard deviation. Currently, since all components were designed for the Wildcard and bitstream merging is not implemented, the TAC2 most closely matches the ratio of hardware implementation size to TAC. For this reason, TAC2 results were used to choose algorithms in our test system.

Table 4.8 summarizes which algorithm execution times fit into our time limits. The exhaustive search algorithm is quite a bit slower than dynamic programming, so the dynamic programming algorithm is used to find all optimal solutions. Optimal solutions are always found quickly for TAC1. There is a very limited range in which optimal solutions are found for the fixed time limit for TAC2 and TAC3; heuristic methods are used for the rest of the pipeline sizes. For TAC2 the adaptive time limit provides enough time to find optimal solutions for pipelines with 1-15 components. For TAC3 the adaptive time limit results are similar to the fixed time limit results. Once again, heuristics are used to solve all pipeline sizes that cannot be solved optimally. All of the heuristics

find solutions for pipelines of any size within both time limits. The next section analyzes how well the heuristics solve PA so recommendations can be made for which heuristic technique to use when optimal solutions cannot find a solution within the time limit.

| Algorithm Summary Per TAC and Time Limit | | | | | | |
|---|---|---|---|---|---|---|
| | Fixed | | | Adaptive | | |
| Algorithm | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| Exhaustive Search | 1-5 | 1-3 | 1-3 | 1-5 | 1-3 | 1-3 |
| Dynamic Programming | 1-20 | 1-7 | 1-6 | 1-20 | 1-15 | 1-7 |
| Greedy | 1-20 | 1-20 | 1-20 | 1-20 | 1-20 | 1-20 |
| Random | 1-20 | 1-20 | 1-20 | 1-20 | 1-20 | 1-20 |
| All SW | 1-20 | 1-20 | 1-20 | 1-20 | 1-20 | 1-20 |
| All HW | 1-20 | 1-20 | 1-20 | 1-20 | 1-20 | 1-20 |
| All Local Search Variants | 1-20 | 1-20 | 1-20 | 1-20 | 1-20 | 1-20 |

Table 4.8: Ranges of Pipeline Sizes for Each Algorithm

### 4.5.2 Solution Quality

Solution quality measures how close a solution from a heuristic technique is to the optimal solution, and is defined as a ratio of latencies:

$$\frac{L(K_{heuristic})}{L(K_{optimal})} \tag{4.6}$$

For all optimal techniques, the solution quality is always one, but there are some interesting results discussed below. As for the heuristic techniques, there are many different ways to analyze the solution quality results. For all heuristic methods, the effect of the TAC and pipeline size on solution quality is analyzed. The local search algorithms are also analyzed for the change in solution quality when moving from the initial solution to the final solution, increasing decision time, increasing neighborhood sizes, increasing the tabu list length, and changing the acceptance criteria. This section covers all of the above mentioned analyses. It should be noted that some of this analysis is difficult to do on the local search variants that use the random initial solution, so these variations will only be included in the discussion when meaningful.

| size | TAC1/TAC2 | TAC2/TAC3 | TAC1/TAC3 |
|---|---|---|---|
| 1 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 1.0077 | 1.0001 | 1.0078 |
| 3 | 1.0411 | 1.0013 | 1.0428 |
| 4 | 1.0114 | 1.0064 | 1.0184 |
| 5 | 1.3246 | 1.0141 | 1.3507 |
| 6 | 1.2912 | 1.0129 | 1.3166 |
| 7 | 1.4987 | 1.0244 | 1.5437 |
| 8 | 1.5971 | 1.0238 | 1.6451 |
| 9 | 1.6306 | 1.0270 | 1.6904 |
| 10 | 2.0298 | 1.0311 | 2.1069 |
| 11 | 1.6823 | 1.0264 | 1.7361 |
| 12 | 1.9709 | 1.0389 | 2.0715 |
| 13 | 1.9892 | 1.0339 | 2.0837 |
| 14 | 1.6538 | 1.0304 | 1.7183 |
| 15 | 2.0310 | 1.0337 | 2.1208 |
| 16 | 2.2992 | 1.0449 | 2.4327 |
| 17 | 2.6335 | 1.0512 | 2.8041 |
| 18 | 2.5957 | 1.0456 | 2.7417 |
| 19 | 2.0171 | 1.0411 | 2.1388 |
| 20 | 2.1292 | 1.0381 | 2.2361 |

Table 4.9: Comparison of Optimal Solutions

### Optimal Techniques

There are some interesting comparisons using solution quality that can be made among the optimal solutions. In particular, solution quality can be used to analyze how the optimal solutions for one device size compare to the optimal solutions for another device size. We compare the latency of the optimal solutions for TAC1 to TAC2, TAC1 to TAC3, and TAC2 to TAC3. These results are shown in Table 4.9. All three ratios are very close for pipeline sizes 1-4, which means that for small pipelines the optimal solutions tend to be very similar regardless of TAC. The TAC2/TAC3 ratio shows that the solutions from these two TACs are on average within 3% of each other. When hardware/hardware reprogramming and bitstream merging is implemented, the TAC2/TAC3 comparison should diverge more. Currently, since only one hardware implementation is used on the hardware device at a time, the techniques cannot fully use the largest TAC size. Because the TAC2 and TAC3 results are so similar, the TAC1/TAC2 and TAC1/TAC3 comparisons are similar. These two comparisons show that as the pipeline size grows, the TAC1 optimal solutions diverge from the TAC2 and

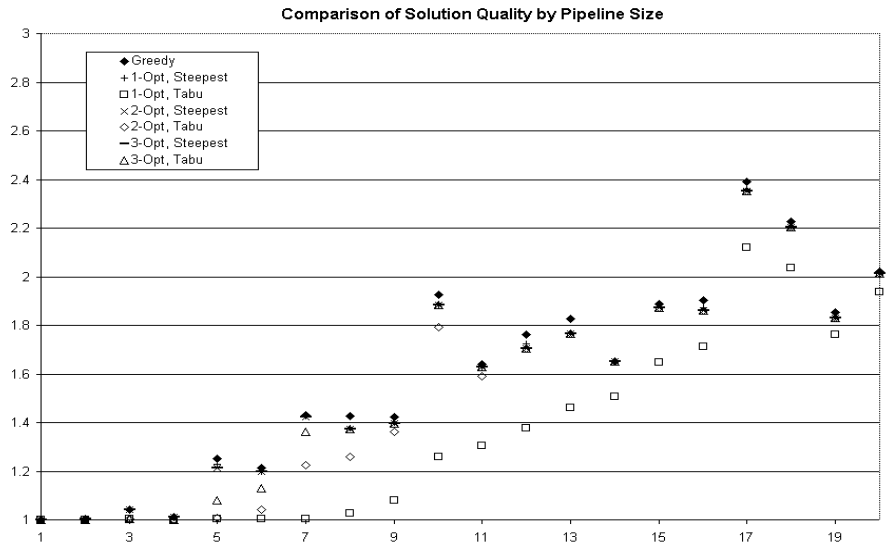


Figure 4.13: Solution Quality for Local Search with Greedy Initial Solution for the Fixed Time Limit

TAC3 optimal solutions. This is due to the fact that TAC2 and TAC3 assign more components to hardware than software as the pipeline size increases.

### Heuristic Techniques

There are several different ways to analyze the solution quality for the heuristic solutions. These include looking for changes in solution quality as TAC and pipeline sizes increase. Relevant analysis for the local search algorithms focuses on several different areas, including how neighborhood size, acceptance criterion, stopping criterion, tabu list length, and initial solution affect the solution quality. The difference in solution quality from the initial to the final solutions is another metric that indicates the quality of the local search algorithm. Also, analyzing the differences in solution quality for different stopping criteria reveals problems in the acceptance criteria. In this section we will discuss how the solution quality changes with these variables.

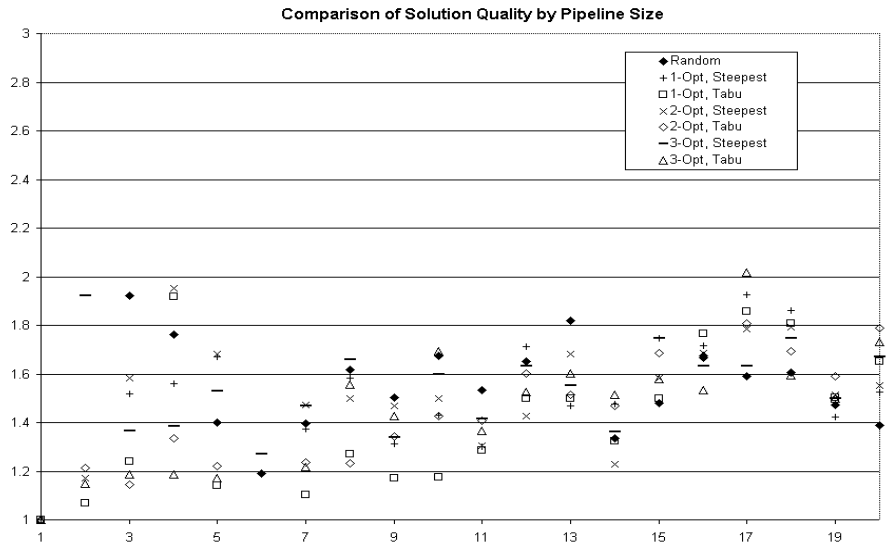


Figure 4.14: Solution Quality for Local Search with Random Initial Solution for the Fixed Time Limit

### TAC Size

The TAC size affects the solution quality results in subtle ways. In the previous section, the average optimal latency for TAC2 and TAC3 results were compared and found to be approximately the same. In fact, without bitstream merging and hardware/hardware reprogramming implemented, in depth analysis of the TAC3 results are not useful. The TAC1 results are trivial, because all algorithms can find the optimal solution. Therefore, only results that use TAC2 will be considered in the rest of the analysis.

### Pipeline Size

Solution quality is proportional to pipeline size. When analyzing the graphs in Figures 4.13 - 4.16 for the fixed time limit and Figures 4.17 - 4.20 for the adaptive time limit, one can see that the algorithms can be divided into two sets. One set has good solution quality with small pipelines and poor solution quality with

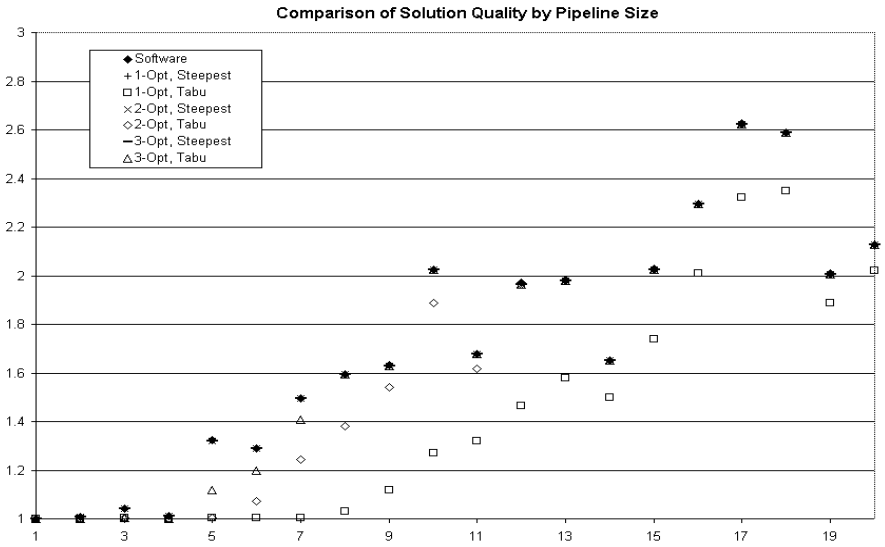


Figure 4.15: Solution Quality for Local Search with All Hardware Initial Solution for the Fixed Time Limit

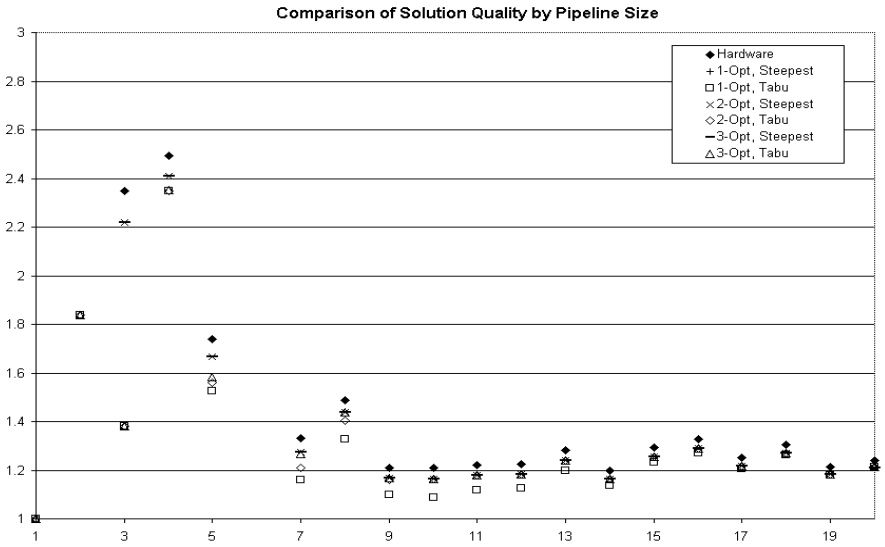


Figure 4.16: Solution Quality for Local Search with All Software Initial Solution for the Fixed Time Limit

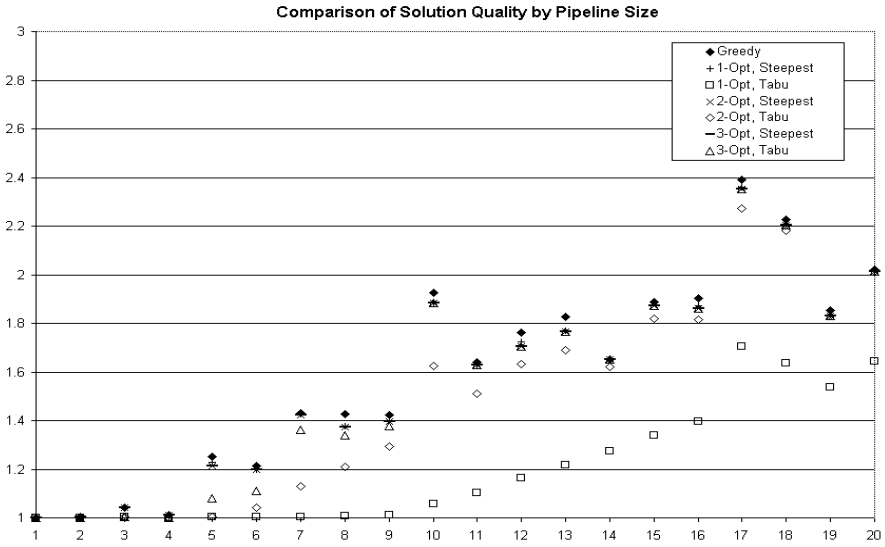


Figure 4.17: Solution Quality for Local Search with Greedy Initial Solution for the Adaptive Time Limit

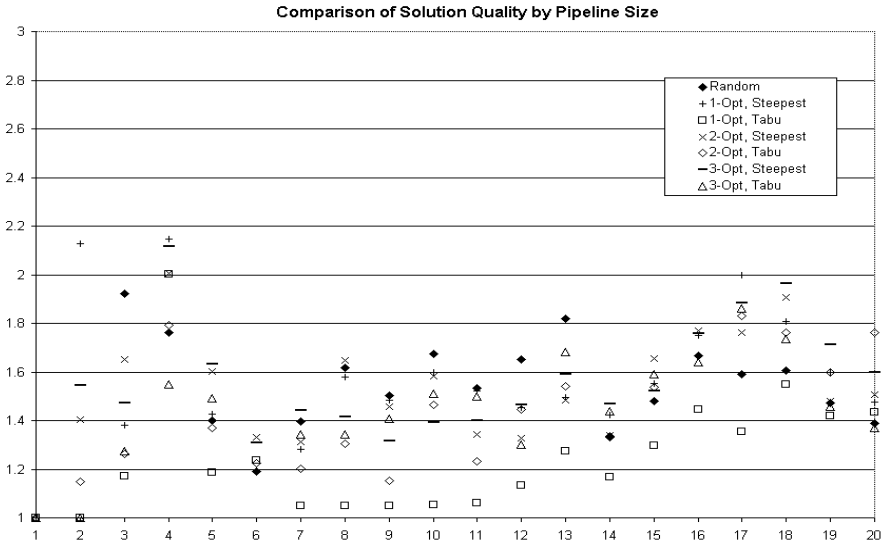


Figure 4.18: Solution Quality for Local Search with Random Initial Solution for the Adaptive Time Limit

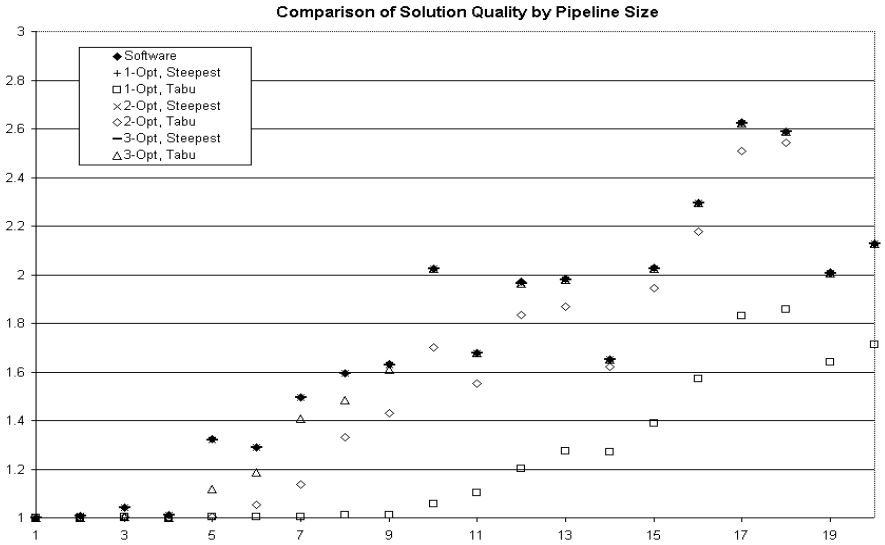


Figure 4.19: Solution Quality for Local Search with All Hardware Initial Solution for the Adaptive Time Limit

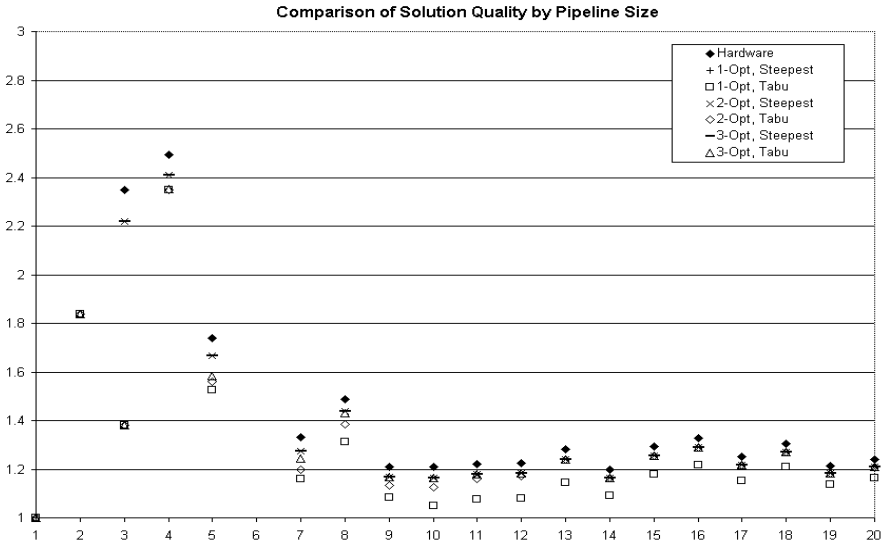


Figure 4.20: Solution Quality for Local Search with All Software Initial Solution for the Adaptive Time Limit

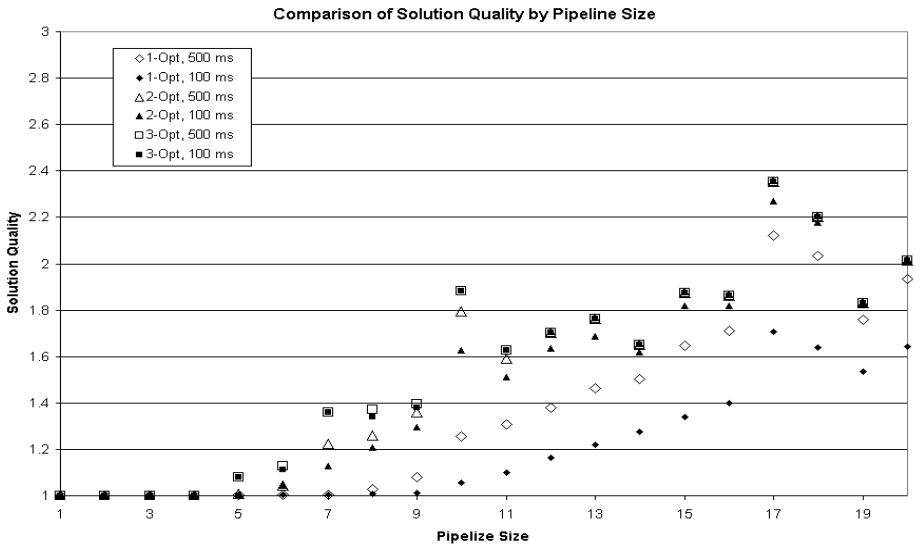


Figure 4.21: Solution Quality for Local Search with Greedy Initial Solution and Tabu Search for Both Stopping Criteria

large pipelines. The other set has good solution quality for large pipelines and poor solution quality for small pipelines. The greedy, all software, local search with a greedy initial solution, and local search with all software initial solution algorithms fall into the first set. The random, all hardware, local search with random initial solution, and local search with all hardware initial solution fall into the second set. These results are consistent across both stopping criteria.

### Comparing Initial and Final Solutions

We expect that initial solutions are improved by using local search, especially with the adaptive time limit. In Figures 4.13 - 4.16 for the fixed time limit and Figures 4.17 - 4.20 for the adaptive time limit, the solid bullets in the graphs indicates the solution quality for the initial solution. In these figures, the other bullets are the solution quality results for the local search variations that use the initial solution being graphed. The tabu local search variants have hollow

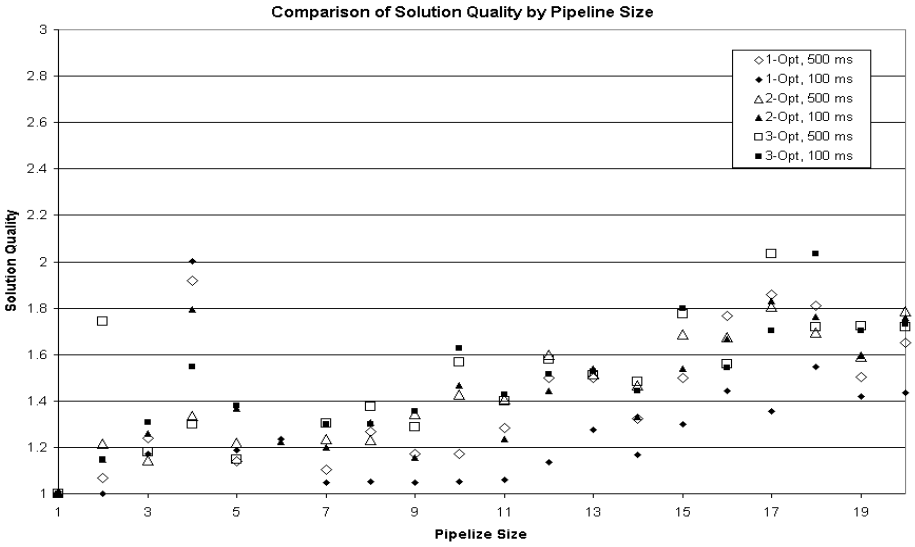


Figure 4.22: Solution Quality for Local Search with Random Initial Solution and Tabu Search for Both Stopping Criteria

bullets; the steepest descent local search variants have line bullets. When the solution quality results for the two acceptance criteria are similar, the bullets are designed so they do not occlude each other. The results show that, when the initial solution is compared to the final solution, local search finds solutions with equivalent or better solution quality than the initial solution. In reality, we only want to use local search if it returns solutions that are closer to optimal than the initial solutions. Figures 4.13 - 4.16 for the fixed time limit and Figures 4.17 - 4.20 for the adaptive time limit show the solution quality is deeply affected by the acceptance criterion, as the tabu search results were closer to the optimal solutions and the steep descent results were closer to the initial solutions. Therefore, the tabu search proves to be more useful than steepest descent.

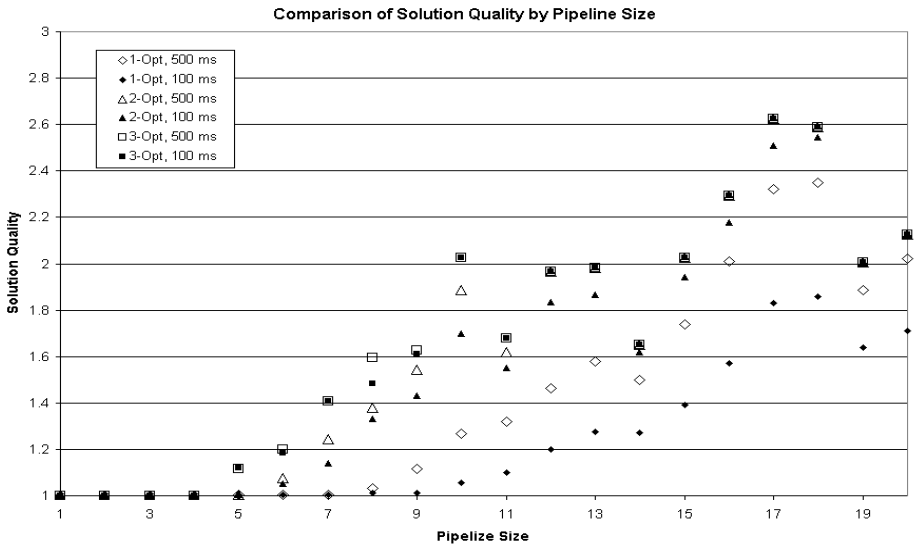


Figure 4.23: Solution Quality for Local Search with All Hardware Initial Solution and Tabu Search for Both Stopping Criteria

### Stopping and Acceptance Criteria

Figures 4.21 - 4.24 compare the solution qualities for the tabu search variants of local search based on the the stopping criteria of the fixed and adaptive time limits. Figures 4.25 - 4.28 compare the solution qualities for the steepest descent variants of local search based on the the stopping criteria of the fixed and adaptive time limits. In these graphs, the bullets for the fixed time limit results are hollow, and for the adaptive time limit results are solid so that the results do not occlude each other when they are similar. These figures show the stopping criterion is also strongly affected by the acceptance criteria. We analyze the stopping criteria based on the four ways to combine the two stopping criteria with the two acceptance criteria. Figures 4.25 - 4.28 show a trend not apparent when analyzing just the fixed time limit results. In particular, the results for the adaptive time limit with steepest descent are the same as the results for the fixed time limit with steepest descent. All of the steepest descent

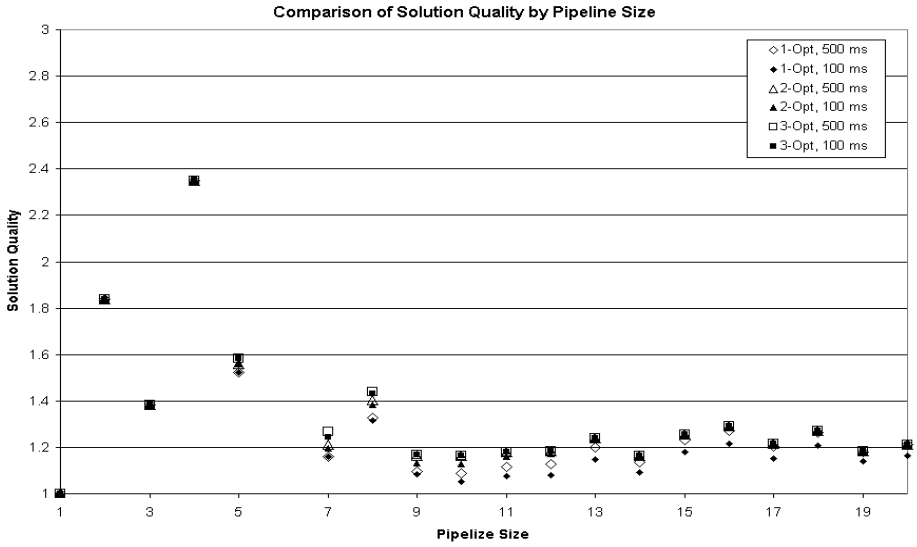


Figure 4.24: Solution Quality for Local Search with All Software Initial Solution and Tabu Search for Both Stopping Criteria

local search variants halt within the 500 ms of the fixed time limit, which means that local minima are found very quickly. This result indicates all of the initial solutions are close to or in local minima. Next, we analyze the tabu search variant with both stopping criteria shown in Figures 4.21 - 4.24. When the tabu search results using the greedy initial solution are analyzed, the adaptive time limit results are on average 4% better than the fixed time limit results when averaged over all pipeline sizes. For the all hardware initial solution the results are only 1% better and for the all software initial solution the results are 6% better when averaged over all pipeline sizes.

### Neighborhood Size

As discussed earlier, increasing the neighborhood size may find solutions closer to optimal than a smaller neighborhood would, if there is enough time to iterate the *improve* function several times. The problem is that even with the adaptive

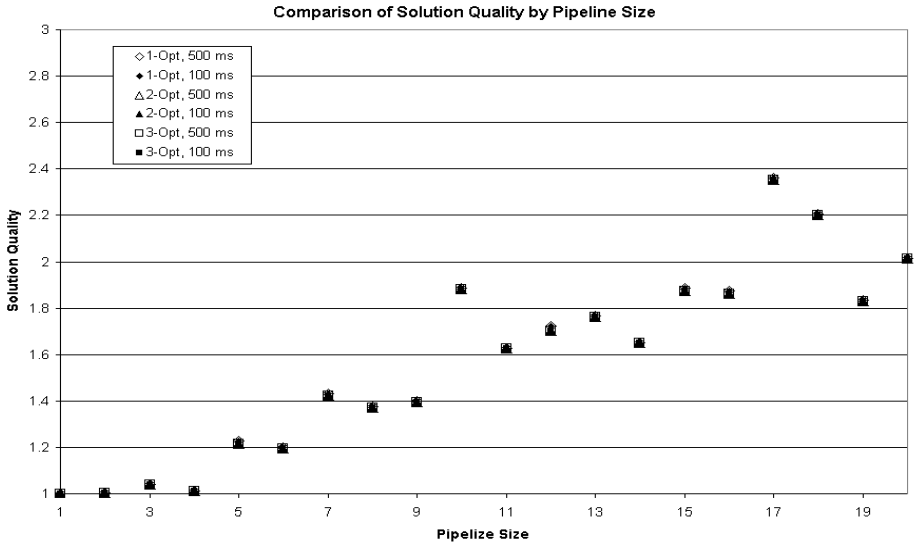


Figure 4.25: Solution Quality for Local Search with Greedy Initial Solution and Steepest Descent for Both Stopping Criteria

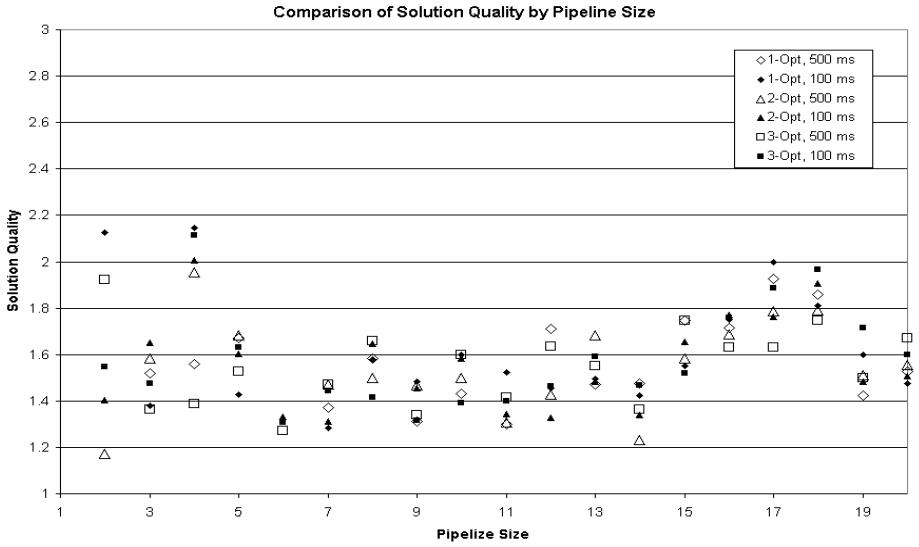


Figure 4.26: Solution Quality for Local Search with Random Initial Solution and Steepest Descent for Both Stopping Criteria

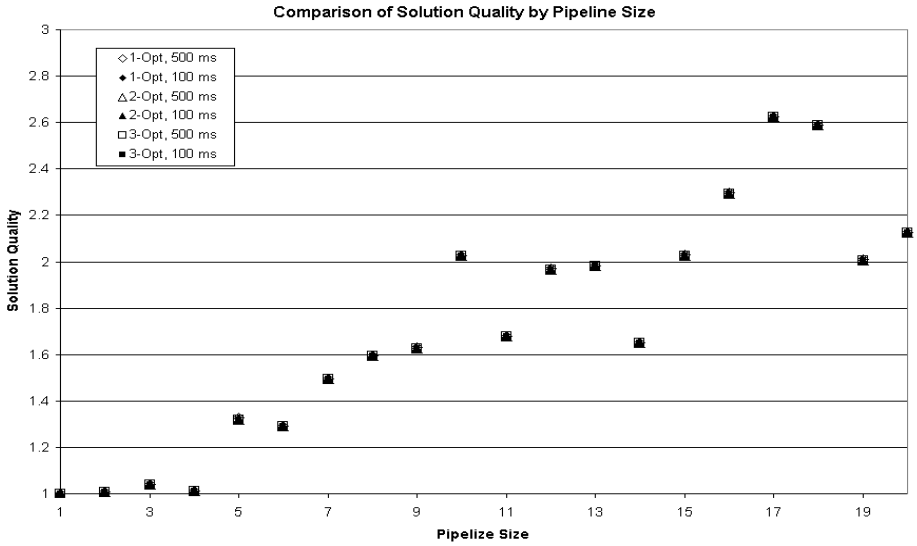


Figure 4.27: Solution Quality for Local Search with All Software Initial Solution and Steepest Descent for Both Stopping Criteria

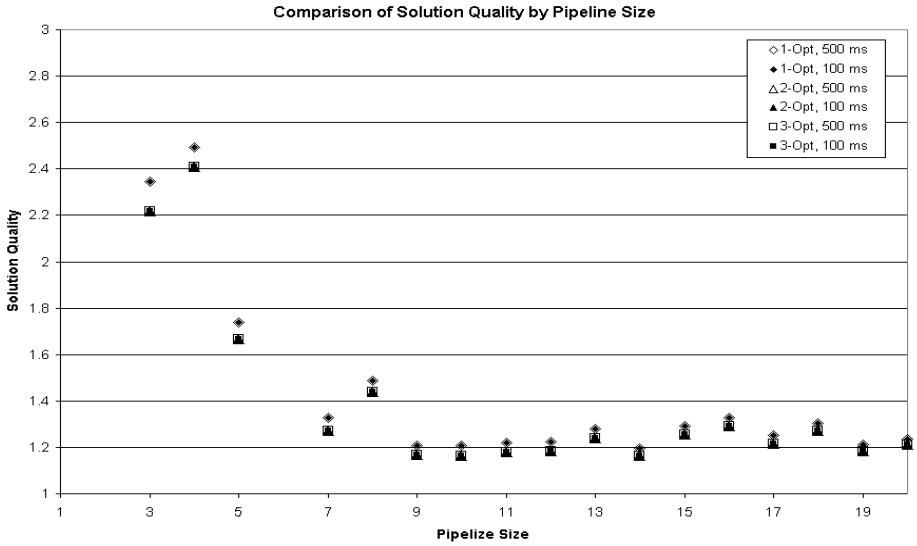


Figure 4.28: Solution Quality for Local Search with All Hardware Initial Solution and Steepest Descent for Both Stopping Criteria

time limit there is not enough time to iterate the *improve* function several times. This is especially a problem for larger pipelines. Therefore, as the value for $k$ in the $k$-Opt neighborhood increases, the solution quality worsens. Figures 4.13 - 4.16 for the fixed time limit and Figures 4.17 - 4.20 for the adaptive time limit show that the 1-Opt neighborhood performs much better than the 2-Opt and 3-Opt neighborhoods. This is because there are more iterations with the 1-Opt neighborhood.

**Acceptance Criteria**

| Initial Solution | Fixed | Adaptive |
|---|---|---|
| Greedy | 7% | 13% |
| All Software | 10% | 16% |
| All Hardware | 87% | 88% |

Table 4.10: Percent Improvement from Using Tabu Search Instead of Steepest Descent for Both Time Limits

Next, the results are analyzed for changes in solution quality between acceptance criteria. In Figures 4.13 - 4.16 for the fixed time limit and Figures 4.17 - 4.20 for the adaptive time limit, we can see that tabu search's solution quality is always better than or equivalent to steepest descent's solution quality. Table 4.10 shows the percentage improvement of the tabu search results over the steepest descent results. The all hardware initial solution quality is so much better than the other two because the steepest descent solution quality for small pipelines is poor. For the local search variations that use the all hardware and all software initial solutions and large pipelines there is little difference between using tabu search or steepest descent except when using 2-Opt and 3-Opt neighborhoods.

### Tabu List Length

All of these results are based on a tabu list of length seven. Increasing the length of the tabu list could potentially produce better solutions, but the length would have to be increased significantly. The number of members in a neighborhood is a function of the number of components in a pipeline, the number of implementations each component in a pipeline has, and the size of the neighborhood. When there are only two possible implementations for each component, a 3-stage pipeline has a 1-Opt neighborhood with three members, a 2-Opt neighborhood with six members, and a 3-Opt neighborhood with seven members. In the problem space where each component has one software and one hardware implementation, a tabu list of length seven contains most or all of the neighborhood. When there are seven possible implementations for each component, a 3-stage pipeline has a 1-Opt neighborhood with 18 members, a 2-Opt neighborhood with 126 members, and a 3-Opt neighborhood with 342 members. In this problem space, the tabu list of size seven contains only a fraction of the neighborhood. Therefore, to make a difference the tabu list would have be increased so that more of the neighborhood is stored in memory, which is not practical. In the future more sophisticated methods for implementing tabu search will be explored to avoid computation cycles.

| Device Size | Dynamic Programming | 1-Opt Tabu Search with Greedy | 1-Opt Tabu Search with All Hardware |
|---|---|---|---|
| TAC1 | 1-20 | – | – |
| TAC2 | 1-7 | 8,9 | 10-20 |
| TAC3 | 1-6 | 7-13 | 14-20 |

Table 4.11: Recommended Algorithms for the Fixed Time Limit

| Device Size | Dynamic Programming | 1-Opt Tabu Search with Greedy | 1-Opt Tabu Search with All Hardware |
|---|---|---|---|
| TAC1 | 1-20 | – | – |
| TAC2 | 1-15 | – | 16-20 |
| TAC3 | 1-7 | 8-19 | 20 |

Table 4.12: Recommended Algorithms for the Adaptive Time Limit

| Test Number | Pipeline Size | Pipeline |
|---|---|---|
| 1 | 1 | SOURCE → hist1c3b16bin → SINK |
| 2 | 5 | SOURCE → mf → emOr → ed → emAnd → ed → SINK |
| 3 | 10 | SOURCE → emAnd → mf → mf → emAnd → mf → mf → emAnd → ed → ed → hist4c4b1bin → SINK |
| 4 | 15 | SOURCE → emOr → mf → ed → ed → ed → minFilter → maxFilter → emAnd → emOr → minFilter → mf → emOr → maxFilter → mf → hist4c4b1bin → SINK |
| 5 | 20 | SOURCE → maxFilter → emAnd → emAnd → emAnd → minFilter → emAnd → ed → ed → minFilter → ed → minFilter → emAnd → minFilter → ed → maxFilter → emOr → emOr → maxFilter → maxFilter → hist4c8b16bin → SINK |

Table 4.13: Test Pipelines for Algorithm Comparisons

| Test Number | Dynamic Prog | 1-Opt Tabu Search with All Hardware | 1-Opt Tabu Search with Greedy | 2-Opt Tabu Search with Greedy | 3-Opt Tabu Search with Greedy | Greedy |
|---|---|---|---|---|---|---|
| 1 | 295 | 295 | 295 | 295 | 295 | 295 |
| 2 | 3328 | 3328 | 3328 | 3328 | 3328 | 6076 |
| 3 | N/R | 4613 | 4473 | 11607 | 13287 | 13287 |
| 4 | N/R | 6661 | 12028 | 17275 | 17275 | 17275 |
| 5 | N/R | 8317 | 15198 | 16792 | 16792 | 16792 |

Table 4.14: Comparison of Predicted Latencies for Selected Algorithms for the Fixed Time Limit Where N/R Means No Result

| Test Number | Dynamic Prog | 1-Opt Tabu Search with All Hardware | 1-Opt Tabu Search with Greedy | 2-Opt Tabu Search with Greedy | 3-Opt Tabu Search with Greedy | Greedy |
|---|---|---|---|---|---|---|
| 1 | 295 | 295 | 295 | 295 | 295 | 295 |
| 2 | 3328 | 3328 | 3328 | 3328 | 3328 | 6076 |
| 3 | 4473 | 4500 | 4473 | 9585 | 11607 | 13287 |
| 4 | 5531 | 6028 | 8070 | 15595 | 17275 | 17275 |
| 5 | N/R | 8038 | 11240 | 16066 | 16792 | 16792 |

Table 4.15: Comparison of Predicted Latencies for Selected Algorithms for the Adaptive Time Limit Where N/R Means No Result

## 4.6 Recommended Algorithms

In summary, we have found trends in local search that indicate some variations work better than others. First, the steepest descent acceptance criterion does not work well with our initial solutions, since most of them are in local minima. Second, increasing neighborhood size does not give better results. Therefore, the recommended algorithms include tabu search with a 1-Opt neighborhood. The recommended algorithms for the fixed time limit are listed in Table 4.11 and for the adaptive time limit are listed in Table 4.12.

To prove that these recommended algorithms work well within their defined ranges we compared the solutions for a few algorithms with five test pipelines of different sizes for TAC2. The selected algorithms include the three algorithms used to solve the pipeline assignment problem for both time limits for a device size of TAC2: dynamic programming, 1-Opt tabu search with greedy initial solution, and 1-Opt tabu search with all hardware initial solution. Three other algorithms were included for comparison: 2-Opt tabu search with greedy initial solution, 3-Opt tabu search with greedy initial solution, and greedy. The pipelines are listed in Table 4.13. The dynamic programming algorithm was tested for the ranges in which it is recommended for both time limits. The tabu search variations were tested using both time limits as stopping criteria. The greedy algorithm was run to completion. The results of running these six algorithms with the five test pipelines are shown in Tables 4.14 and 4.15. For test pipelines that were not solved by dynamic programming the abbreviation N/R is used to indicate that there is no result.

The results in Tables 4.14 and 4.15 show that the recommended algorithms find the shortest latency solution over the compared algorithms for the pertinent ranges. For the two smallest test pipelines, the local search variants were able to find the optimal solution. The recommended algorithm for 10- to 20-stage pipelines using the fixed time limit is a local search variant using a 1-Opt neighborhood, a tabu acceptance criterion, and an all hardware initial solution. In Table 4.14 for the three test pipelines in the 10- to 20-stage pipeline range

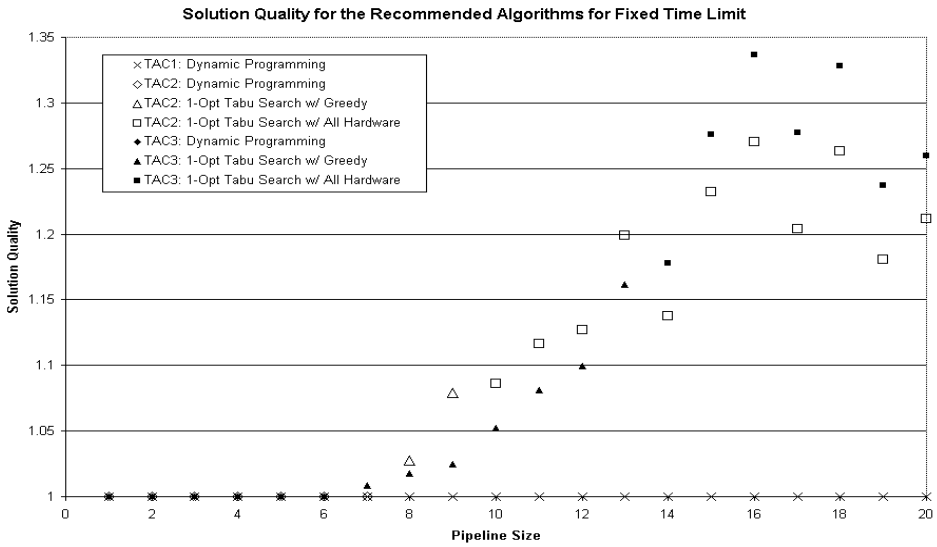


Figure 4.29: Solution Quality for the Recommended Algorithms for the Fixed Time Limited

the recommended algorithm finds the shortest latency solution. The same local search variant is recommended for 20-stage pipelines using an adaptive time limit and Table 4.15 shows that this algorithm finds the shortest latency solution.

Figures 4.29 and 4.30 show the solution quality for the recommended algorithms for all three device sizes. These graphs summarize the results of using different algorithms based on pipeline size. In both graphs the solution quality for TAC1 is denoted with an X, TAC2 uses hollow bullets, and TAC3 uses solid bullets so the results for one device size do not occlude the results for the other two. In both graphs TAC2 and TAC3 using dynamic programming are shown with diamond bullets, 1-Opt tabu search with greedy initial solution uses triangle bullets and 1-Opt tabu search with all hardware initial solution uses square bullets so that the algorithms can be distinguished. Both figures show that in the regions solved by heuristics the solution quality is generally good while still returning results quickly. For Figure 4.29, which shows the solution quality for the recommended algorithms for the fixed time limit, we can see that

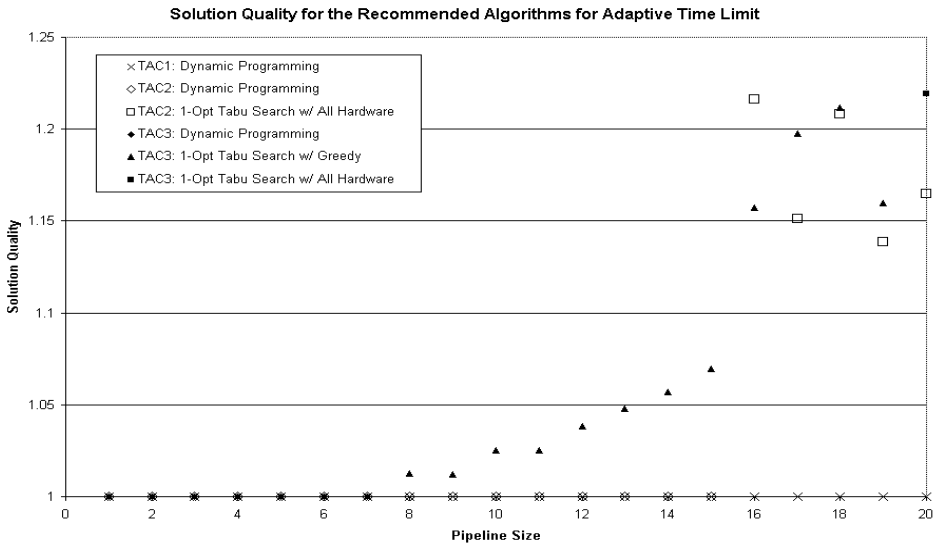


Figure 4.30: Solution Quality for the Recommended Algorithms for the Adaptive Time Limited

on average the worst solution quality for TAC1 is 1, for TAC2 is 1.27, and for TAC3 is 1.34. For all three device sizes, pipelines with 15 or fewer components can be solved with a solution quality between 1-1.2 when using a fixed time limit. For Figure 4.30, which shows the solution quality for the recommended algorithms for the adaptive time limit, we can see that on average the worst solution quality for TAC1 is 1, and for TAC2 and TAC3 is 1.22. For all three device sizes, pipelines with 15 or fewer components can be solved with a solution quality between 1-1.07 when using an adaptive time limit.

## 4.7 Conclusions

This chapter covers the pipeline assignment problem definition and techniques for solving it. Three optimal algorithms for solving the pipeline assignment problem were presented: exhaustive, integer linear programming, and dynamic programming. Several heuristic algorithms were also discussed: greedy, random,

and several variations of local search. The chapter concludes with a comparison of the algorithms' runtimes and solution quality so that algorithms and a recommendation for use in the runtime environment. These algorithms were used in our runtime environment, which is discussed in the next chapter.

# Chapter 5

# Dynamo

## 5.1 Introduction

The Dynamo system provides a flexible hardware/software runtime environment for image processing applications. Dynamo is a Java-based runtime partitioning, interface synthesis and execution system. An image analyst uses the Dynamo system by specifying the pipeline and an image (or images) to be processed. The pipeline is composed of components defined in the image processing Basic Library of Components (ipBLOC). From this specification, Dynamo finds the most efficient combination of hardware and software component implementations to minimize pipeline runtime (*pipeline assignment*), generates the source code (*pipeline compilation*), processes the input image using the generated pipeline (*pipeline execution*), and returns the result to the analyst.

The goals of the Dynamo pipeline image processing system are:

- to allow an image analyst to focus solely on pipeline selection,
- to dynamically create image processing pipelines,
- to make efficient use of software and FPGA hardware, and
- to build pipelines that minimize the image processing time for the entire pipeline.

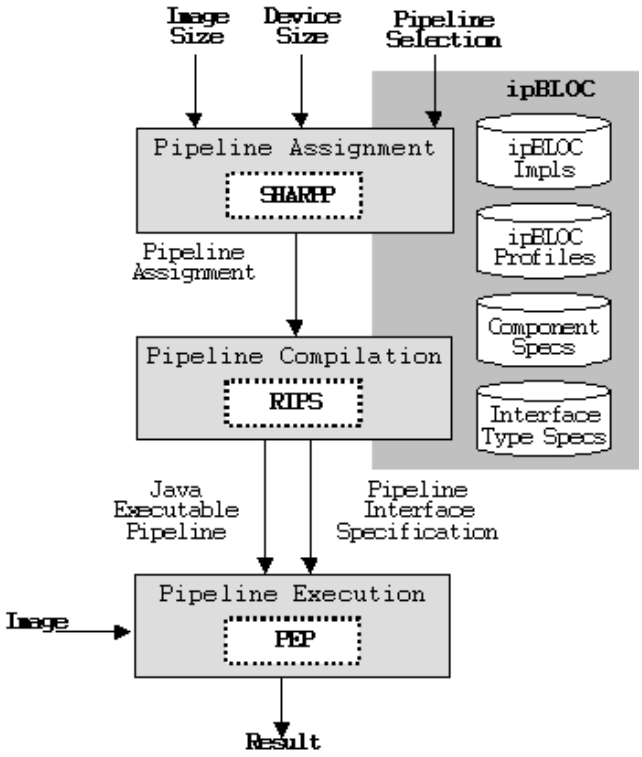


Figure 5.1: System Overview

To meet these goals, Dynamo performs pipeline assignment, pipeline compilation, and pipeline execution dynamically and automatically. As input, the analyst provides a pipeline specification and the image to be processed. Dynamo's design has three major subsystems which implement pipeline assignment, compilation and execution. These subsystems interact with Dynamo's predefined libraries including the ipBLOC. Figure 5.1 shows an overview of the Dynamo system and also gives an overview of how data flows through the system.

The Dynamo subsystem responsible for solving the pipeline assignment problem (PA) as specified in Chapter 4 is named the Software/HArdware Runtime Procedural Partitioning (SHARPP). SHARPP solves PA for the chosen pipeline

selection and image at runtime with a set time limit. SHARPP uses either optimal or heuristic methods depending on the pipeline size. SHARPP's output is an annotated solution that defines the assigned implementations.

Dynamo's Runtime Interfacing for Pipeline Synthesis (RIPS) subsystem performs interface synthesis. The RIPS input is the annotated solution from the SHARPP. RIPS parses the pipeline assignment and combines the component implementations to create executable pipelines. The pipeline code includes code to invoke the implementations and extra processing methods. The pipeline assignments are constructed at runtime because pre-constructing them for all possible combinations of components and component implementations is not practical. RIPS outputs a named executable pipeline.

Dynamo's execution subsystem uses the Packet Exchange Platform (PEP) as a communication layer to execute the compiled pipeline. The PEP connects the RIPS generated hardware/software solution to the runtime environment [75].

Figure 5.1 also shows pre-defined libraries shared by the SHARPP, RIPS, and PEP subsystems. The ipBLOC maintains implementation, component, and interface specifications which encapsulate information about them. Dynamo was architected so that subsystems are specification driven in order to make the system easily extensible. New components are added to the ipBLOC by importing the specification and implementation code for the component into the library.

The Dynamo runtime environment is the focus of this chapter. The chapter begins with a description of software engineering techniques used to create the runtime environment. The abstract communication layer is presented in Section 5.3. Sections 5.4, 5.5, and 5.6 discuss the three subsystems that make up Dynamo. Section 5.7 describes how the runtime environment can be used to do co-verification and co-simulation. The chapter ends with experimental results.

## 5.2 Module-based System Design

The goal of our research was to create fast, flexible image processing applications that are responsive to the image analyst's requests. The runtime environment was designed using component-based design techniques [78] so that it could respond easily to what the user wants. Component-based design breaks systems into modules that are reusable amongst many applications.[1] With a module-based design, the runtime environment is broken into several types of specialized, self-contained modules for displaying output, gathering user input, solving PA, and building pipelines. Pipelines, components, and implementations create natural modules. This section gives an overview of the modules that make up the runtime environment.

A module-based runtime environment allows the pipeline, components, and implementations to be dynamically bound. We would like to separate the pipeline from the runtime environment, the component from the pipeline, and the implementation from the component. The runtime environment has been divided into independent modules which are grouped into broad categories based on input/output interface and tasks performed (*behavior*). Modules are classified into these categories: display method, algorithm for solving the pipeline assignment problem, interface synthesis, user input method, pipeline, interface, component, and implementation. The extra processing methods, such as image padding, needed to implement a pipeline solution are categorized as component modules.

Each module in a category has the same defined input/output interface and behavior. For all modules in a given category, these definitions must be met. The input/output interface for a category controls what data is passed into or out of a module. The behavior for a category depends on the category. For example, the display modules must show the pipeline's output, but the format and method used to display the results depends on the module. The category

[1] Since we use components as building blocks in the pipeline, we will use the term *module* when discussing the runtime environment's architecture.

for the components modules is different than other categories. Components are defined with a standardized interface in Chapter 3 and these interfaces can be used as subcategories of the component modules. The pipeline modules interact with the component modules by using their interface modules.

## 5.3 Abstract Communication Layer

To make late or dynamic binding of the pipeline modules feasible in the runtime environment's execution subsystem, a layer of abstraction is required between pipeline components so that they do not interface directly together. This abstraction allows modules to be defined independently so that each module's code and interface are separated from all other modules. A communication layer is interposed between the modules so that modules pass data solely through this communication layer. Once this separation is supported, pipelines, components and implementations can change their code as long as the behavior and the input/output interface remain the same. The execution subsystem controls which components are used and how components interact with each other. Therefore, the execution subsystem can change which pipeline it is using, and the components in a pipeline assignment can be assigned to different implementations. This section describes the abstract communication layer we created for the execution subsystem.

The abstract communication layer, called the *Packet Exchange Platform (PEP)*, is built on the idea of producer/consumer communication. Each component has a communication agent as shown in Figure 5.2. An agent can communicate with its module and another agent. The agent-to-module communication focuses on the data that needs to be passed between modules, and on the agent receiving the correct data at the correct time. The component determines whether the agent is waiting to receive data, sending data, or idle. The agent-to-agent communication passes the data between two modules, so that one module does not directly interface with another module. Once an agent receives data it passes the data to the consuming agent.

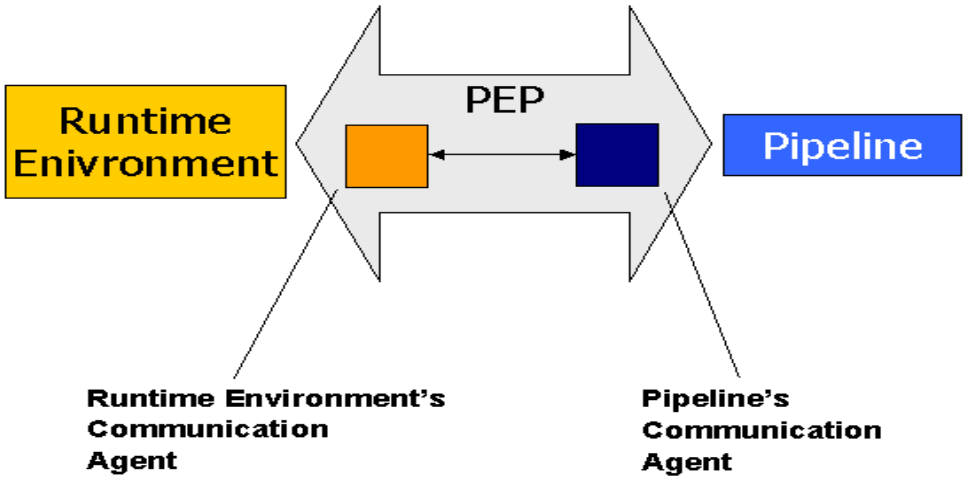


Figure 5.2: The Packet Exchange Platform

To make this abstraction work, the communication layer must have the following properties:

**Abstract:** The behavior of each module is hidden so the module's interface remains the same regardless of module changes.

**Flexible:** Late or dynamic binding to modules is supported to allow the execution subsystem to change which pipeline instance, component or implementation is being used.

**Reliable:** Agent-to-module and agent-to-agent communication must ensure there is no data loss, duplication or corruption.

The communication layer hides the specific mechanism used to transport the data from the producers and consumers of the data. The PEP implements the communication layer as shared memory with mutual exclusion via monitor locks to protect against concurrent access. In the shared memory model, the communication layer is implemented as the shared data area within a single process on a single host where individual threads represent the component's execution

space. Threads communicate with each other via a shared data structure. This system is modeled after consumer/producer communication, where one component produces the data and another component consumes the data. When the producer is ready to pass data to the consumer, the producer passes the data to its agent so that the producer's agent can put the data into the PEP. If the consumer's agent is waiting for data from the producer, the agent will be notified that data is in the PEP. The consumer's agent then retrieves the data for the consumer. The retrieval process also removes the data from the shared memory. If the producer's agent is ready to send more data to the consumer, once the PEP's shared memory is empty it is notified that the shared memory is ready for more data. In the case of hardware implementations, the software controller for the hardware device also controls the producer and consumer agents for the hardware implementation. Therefore, the hardware device has no knowledge about the PEP and data from the PEP is written to the hardware device using standard software controls. This model is easily extended in the future to allow agent-to-agent communication to occur across a network so that modules can be distributed across many machines in a network of workstations.

## 5.4 Pipeline Assignment - SHARPP

Dynamo's SHARPP subsystem is responsible for solving the pipeline assignment problem. There are three options for optimally solving the pipeline assignment problem: exhaustive search, integer linear programming, and dynamic programming. Dynamic programming performs the fastest of these three algorithms, but still cannot process the entire range of pipeline sizes we are interested in within the time limits. Consequently, SHARPP uses heuristics for larger pipelines when optimal techniques do not finish within the time limit. There are three heuristic techniques implemented in Dynamo: greedy, random, and local search. The SHARPP dynamically chooses which algorithm to use at runtime based on the size of the pipeline.

Results for solving the pipeline assignment problem were presented in Chapter 4. Experiments used two time limits: fixed (500 ms) and adaptive (100 ms per pipeline stage). The experiments also studied three TACs: 0, the Wildcard device size, and the maximum integer value. In the Dynamo system we consider only the TAC2 results for the two time limits. The recommended algorithms for the fixed time limit are: dynamic programming for 1- to 7-stage pipelines; local search with a greedy initial solution, 1-Opt neighborhood, and tabu search for 8- and 9-stage pipelines; and local search with an all hardware initial solution, 1-Opt neighborhood, and tabu search for 10- to 20-stage pipelines. The recommended algorithms for the adaptive time limit are: dynamic programming for 1- to 15-stage pipelines; and local search with an all hardware initial solution, 1-Opt neighborhood, and tabu search for 16- to 20-stage pipelines.

Internally, SHARPP uses a directed graph to represent the pipeline, where each node is a pipeline component, and each edge is connected such that pipeline order is preserved. When solving the pipeline assignment problem, SHARPP annotates each node in the graph with the implementation choice (hardware or software) for the component and each edge with the processing needed to fulfill the coupling rules. Nodes that are assigned to hardware are also annotated with the name of the bitstream that should be used. Examples of the SHARPP output are shown in Table 5.1 and discussed further in Section 5.8.

## 5.5 Pipeline Compilation - RIPS

Dynamo's RIPS subsystem takes the annotated pipeline assignment from the SHARPP as input. The RIPS produces an executable pipeline and a pipeline interface specification document from the input. The pipeline specification, like the other specification documents, encapsulates information about the executable pipeline such as the solution name, pipeline assignment, image size, inputs required from the analyst to run the pipeline, and how to display the final result. The RIPS stores the executable pipeline and pipeline specification in the pipeline library, then returns the solution name to the user so it can be

compiled before execution. This section explains how the RIPS transforms the pipeline assignment into an executable pipeline so data integrity is maintained during execution.

An executable pipeline is created by traversing the graphical representation of the pipeline assignment from source to sink node and translating all the implementation and extra processing methods into Java source code. Each implementation and extra processing method can be invoked using Java's reflection technology. While the pipeline assignment graph is traversed, the reflection commands to invoke each method are written into a source file that can be compiled and executed to process an image.

The RIPS must also make certain that the executable pipeline can interact with the modules in the execution subsystem. In particular, the pipeline needs the user interface and display modules. Once the executable pipeline reads the pipeline interface specification document, it starts the user interface module to the input data the document has specified. The user interface module we use is a dialog box that lists the variable names with text boxes for the user to enter the data. A text-based user interface module could also be used. The user interface module queries the user for data, and then verifies the data's types match what the pipeline interface specification defines them as. The type checked data are returned to the executable pipeline at which point the pipeline begins processing. Once the executable pipeline finishes processing the image a display module presents the results to the user. There are many different display modules; the pipeline interface specification document determines which one is used. There are display modules that present the input and output images as well as ones for returning graphical histogram bargraphs and textual histogram tables.

Data and data flow is an important part of the executable pipeline. First, the executable pipeline needs to receive the appropriate data from the user. A union of all of the pipeline's components' interfaces determines all of the unique inputs that a user needs to supply to run the pipeline. Figure 5.3 shows an example where four variables are needed for execution: image, height, width, and

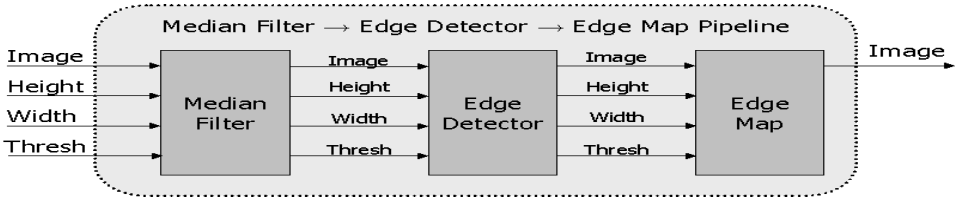


Figure 5.3: Data Flow through an Executable Pipeline

threshold. The user interface module queries the user for input variables and then type checks the data before passing the data to the executable pipeline. The executable pipeline passes all of the data to the first component. Each component passes all of the user's input data from component to component whether the data is needed or not. Any data that changes during processing, most notably the image, is changed for the rest of the components in the executable pipeline. The first and last components in the pipeline are the only components that have to directly communicate data with the pipeline module's interface.

## 5.6 Pipeline Execution - PEP

Dynamo's final phase runs the pipeline and outputs the results. This subsystem was built with the Packet Exchange Platform (PEP). Dynamo spawns threads to process the input image using the executable pipeline built by the RIPS. Data is passed between threads using the PEP. The hardware implementations have software controllers that coordinate the PEP agents and the FPGA board. The executable pipeline finds and invokes all of the necessary implementations and extra processing methods to process the image.

Pipelines are run in three phases: pre-processing, processing, and post-processing. Pre-processing converts the image file into a data array, parses the pipeline specification, prompts the analyst for any pipeline input variables needed such as image name, and validates analyst inputs. Once the pre-processing

steps are done, the pipeline is executed. The post-processing phase outputs the results to a display module using the output format in the pipeline specification.

## 5.7 Co-simulation and Co-verification

One advantage of the module-based runtime environment is that it can be used to perform co-simulation and co-verification. Due to the modular nature of the runtime environment, Dynamo switches between running a component's implementations in software and hardware easily. A hardware device can be replaced with a hardware simulator so that new component implementations can be tested in a pipeline with real image data. The runtime environment can also be used to do co-verification of hardware and software implementations to make certain new implementations work with the older hardware and software implementations before the analyst attempts to use them. This section covers how co-simulation and co-verification can be done using Dynamo.

First, we present how Dynamo can be used as a co-simulation tool. All hardware implementations were built in an experimental Java-based hardware description language (HDL), called JHDL [47, 6]. JHDL includes a built-in simulation environment accessed by executing a hardware circuit's testbench. The JHDL testbench offers a great deal of flexibility in the way one tests a circuit since it is an executable Java program. Since the testbench is software, it can be treated like another type of implementation executed in a pipeline. This way, a pipeline can be simulated while the hardware implementation is being completed.

Co-verification is also a necessary aspect of testing new implementations. Since all implementations of a component must have the same outputs, they can be tested against each other. The runtime environment has the ability to run two pipelines in co-verification mode. Once the execution is completed, the results from the two pipelines are compared to see if the results match. The display modules used in co-verification mode show the output of both pipelines as well as a comparison of the results. For example, if two median filter implementations

are being tested, then a pipeline is built for each implementation. The two pipelines are executed in the runtime environment and the output images are compared. The compared results for this case is an image of the same dimensions as the output image, where every pixel that does not match is assigned to black and every pixel that does match is assigned to white. The co-verification display module shows this comparison image with the two output images so that the designer can assess the cases where the output images do not match.

## 5.8 Experiments

This section describes two experiments which demonstrate Dynamo in action. All tests were run on our target architecture of general purpose processor (GPP) and one FPGA board. The GPP is an Intel Pentium 3 chip with a 650 Mhz clock. The FPGA is the Wildcard described in Chapter 3. All of these tests involve measuring pipeline latency, which were measured in the Eclipsecolorer profiling tool [21] for the Eclipse [20] Java integrated development environment (IDE). The first experiment considers all possible combinations of a two component pipeline, and shows that Dynamo can build and execute them. The second experiment compares predicted and actual results for several test pipelines. This section concludes with a discussion of error in the pipeline assignment's latency calculation.

### 5.8.1 Two Component Pipeline Example

This example uses the two component pipeline: median filter → histogram. The histogram uses three channels (red, green, and blue) of the image, the upper four bits of each channel and 16 bins per channel. There are three different hardware implementations of the median filter: one without extra processing elements ($HW_1$), one with only padding ($HW_2$), and one with only unpadding ($HW_3$). There is no median filter implementation with both pad and unpad methods as the circuit is too large. The hardware implementation with the fix

| | Annotated Solution |
|---|---|
| SW/SW | (SOURCE:sw) → [none] → (mf:sw) → [none] → (histogram:sw) → [none] → (SINK:sw) |
| SW/HW | (SOURCE:sw) → [none] → (mf:sw) → [comm:in + reprog] → (histogram:hw:histogramRight.x86) → [comm:out] → (SINK:sw) |
| $HW_1$/SW | (SOURCE:sw) → [Pad:in + comm:in + reprog] → (mf:hw:mfRight.x86) → [Unpad:out+ comm:out] → (histogram:sw) → [none] → (SINK:sw) |
| $HW_2$/SW | (SOURCE:sw) → [comm:in + reprog] → (mf:hw:PadmfRight.x86) → [Unpad:out+ comm:out] → (histogram:sw) → [none] → (SINK:sw) |
| $HW_3$/SW | (SOURCE:sw) → [Pad:in + comm:in + reprog] → (mf:hw:mfUnpadRight.x86) → [comm:out] → (histogram:sw) → [none] → (SINK:sw) |
| $HW_1$/HW | (SOURCE:sw) → [Pad:in + comm:in + reprog] → (mf:hw:mfRight.x86) → [Unpad:out + comm:out + comm:in + reprog] → (histogram:hw:histogramRight.x86) → [comm:out] → (SINK:sw) |
| $HW_2$/HW | (SOURCE:sw) → [comm:in + reprog] → (mf:hw:PadmfRight.x86) → [Unpad:out + comm:out + comm:in + reprog] → (histogram:hw:histogramRight.x86) → [comm:out] → (SINK:sw) |

Table 5.1: The Solutions to Median Filter → Histogram

image padding method is not used in this pipeline assignment because that implementation cannot combine with histogram. All possible pipeline assignments in the problem space for this pipeline, including all extra processing, are shown in Table 5.1. The SHARPP automatically adds source and sink components to model communication and extra processing needed at the beginning and end of the pipeline. The component assignments are shown within parentheses with ":sw" indicating software and ":hw" indicating hardware. The bitstream name follows the ":hw" annotation.[2] The processing for the interfaces is in square brackets. These annotated solutions are exactly what the SHARPP outputs to the RIPS.

[2]Note that bitstream names have the form *bitstreamName.x86*.

All solutions in Table 5.1 meet the TAC requirement. There are two hardware/hardware solutions, because removing the image padding in software inserts the necessary software/hardware boundary which permits reprogramming. The $HW_3$/HW solution does not exist as the area constraint is not met without allowing hardware/hardware boundary reprogramming, which is not currently supported.

| Pipe Asst | Predicted Latency with No Overhead | Predicted Latency with Comm Costs | Predicted Latency with Comm and HW Init Costs | Predicted Latency with All Costs | Actual Latency |
|---|---|---|---|---|---|
| SW/SW | 2509 | 2509 | 2509 | 2509 | 2141 |
| SW/HW | 2516 | 2541 | 4905 | 4905 | 3967 |
| $HW_1$/SW | 376 | 461 | 2825 | 2860 | 2975 |
| $HW_2$/SW | 392 | 469 | 2833 | 2851 | 3141 |
| $HW_3$/SW | 577 | 654 | 3018 | 3035 | 3004 |
| $HW_1$/HW | 383 | 493 | 2857 | 2892 | 3042 |
| $HW_2$/HW | 399 | 501 | 2865 | 2883 | 2803 |

Table 5.2: The Predicted and Actual Runtimes in Milliseconds

| Pipe Asst | ARE with No Overhead | ARE with Comm Costs | ARE with Comm and HW Init Costs | ARE with All Costs |
|---|---|---|---|---|
| SW/SW | 0.1719 | 0.1719 | 0.1719 | 0.1719 |
| SW/HW | 0.3658 | 0.3595 | 0.2365 | 0.2365 |
| $HW_1$/SW | 0.8736 | 0.8450 | 0.0504 | 0.0387 |
| $HW_2$/SW | 0.8752 | 0.8507 | 0.0981 | 0.0923 |
| $HW_3$/SW | 0.8079 | 0.7823 | 0.0047 | 0.0103 |
| $HW_1$/HW | 0.8741 | 0.8379 | 0.0608 | 0.0493 |
| $HW_2$/HW | 0.8577 | 0.8213 | 0.0221 | 0.0285 |

Table 5.3: The Absolute Relative Error (ARE) for the Predicted Latencies

The latency for each solution for an image of size 40185 pixels was estimated using the SHARPP and is shown in Table 5.2. We also estimated what the latencies would be if overhead costs were ignored. In Table 5.2 there are three sets of predicted latencies without all of the overhead costs: without all overhead costs, with only communication costs[3], and with only communication and hardware initialization costs. RIPS built executable pipelines for each solution. These pipelines were executed once in the Eclipsecolorer profiling tool so that the actual execution times could be compared with SHARPP's estimates. SHARPP selected the SW/SW pipeline assignment as the optimal solution which was the fastest solution in both actual results and estimated results. For small pipelines, the all software solution is often the optimal solution, because the hardware initialization cost is several times larger than processing the pipeline entirely in software.

In Table 5.3 the actual pipeline solution latencies are compared to the estimated latencies using the *absolute relative error (ARE)*. The relative error is:

$$\frac{(\text{measured latency} - \text{estimated latency})}{\text{measured latency}}. \tag{5.1}$$

Relative error can be either negative or positive, but we are more interested in the absolute distance the relative error is from zero. ARE gives us the absolute distance from zero. Our model of the target architecture that includes the overhead costs of the device initialization, reprogramming and communication costs is much more accurate than if they had not been included. As stated previously, most FPGA-based co-synthesis tools ignore most or all overhead costs. From Table 5.3, the range in ARE for the predicted latency with all overhead costs is 1-24%, but most are less than 10%. Table 5.3 shows the ARE for the three sets of predicted latencies that either have none or some of the overhead costs. The predicted latencies without any of the overhead are generally much shorter than the predicted latencies with overhead, especially when many components are

[3]Codesign environments that take communication costs into account are ASIC-based systems and not FPGA-based.

```
generatePipeline(size) {
  pipeline p = new pipeline();

  for (int i = 0; i < (size - 1); i++}
    component c = randomPickFromNonHistComponent();
    p.addToPipeline(c);
  }

  component c = randomPickFromAllComponents();
  p.addToPipeline(c);

  return p;
}
```

Figure 5.4: Algorithm to Generate Random Pipelines

assigned to hardware. When none of the overhead costs are accounted for, the error of the predicted latencies have a range of 17-88% and most of the values have an ARE greater than 80%. If the communication costs are accounted for, the ARE improves slightly when compared to the predicted latencies without any overhead costs accounted for, but most of the AREs are greater than 78%. When communication and hardware initialization costs are accounted for in the predicted latencies, the AREs are similar to the case where all overhead costs are accounted for. In this final set of predicted latencies, most of the AREs are below 10%.

If we had ignored all overhead costs or included only the communication costs, the $HW_1$/SW pipeline assignment would be selected as the optimal solution, which has an actual latency approximately 40% slower than the optimal pipeline assignment's actual latency. In general, the actual latencies for the pipeline assignments with hardware assignments were approximately one order of magnitude slower than the predicted latencies without overhead costs. When the communication and initialization costs are accounted for, the correct optimal solution is picked for this example.

### 5.8.2 SHARPP Experiment

In this experiment, forty different pipelines were randomly generated. Two pipelines for each pipeline size from one to 20 were generated. The algorithm

| Test Number | Pipeline |
|---|---|
| 1 | (SOURCE) → (minFilter) → (SINK) |
| 5 | (SOURCE) → (ed) → (emAnd) → (maxFilter) → (SINK) |
| 10 | (SOURCE) → (ed) → (minFilter) →(emAnd) → (maxFilter) → (hist3c8b1bin) → (SINK) |
| 15 | (SOURCE) → (minFilter) → (minFilter) → (minFilter) → (mf) → (minFilter) → (maxFilter) → (ed) → (ed) → (SINK) |
| 20 | (SOURCE) → (mf) → (minFilter) → (mf) → (minFilter) → (minFilter) → (minFilter) → (ed) → (emAnd) → (maxFilter) → (hist1c3b16bin) → (SINK) |
| 25 | (SOURCE) → (minFilter) → (minFilter) → (mf) → (minFilter) → (maxFilter) → (minFilter) →(emAnd) → (emAnd) → (mf) → (minFilter) → (maxFilter) → (emAnd) → (emAnd) → (SINK) |
| 30 | (SOURCE) → (mf) → (mf) → (emAnd) → (mf) → (maxFilter) → (emAnd) → (mf) → (maxFilter) → (maxFilter) → (maxFilter) → (minFilter) →(emAnd) → (emAnd) → (maxFilter) → (hist3c8b1bin) → (SINK) |
| 35 | (SOURCE) → (mf) → (maxFilter) → (minFilter) → (maxFilter) → (ed) → (ed) → (maxFilter) → (emOr) → (emAnd) → (emOr) → (ed) → (minFilter) → (ed) → (ed) → (mf) → (emAnd) → (emAnd) → (hist1c3b16bin) →(SINK) |
| 40 | (SOURCE) → (ed) → (ed) → (minFilter) → (minFilter) →(emAnd) → (ed) → (ed) →(emAnd) → (emAnd) → (minFilter) → (maxFilter) → (emAnd) → (emAnd) → (mf) → (ed) → (mf) → (minFilter) → (minFilter) → (mf) → (emAnd) → (SINK) |

Table 5.4: Selected Pipelines from the SHARPP Test

for creating random pipelines is shown in Figure 5.4. Random pipelines are built using the ipBLOC components listed in Table 3.1 in Chapter 3. The histogram components can only use used for the last component of a pipeline, so the first $N-1$ components in an $N$-stage pipeline are chosen from the remaining components: median filter, maximum filter, minimum filter, edge detector, and the two edge map components.

Dynamo was used to solve the pipeline assignment problem for each test pipeline and for a 40185 pixel image with the fixed time limit. Once PA was solved the pipelines were built. We used Dynamo to calculate the predicted latency (with and without overhead costs) for the optimal solution. The executable pipelines were then run once in the Eclipsecolorer profiler to measure the actual latencies.

Appendix D shows the pipeline assignments made by Dynamo for the test pipelines. These results show pipelines with five or fewer components tend to be assigned to software. Large pipelines, especially ones with ten or more components, tend to be assigned completely to hardware. Appendix D also

| Test Number | Predicted Latency with No Overhead | Predicted Latency with Comm Costs | Predicted Latency with Comm and HW Init Costs | Predicted Latency with All Costs | Actual Latency |
|---|---|---|---|---|---|
| 1 | 1111 | 1111 | 1111 | 1111 | 1309 |
| 5 | 375 | 504 | 2868 | 2920 | 3169 |
| 10 | 743 | 947 | 3311 | 3380 | 3571 |
| 15 | 1411 | 2002 | 4366 | 4538 | 4789 |
| 20 | 1701 | 2317 | 4681 | 4854 | 5955 |
| 25 | 1785 | 2426 | 4790 | 4998 | 6012 |
| 30 | 2114 | 2757 | 5121 | 5385 | 8560 |
| 35 | 2575 | 3453 | 5817 | 6094 | 10922 |
| 40 | 3450 | 4555 | 6919 | 7305 | 12217 |

Table 5.5: Selected Predicted and Actual Latency Results from the SHARPP Test

shows the predicted (with and without overhead costs) and actual latencies for the test pipelines for a 40185 pixel image. A subset of the test pipelines are shown in Table 5.4. The pipelines in Table 5.4 are numbered so that they can be referenced in Table 5.5, which shows the predicted and actual latencies for these pipelines, and Table 5.6, which shows the absolute relative error (ARE) for these pipelines.

As with the previous tests, the quality of the predicted latencies depends on the overhead costs included. Figure 5.5 shows how the AREs for the four types of predicted latencies compare. When all overhead costs are accounted for, most of the pipelines with less than or equal to 15 components have an ARE less than 20%. When the hardware initialization and communication costs are included in the predicted latency, the ARE is similar to but still worse than the ARE for the predicted latency with all overhead costs. When no overhead costs are accounted for, the test pipelines have AREs between 70-90%. When the communication cost is included in the predicted latency, the ARE is smaller than the ARE for the predicted latency with no overhead costs, but still worse

| Test Number | ARE with No Overhead | ARE with Comm Costs | ARE with Comm and HW Init Costs | ARE with All Costs |
|---|---|---|---|---|
| 1 | 0.1513 | 0.1513 | 0.1513 | 0.1513 |
| 5 | 0.8817 | 0.8410 | 0.0950 | 0.0786 |
| 10 | 0.7919 | 0.7348 | 0.0728 | 0.0535 |
| 15 | 0.7054 | 0.5820 | 0.0883 | 0.0524 |
| 20 | 0.7144 | 0.6109 | 0.2139 | 0.1849 |
| 25 | 0.7031 | 0.5965 | 0.2033 | 0.1687 |
| 30 | 0.7530 | 0.6779 | 0.4018 | 0.3709 |
| 35 | 0.7642 | 0.6838 | 0.4674 | 0.4420 |
| 40 | 0.7176 | 0.6272 | 0.4337 | 0.4021 |

Table 5.6: Selected Absolute Relative Error (ARE) Results from the SHARPP Test

than the ARE with all overhead costs included. Including all of the overhead costs is the only way to ensure the most accurate predicted latencies.

### 5.8.3 Error Analysis

From these two experiments, we can see that error tends to accumulate in the pipeline assignment latency predictions even when accounting for overhead costs. On average, the predicted latencies with overhead costs included have an ARE of 23%, although the ARE is strongly affected by pipeline size. We expect some issues with the SHARPP accurately predicting the optimal solution. If two pipeline assignments have approximately equal latencies, then the amount of accumulated error in a pipeline assignment might change which one is selected as fastest. In the two component experiment, $HW_1$/SW is expected to be slightly slower than $HW_2$/SW, but is actually slightly faster in execution. We are interested in minimizing the error in the pipeline assignment latency calculation to ensure the optimal solution is chosen. This section addresses the cause of the error and an idea of how to correct it.

First, we present the reasons why error is entering the latency calculation.

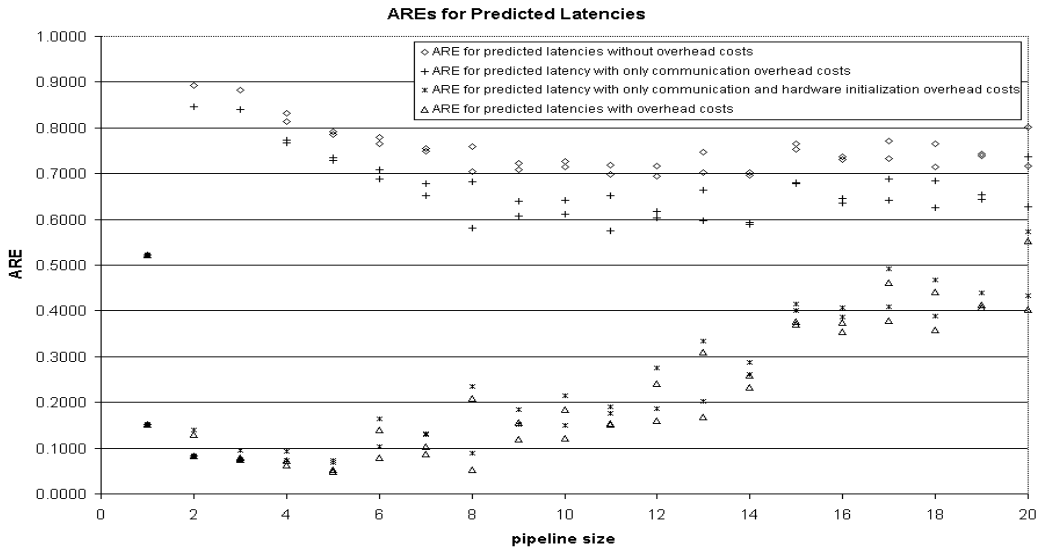


Figure 5.5: Absolute Relative Error with and without Overhead Costs for the SHARPP Test

The error is primarily from noise in the implementation profiles and overhead cost profiles. All profile data are based on the average of three runs, and the execution time for each implementation and overhead cost varies. These variations are caused by several factors:

1. The operating system could raise or lower the priority of the profiler task.
2. The Java virtual machine starts or ends the garbage collection thread.
3. The software process monitoring the *done* register on hardware implementations takes an inconsistent amount of time.
4. The profiler has a measurement error of $\pm 5$ ms.

This noise might be barely noticeable, but the runtime of most of the implementations is so small that these factors can have a profound affect on the profiles. Some of the implementations have a large standard deviation in the average runtime in the profiles, which means that the actual latencies might deviate widely

from the predicted amount. The error could be minimized by profiling all of the implementations more, which will be done in the future.

Even with the noise in the profiles minimized, error will still accumulate in the latency calculation. Since the latency calculation is done by summing all of the implementation and overhead costs, all of the noise in each profile accumulates as the number of profiles included in a latency calculation increases. For example, if we just focus on the ±5 ms measurement error in the profiler, a pipeline assignment that has 50 profiles in its latency calculation will be off by as much as ±250 ms for just profiler error. Since error cannot be removed, the error needs to be predicted so that it can be compensated for. Therefore, we need a way to estimate the potential amount of error in a pipeline assignment, gauge its impact in the latency calculation, and then correct the latency calculation.

Figure 5.5 from Section 5.8.2 shows the absolute relative error as a function of pipeline size. As expected, the error is proportional to pipeline size, although it is also dependent on the number of profiles in the pipeline assignment. If there are a cluster of solutions around the optimal latency, then which solution from the cluster we choose does not matter. We analyze the annotated solutions to the two stage pipeline described in Section 5.8.1 in Table 5.1. The summary of the profiles used in the latency calculation for each pipeline assignment in Table 5.1 are found in Table 5.7. In this table each of the possible implementations and overhead methods are enumerated across the top of the table. The software implementations' and overhead methods' profiles only include the runtime ($R$). The hardware implementations' profiles include runtime ($R$), input communication ($c_i$) and output communication ($c_o$). The values in the table indicate how many occurrences of each profile occur in the pipeline assignment's latency calculation. From the last column in Table 5.7, we can see that the pipeline assignments have between two and ten profiles for the same 2-stage pipeline. The error accumulated in the SW/SW solution is from adding the *mf:sw* profile to the *histogram:sw* profile for a total of two profiles. The error accumulated in the $HW_2$/HW solution is from adding the profiles from *mf:*$HW_2$ (comm:in + runtime + comm:out), *histogram:HW* (comm:in + runtime + comm:out),

| | mf:sw | mf:hw$_1$ | | | mf:hw$_2$ | | | mf:hw$_3$ | | | Pad | Unpad | hist:sw | hist:hw | | | Reprog | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Soln | R | $c_i$ | R | $c_o$ | $c_i$ | R | $c_o$ | $c_i$ | R | $c_o$ | R | R | R | $c_i$ | R | $c_o$ | R | Total |
| SW/SW | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 |
| SW/HW | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 5 |
| HW$_1$/SW | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 7 |
| HW$_2$/SW | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 6 |
| HW$_3$/SW | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 6 |
| HW$_1$/HW | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 2 | 10 |
| HW$_2$/HW | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 2 | 9 |

Table 5.7: Summary of Profiles Used in the Latency Calculation for Median Filter → Histogram

unpadding and two reprogramming for a total of nine profiles. Therefore, the $HW_2$/HW pipeline assignment has potentially five times as much error in the latency calculation compared to the SW/SW pipeline assignment.

In the future, we plan to implement an error model for the pipeline assignment problem to correct for the error accumulation in the latency calculation. This model will use the standard deviation for each implementation and overhead method, as one noisy profile can introduce more error than several profiles with minimal noise. With further experimentation, we will be able to determine if the runtimes for each implementation and overhead method are normally distributed. Statistical analysis [79] can be used to develop an error model if the error is normally distributed.

## 5.9 Conclusions

The Dynamo system embodies runtime solutions to several different problems relevant to hardware/software codesign for image processing pipeline applications. This system implements all aspects of my research, including runtime partitioning, runtime interface synthesis, and the execution subsystem. Our experiments show that pipeline specifications can be transformed into executable pipelines through runtime partitioning and interface synthesis. Our model for runtime partitioning uses a more accurate formalization of overhead costs than other FPGA-based co-synthesis environments, which gives us more accurate results for latency predictions. In the next chapter, we discuss the future research goals of our research.

# Chapter 6

# Conclusion and Future Work

## 6.1 Summary

This dissertation has introduced the pipeline assignment problem (PA), algorithms for solving PA, and a runtime environment for doing runtime synthesis of pipelines. The pipeline assignment problem is the optimization problem of finding a pipeline assignment that minimizes latency and meets area constraints for a given hardware device. The pipeline assignment problem is a variation of the hardware/software codesign partitioning problem formulated specifically for Field Programmable Gate Arrays (FPGAs). This problem was solved using both optimal and heuristic techniques. Three optimal techniques were presented: exhaustive search, integer linear programming, and dynamic programming. Three heuristic techniques were presented: greedy, random, and local search.

These algorithms were analyzed for two different time limits to determine which algorithms to use in the runtime environment. The recommended algorithms for the fixed time limit of 500 ms are:

- **TAC1:** Dynamic programming for 1- to 20-stage pipelines.
- **TAC2:** Dynamic programming for 1- to 7-stage pipelines; local search with a greedy initial solution, 1-Opt neighborhood, and tabu search for 8-

and 9-stage pipelines; and local search with an all hardware initial solution, 1-Opt neighborhood, and tabu search for 10- to 20-stage pipelines.

- **TAC3:** Dynamic programming for 1- to 6-stage pipelines; local search with a greedy initial solution, 1-Opt neighborhood, and tabu search for 7- to 13-stage pipelines; and local search with an all hardware initial solution, 1-Opt neighborhood, and tabu search for 14- to 20-stage pipelines.

The recommended algorithms for the adaptive time limit of 100 ms per pipeline stage are:

- **TAC1:** Dynamic programming for 1- to 20-stage pipelines.
- **TAC2:** Dynamic programming for 1- to 15-stage pipelines; and local search with an all hardware initial solution, 1-Opt neighborhood, and tabu search for 16- to 20-stage pipelines.
- **TAC3:** Dynamic programming for 1- to 7-stage pipelines; local search with a greedy initial solution, 1-Opt neighborhood, and tabu search for 8- to 19-stage pipelines; and local search with an all hardware initial solution, 1-Opt neighborhood, and tabu search for 20-stage pipelines.

The runtime environment is responsible for taking the pipeline specification from the user, solving the pipeline assignment problem, building the executable pipeline, and executing the pipeline. The three major subsystems are pipeline assignment, pipeline compilation, and pipeline execution. The Software/HArdware Runtime Procedural Partitioning (SHARPP) tool solves the pipeline assignment problem with the algorithms outlined above. The Runtime Interfacing for Pipeline Synthesis (RIPS) tool creates executable pipelines. The Packet Exchange Platform (PEP) connects the pipelines built by the RIPS to the runtime environment, so that data can pass between the pipeline and the runtime environment. Pipelines can be specified, implemented, and run with very short turn around times. Experimentation showed an average absolute

relative error of 23% between the pipeline assignment's predicted and actual latency due to measurement noise in the component implementation profiles. The use of overhead costs in the latency model leads to a more accurate prediction of the latency for a pipeline assignment than if they were not included.

## 6.2 Future Work

There are a number of directions this research can be continued. More components and implementations can be added to the image processing Basic Library of Components. Runtime partitioning can benefit from extending the pipeline assignment problem to allow more reprogramming scenarios, complex pipeline structures, more complex objective functions, and different target architectures. There are a few changes to the runtime environment that would allow the pipeline compilation stage to be skipped, as well as allowing the runtime environment to work in on a network of workstations. Finally, the error model discussed in Chapter 5 will be implemented in the SHARPP. These topics are discussed in detail below.

Reprogramming constraints can be loosened to allow more assignments to hardware implementations. Reprogramming on hardware/hardware boundaries will allow longer hardware subsequences to be realized. Bitstream merging will be implemented to allow the bitstreams of several small components to be combined so that they can execute on the hardware together without reprogramming. Reprogramming in parallel with software processing will be supported to obtain faster execution times.

In the future we plan to support complex pipeline structures in the pipeline specification. Such structures will include pipelines with branches and loops. There are several cases where control structures would be helpful. Median filtering can be applied multiple times to an image based on either a set number of iterations or on the amount of noise removed from the previous iteration. Histograms give valuable information about whether the image contrast needs adjusting, so the histogram output can be used to decide whether pre-processing

is needed on the image before the rest of the pipeline executes. These scenarios fundamentally change the pipeline assignment problem, since some of the knowledge about how the pipeline will execute is runtime dependent. Allowing complex pipeline structures will allow more complex image processing applications to be specified. Only linear pipeline structures are supported.

During optimization, pipeline assignment minimizes only latency. Other factors might also benefit from minimization, such as power consumption. Minimizing power consumption or a combination of power and latency may be a better optimization goal for domains such as embedded computing or space-based processing.

There are many other target architectures that will be explored. Our long range plans are to target the embedded processors on some newer FPGA devices such as the Altera Stratix or Xilinx Virtex-II Pro. Here the software runs on an embedded processor and the hardware on the closely connected reconfigurable fabric. While the design process is similar, interfaces and coupling costs for this new architecture are significantly different. In addition, a system with both a host processor and embedded processors for running software, as well as reconfigurable hardware for running hardware implementations, presents new challenges for dynamic hardware/software codesign of embedded systems. The Dynamo system was designed to easily adapt to these differences and should be useful for solving this problem at runtime. In addition, just as the pipeline assignment problem can be extended for different types of reconfigurable devices, it can also be extended to a Network of Workstations (NOW) environment. The coupling costs will be extended to include network communication times. The implementation latency will also incorporate different processor types and speeds.

There a few key areas of improvement for the runtime environment. The interface synthesis subsystem will be altered to execute the components directly, instead of building a Java executable. Being able to execute the pipeline solution on the fly will avoid creating files that need to be stored and compiled before the pipeline can be executed. Next, the communication layer can be expanded

to support message-based remote execution using a system such as JavaPorts [29, 62] to support reconfigurable hardware on remote machines. This model supports message exchanges between processes on different machines, and permits an application to use specialized hardware on remote machines for computation. While the shared memory model currently used in the packet exchange platform demonstrates that a communication layer can be abstract and flexible, the message passing model extends the communication layer so the packet exchange platform can also be distributed.

Finally, the accumulating error in the pipeline assignment's latency calculation will be accounted for in the pipeline assignment problem. While the accumulating error cannot be completely removed from the latency, the error can be minimized. First of all, more component profiling will reduce the overall error in the latency calculation. Second, an error model based on the standard deviation of the profiles will be added to the latency calculations. As a results of these two steps, the algorithms for solving the pipeline assignment problem will more accurately predict a pipeline assignment's latency.

## 6.3 Conclusions

In summary, we have presented a runtime co-synthesis environment for image processing. Users can specify simple applications using components from the ipBLOC library. Our runtime environment solves the pipeline assignment problem, builds the executable pipeline and executes it all within strict time limits. Our runtime environment includes a model of how the target architecture can be used and how to combine components so that a pipeline assignment's latency and area properties are accurately calculated. Our target architecture model for one general purpose processor and one FPGA formulates all of the possible overhead costs associated with it, which makes this research unique. When overhead costs are included the average absolute relative error of the estimated latencies is 23%, which is nearly a 50% improvement over models not using overhead costs.

# Appendix A

# Glossary

$AFS(K_N)$ Given a pipeline $P_N$ with solution $K_N$, $AFS(K_N)$ is the area of the final subsequence.

**Area Profile** The number of slices a particular hardware implementation takes on an FPGA device, denoted $a(c_{ix})$ where $c_{ix}$ is an implementation of component $c_i$ with implementation type $x$. *See slice.*

**Augmented Solution** Given a pipeline $P_N$ with a solution $K_N$, a solution $K_{N+i}$ for the superpipeline $P_{N+i}$ can be defined by $K_{N+i}(c_j) = K_N(c_j)$ for $j \in [1, N]$ such that all the values of $K_{N+i}(c_j)$ for $j \in [N+1, N+i]$ are defined. *See superpipeline.*

**Basic Block** A set of sequentially executing lines of code usually without control code.

$c_{ix}$ Given a pipeline $P_N$ and $i \in [1, N]$, $c_{ix}$ is an implementation of the component $c_i$ with implementation type $x$.

**Component** Represents an algorithm that can be implemented in hardware or software.

**Final Subsequence** Given a pipeline $P_N$ with solution $K_N$, the subsequence that contains implementation $K(c_N)$ is the final subsequence.

**Grain** An individual line or block of code used as one unit when partitioning a system specification. The size of a grain depends on the granularity of the partitioning tool. *See granularity.*

**Granularity** A term used in both the hardware/software codesign problem and the parallel computing scheduling problem that refers to the unit of code the partitioning or scheduling tool uses when solving the problem. Scheduling tools have traditionally used a dynamic granularity size, while most partitioning tools use a fixed granularity.

$H$ Given a pipeline $P_N$ with solution $K_N$, $H$ is the set of all hardware subsequences. $H$ is null if there are no components assigned to hardware implementations in the solution $K_N$.

**Hardware Subsequence** A subsequence where the implementation type is hardware.

$hwInitCost$ The constant latency needed to initialize the FPGA device.

$hwInitCostFn(H)$ If $H \neq \emptyset$, then $hwInitCostFn(H) = hwInitCost$. Otherwise, $hwInitCostFn(H) = 0$.

**I** The set of all implementations for all components in the ipBLOC.

$I_i$ Given a pipeline $P_N$ and $i \in [1, N]$, $I_i$ is the set of all implementations for component $c_i$.

**Implementation Type** Hardware or software.

$K_N$ Given a pipeline $P_N$, $K_N$ is the function that maps the elements of $P_N$ onto **I**.

**Latency Profile** The amount of execution time a component implementation takes in milliseconds for an image size $s$, denoted $l(c_{ix}, s)$. *See* $c_{ix}$.

**Maximal Subsequence** Given a solution $K_N$ to a pipeline $P_N$, and $i$ and $j$ such that $1 \leq i \leq j \leq N$, a *maximal subsequence* of $K$ is a sequence of implementations $K_N(c_m)$ for $m \in [i, j]$ such that all implementations in the sequence have the same type $x$ and components $c_{i-1}$ and $c_{j+1}$ are not assigned to type $x$.

**Optimal Pipeline Assignment** A valid pipeline assignment that has minimal latency for an image size $s$.

**Optimal Solution** *See Optimal Pipeline Assignment.*

**Optimal Substructure** A problem has optimal substructure when the optimal solution to a size $N$ problem contains an optimal solution to the size $N-1$ problem.

**Pipeline** An ordered series of components, denoted $P_N = (c_1, c_2, ..., c_N)$. A pipeline $P_N$ has $N$ components, which makes it an $N$-stage pipeline or a size $N$ pipeline.

**Pipeline Assignment** A solution to the pipeline assignment problem is a function $K_N$ that maps the elements of $P_N$ onto $\mathbf{I}$, where $\mathbf{I}$ is the set of implementations for all components, such that $K_N(c_i) = c_{ix}$ where $x$ is the implementation type and $c_{ix} \in I_i$.

**Pipeline Assignment Problem** the optimization problem of finding a pipeline assignment $K_N$ for a pipeline $P_N$ for an image size $z$ with the minimum latency such that the area for $K_N$ is less than or equal to the hardware device.

**Restricted Solution** Given a pipeline $P_N$ with a solution $K_N$ and $i \in [1, N-1]$, a solution $K_i$ for the subpipeline $P_i$ can be defined such that $K_i(c_j) = K_N(c_j)$ for all $j \in [1, i]$. *See subpipeline.*

**RIPS** The Runtime Interfacing for Pipeline Synthesis tool determines how to combine implementations to create an executable pipeline.

$S$ Given a pipeline $P_N$ with solution $K_N$, $S$ is the set of all software subsequences. $S$ is null if there are no components assigned to software implementations in the solution $K_N$.

**SEHA** *See Solution Ending with a Hardware Assignment.*

**SESA** *See Solution Ending with a Software Assignment.*

**SHARPP** The Software/HArdware Runtime Procedural Partitioning tool solves the pipeline assignment problem to find an assignment of components to implementations that minimize the pipeline's execution time and meets the area constraint for a given image and device size.

**Sink** A special component used to end a pipeline so that edge processing for the last component in a pipeline and post-processing can be modeled. Must be implemented in software.

**Slice** A basic configurable logic block on an FPGA used to implement the hardware circuit. FPGA devices have a finite number of slices available.

**Software Subsequence** A subsequence where the implementation type is software.

**Solution** *See Pipeline Assignment.*

**Solution Ending with a Hardware Assignment (SEHA)** A solution $K_N$ for a pipeline $P_N$ where $K_N(c_N)$ is assigned to a hardware implementation.

**Solution Ending with a Software Assignment (SESA)** A solution $K_N$ for a pipeline $P_N$ where $K_N(c_N)$ is assigned to a software implementation.

**Source** A special component used to start a pipeline so that edge processing for the first component in a pipeline and pre-processing can be modeled. Must be implemented in software.

**Subpipeline** Given a pipeline $P_N$ and some $i \in [1, N-1]$, a subpipeline is a pipeline $P_i = (c_1, c_2, ..., c_i)$ of the first $i$ components of $P_N$.

**Superpipeline** Given a pipeline $P_N$ and some $i \in \mathbb{Z}^+$, the pipeline $P_{N+i} = (c_1, c_2, ..., c_{N+i})$, where the first $N$ components are the pipeline $P_N$.

**Subsequence** Given a solution $K_N$ to a pipeline $P_N$, and $i$ and $j$ such that $1 \leq i \leq j \leq N$, a *subsequence* of $K$ is a sequence of implementations $K_N(c_m)$ for $m \in [i, j]$ such that all implementations in the sequence have the same type.

**TAC** Total Area Constraint represents the size of the FPGA device in number of slices.

**Valid Pipeline Assignment** A pipeline assignment that has area less than or equal to the total area constraint.

**Valid Solution** *See Valid Pipeline Assignment.*

# Appendix B

# Pipeline Assignment Proofs

## B.1 NP-Complete Proof

A *pipeline assignment*, $K_N$, maps the components of a pipeline $P_N$ to hardware and software implementations. There are two primary characteristics of a pipeline assignment: latency and area. The latency of a pipeline assignment is the total execution time of all of the implementations in the pipeline assignment plus all of the overhead costs. The area of the pipeline assignment is determined by hardware implementations executed in series (*hardware subsequences*). The area of a pipeline assignment is the largest hardware subsequence area. The pipeline assignment problem (PA) is the optimization problem of finding a pipeline assignment $K_N$ for a pipeline $P_N$ and image size $z$ with the minimum latency, such that the area for $K_N$ is less than or equal to the area of the hardware device. A *valid* pipeline assignment (or valid solution) has an area that meets the area requirements of the FPGA. An *optimal* pipeline assignment (or an optimal solution) is a valid solution with minimum latency.

The decision problem statement for the pipeline assignment problem is:

> *Given a pipeline $P_N$ and positive integers $k$, $l$ and $m$, is there a pipeline assignment $K_N$ with area $A(K_N) \leq k$, and $L(K_N, m) \leq l$ for image size $m$?*

Let $Q$ be the decision problem statement and $PS$ be the set of all possible pipeline assignments. Let $M = \{x \in PS : Q(x) = \text{ “yes”}\}$ . Given an instance, $x$, a deciding algorithm accepts $x$ if the output of the algorithm is “yes”. If the output is “no,” the algorithm rejects $x$. A language is considered decided by an algorithm if every instance in $L$ is accepted and every instance not in $M$ is rejected. If a language can be decided in polynomial time, the language is part of the complexity class $P$.

The pipeline assignment problem is first checked to be a member of the complexity class $NP$. To be a member of $NP$, for each “yes” instance of the problem, $x$, there must exist a concise certificate, $y$, that proves $x \in M$ in polynomial time. Furthermore, for each “no” instance, no certificate exists. For the pipeline assignment decision problem, the certificate is a solution $K_N$ such that for all $hss \in H$ $A(hss) \leq k$, and $L(K_N, m) \leq l$ for image size $m$. The pipeline assignment's area and latency characteristics can be determined in $O(N)$. If the answer to the decision problem is “no,” no solution that meets all the criteria exists and hence no certificate exists for the instance. Based on these criteria, the pipeline assignment decision problem is a member of the complexity class $NP$.

Next the problem is checked for inclusion in the $NPC$ complexity class. Since the problem is a member of $NP$, the problem is in $NPC$ if one of two statements are proved:

$$\forall A \in NP, A \leq PA$$

or

$$\exists A \in NPC, A \leq PA$$

The easiest path to take is to find a decision problem in NPC for which a polynomial time transformation between it and pipeline assignment exists. The best candidate decision problem is Subset Partition (SSP) [31]. The problem statement for SSP is:

*given a set $E$ of weighted elements ($w : E \rightarrow \mathbb{Z}^+$), does there exist a*

*partition* $(X, Y)$ *of* $E$ *such that* $w(X) = w(Y)$*?*

Given an SSP instance, $x$, it is transformed into a pipeline assignment instance, $x'$. First, for each member of $E$ a software and hardware implementation is determined. All of the implementations have no latency for all image sizes. All of the coupling costs between elements is zero. The area of each implementation is given the weight of the original element in $E$. Since the area for both implementations is the same, the nodes can be assigned to either $H$ or $S$ and still have the same weight as they would have in SSP. There is no implicit ordering of the set so all possible arrangements of the elements can be explored. Without ordering, all components assigned to $S$ are considered one software subsequence and all components assigned to $H$ are considered one hardware subsequence. The upper bound on the latency ($l$) will be zero. The area constraint ($k$) is $1/2 * w(E)$.

**"yes" → "yes"**

Let $x$ be a "yes" instance of SSP. Let $x$' be a pipeline assignment instance created by transforming $x$. All possible solutions for $x$' pass the time constraint. Only valid solutions for $x'$ pass the area constraint, and are solutions to the SSP problem for the instance $x$. Solution $K$ is a valid solution if the the area for $H$ and $S$ are $\leq 1/2 * w(E)$. If for a given solution, $K$, $H$ has an area $< 1/2 * w(E)$, then $S$ must have an area $> 1/2 * w(E)$. Therefore, if a valid solution exists, then the area for $H$ and $S = 1/2 * w(E)$. Since the size of both partitions is $1/2 * w(E)$, the partitions are a solution to the SSP problem.

**"no" → "no"**

Let $x$ be a "no" instance of SSP. Let $x$' be a pipeline assignment instance created by transforming $x$. All possible solutions for $x$' pass the time constraint. Only valid solutions for $x'$ pass the area constraint, and are solutions to the SSP problem for the instance $x$. Solution $K$ is a valid solution if the the area for $H$ and $S$ are $\leq 1/2 * w(E)$. Since there is no solution to a "no" instance, there

is no solution that passes the area constraint. Since the total area constraint is equal to $1/2 * w(E)$ there is no way to partition the instance $x'$ into two partitions and satisfy the total area constraint. Therefore, "no" instances in SSP are transformed into "no" instances in pipeline assignment. □

## B.2 Optimal Substructure Proof

Given a pipeline $P_N$, an implementation $x$ of component $c_N$ and an area $y \in [0, TAC]$, $f_N^*(x, y)$ is the minimum latency of an optimal solution $K_N^*$ for $P_N$, if one exists, such that the last component's implementation $K_N^*(c_N)$ equals $x$ and the final subsequence's area $AFS(K_N^*)$ equals $y$.[1] If there does not exist a $K_N^*$ that satisfies the given $x$ and $y$, $f_N^*(x, y) = \infty$. Given the pipeline $P_N$ and some $i \in [1, N]$ the optimal latency for the subpipeline $P_i$ for an implementation $x = K_i(c_i)$ and area of the final subsequence $y = AFS(K_i)$ is defined recursively as:

$$f_i^*(x,y) = \begin{cases} l(x,s) & \text{if } i = 1 \\ \min\limits_{\substack{z \in I_{i-1} \\ a \in B(x,y)}} \{f_{i-1}^*(z,a) + X(z,x,s) + l(x,s)\} & \text{otherwise} \end{cases}$$

where component $c_i$ has implementation $x$, the image size is $z$, the final subsequence has area $y$, $I_{i-1}$ is the set of all implementations for component $c_{i-1}$ and $B(x, y)$ is defined as:

$$B(x,y) = \begin{cases} \{0, ..., TAC\} & \text{if } x \text{ is implemented in software} \\ \{y - A(x)\} & \text{if } x \text{ is implemented in hardware.} \end{cases} \quad \text{(B.1)}$$

The function $B(x, y)$ defines the set of possible values $AFS(K_{N-1})$ can have to satisfy $K_N^*$. For example, given $f_i^*(x, y)$ where $x$ is a hardware implementation

[1] The latency and solution functions known to be optimal are denoted with astricks.

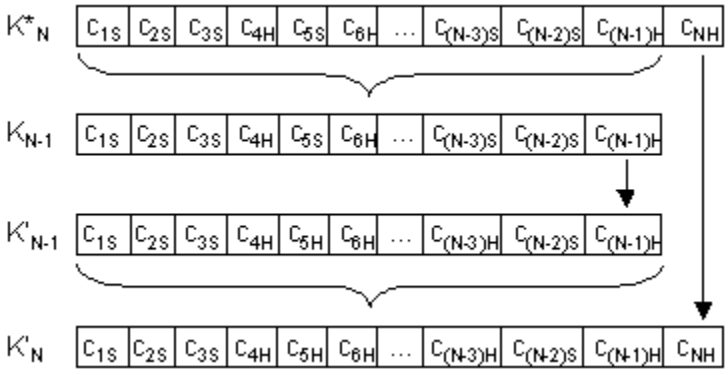


Figure B.1: Overview of How the Solutions $K^*_{N-1}$, $K'_{N-1}$, and $K'_N$ Are Created

of component $c_i$, the only $K_{i-1}$ solutions that are possible subsolutions to $K^*_i$ have $AFS(K_{N-1}) = y - a(x)$. If $x$ is a software implementation, reprogramming has occurred and any valid $K_{i-1}$ solution is a candidate subsolution for $K^*_i$. Therefore, there is only one value for the AFS that candidate subsolutions can have for hardware implementations and TAC values for the AFS that candidate solutions can have for software implementations. Hardware initialization adds a non-linearity to the model, and is not part of the formulation. A workaround for the non-linearity is discussed in Chapter 4.

**Proof:** Let $P_N$ be some pipeline $(c_1, c_2, ..., c_N)$ with optimal solution $K^*_N$ for a device of size $TAC$ and an image of size $z$. Let $x = K^*_N(c_N)$ and $y = AFS(K^*_N)$. The optimal solution $K^*_N$ has latency $f^*_N(x, y)$. An example of a solution $K^*_N$ is in Figure B.1. A restricted solution $K_{N-1}$ for subpipeline $P_{N-1}$ is constructed from $K^*_N$. An example of the transformation from $K^*_N$ to $K_{N-1}$ is shown in Figure B.1. We claim that $K^*_N$ has optimal substructure, so $K_{N-1}$ is an optimal solution for the $P_{N-1}$ pipeline under the restriction that the implementation of $c_{N-1}$ is $K^*_N(c_{N-1})$ and the area of the final subsequence is $AFS(K_{N-1})$. We need to prove that the solution $K_{N-1}$ has optimal latency under these restrictions, i.e. that $L(K_{N-1}) = f^*_{N-1}(K_{N-1}(c_{N-1}), AFS(K_{N-1}))$. This proof is by contradiction.

Assume that $K_{N-1}$ is not an optimal solution for $P_{N-1}$ under these restrictions. There must then exist a solution $K'_{N-1}$ such that $AFS(K'_{N-1}) = AFS(K_{N-1})$, $K'_{N-1}(c_{N-1}) = K_{N-1}(c_{N-1})$, and $L(K'_{N-1}) < L(K_{N-1})$. An example of solution $K'_{N-1}$ is shown in Figure B.1[2]. The solution $K'_{N-1}$ is augmented to create the solution $K'_N$ for the pipeline $P_N$ by defining $K'_N(c_N) = x$. We next prove that $K'_N$ is a valid solution to pipeline $P_N$ with better latency than $K^*_N$, which contradicts the fact that $K^*_N$ is optimal.

First, $K'_N$ is shown to be a valid solution for the $P_N$ pipeline. Solution $K'_{N-1}$ does not violate the area constraints, otherwise it would not be considered a solution, so the concern lies in augmenting $K'_{N-1}$ with implementation $x$ to create $K'_N$. There are two cases to consider depending on the implementation of $x = K^*_N(c_N)$.

1. $x$ implemented in software: Augmenting $K'_{N-1}$ with a software implementation causes $AFS(K'_N) = 0$, which trivially satisfies the area constraint.

2. $x$ implemented in hardware: For all solutions $K_i$ defined such that $K_i(c_i) = c_{ih}$ the area of the final subsequence is defined as $AFS(K_i) = AFS(K_{i-1}) + a(c_{ih})$. By construction $AFS(K'_{N-1}) = AFS(K_{N-1})$, so $AFS(K'_N) = AFS(K^*_N)$. Since $K^*_N$ did not violate the area constraint, $K'_N$ does not violate the area constraint either.

For both cases of $x$, $AFS(K'_N) \leq TAC$.

Second, the fact that $L(K'_N) < f^*_N$(x,y) is shown. $K'_N$ has latency:

$$L(K'_N) = L(K'_{N-1}) + X(K'_{N-1}(c_{N-1}), x) + l(x). \tag{B.2}$$

The latency for $K^*_N$ can be written in the same manner:

$$L(K^*_N) = L(K_{N-1}) + X(K_{N-1}(c_{N-1}), x) + l(x). \tag{B.3}$$

[2]If the final subsequence for solution $K_{N-1}$ is a hardware subsequence with $m$ components, then for both $K_{N-1}$ and $K'_{N-1}$ the components from $c_{N-m-1}$ to $c_{N-1}$ are assigned to their hardware implementations and component $c_{N-m-2}$ is assigned to a software implementation. Therefore, $K_{N-1}(c_i) = K'_{N-1}(c_i)$ for $i \in [N-m-2, N-1]$.

Since $K_{N-1}(c_{N-1}) = K'_{N-1}(c_{N-1})$ by construction, the boundary costs in Equations B.2 and B.3 are equal. By assumption $L(K'_{N-1}) < L(K_{N-1})$, so $L(K'_N) < L(K^*_N)$, which contradicts the fact that the $K^*_N$ solution is optimal. Therefore, there cannot exist a solution $K'_{N-1}$ with $AFS(K'_{N-1}) = AFS(K_{N-1})$, and $K'_{N-1}(c_{N-1}) = K_{N-1}(c_{N-1})$ that has lower latency than the $K_{N-1}$ solution. Therefore, the solution $K_{N-1}$ has optimal latency $L(K_{N-1})$ which is equal to:

$$f^*_{N-1}(K_{N-1}(c_{N-1}), AFS(K_{N-1})) \tag{B.4}$$

and PA has the optimal substructure property. □

## B.3 Maximum Number of Stored Partial Assignments Formula

This section proves the maximum number of solutions saved per iteration in dynamic programming. Assume that for all iterations $i \in [1, N]$ that each component $c_i \in P_N$ has $|h|$ hardware implementations and $|s|$ software implementations. We will prove that in the worst case

$$|h|^i + |s| \sum_{k=0}^{i-1} |h|^k \tag{B.5}$$

solutions are saved for iteration $i$. We will prove this by induction. Without loss of generality, assume that the TAC is large enough to fit the entire pipeline $P_N$ into hardware, so that SEHAs are not eliminated based on area. Also assume that for all iterations $i \in [1, N]$ that given any two SEHAs $K'$ and $K''$ that $AFS(K') \neq AFS(K'')$ even if $K'(c_i) \neq K''(c_i)$. Finally, assume that all hardware implementations in **I** have unique area. These three assumptions ensure that no SEHAs are pruned because they conflict with another solution's AFS.

**Iteration 1:** For iteration 1 $|s|$ SESAs and $|h|$ SEHAs are saved. We know that all of the SESAs have a final subsequence with no area. By assumption all

of the SEHAs have a unique area for their final subsequence.

**Iteration $i$:** Assume for iteration $i$ there are $|h|^i + |s| \sum_{k=0}^{i-1} |h|^k$ solutions saved. There are $|s|$ SESAs and $|h|^i + |s| \sum_{k=1}^{i-1} |h|^k$ SEHAs. Each SEHA has a unique final subsequence area by assumption.

**Iteration $i$+1:** For iteration $i$+1 each solution from iteration $i$ is augmented with each of the $|s|$ software implementations. SESAs are saved under two criteria. Given $K_{i+1}$ such that $K_{i+1}(c_{i+1}) = c_{(i+1)x}$ where $x \in s_{i+1}$ then:

1. If there does not exist a solution $K'_{i+1}$ such that $K'_{i+1}(c_{i+1}) = c_{(i+1)x}$, then $K'_{i+1}$ is added.

2. If there does exist a solution $K'_{i+1}$ such that $K'_{i+1}(c_{i+1}) = c_{(i+1)x}$, then $K_{i+1}$ is added if and only if $L(K_{i+1}) < L(K'_{i+1})$.

Therefore, the only SESAs that are saved for iteration $i + 2$ are the optimal solutions for each $x \in s_{i+1}$. Therefore $|s|$ SESAs are saved for the next iteration.

For iteration $i + 1$ SEHAs are created by augmenting each solution from iteration $i$ with each of the $|h|$ hardware implementations. All of the SEHAs created by this process are valid solutions by assumption. SEHAs are saved under two criteria. Given $K_{i+1}$ such that $K_{i+1}(c_{i+1}) = c_{(i+1)x}$ where $x \in h_{i+1}$ then:

1. If there does not exist a solution $K'_{i+1}$ such that $K'_{i+1}(c_{i+1}) = c_{(i+1)x}$, then $K'_{i+1}$ is added.

2. If there does exist a solution $K'_{i+1}$ such that $K'_{i+1}(c_{i+1}) = c_{(i+1)x}$, then $K_{i+1}$ is added if and only if:

   (a) $AFS(K_{i+1}) \neq AFS(K'_{i+1})$ or

   (b) $AFS(K_{i+1}) = AFS(K'_{i+1})$ and $L(K_{i+1}) < L(K'_{i+1})$.

In case 2a) the SEHA $K_{i+1}$ can be added without eliminating another solution. In case 2b) the hardware solution can only be added at the elimination of another solution. By assumption all SEHAs have unique area, so each hardware solution

is added without eliminating any other solutions. Therefore, the maximum number of solutions are saved, which means that $|h|^i + |s| \sum_{k=0}^{i-1} |h|^k$ solutions will be saved for each hardware implementation. The total number of SEHAs saved in iteration $i+1$ is $(|h|^i + |s| \sum_{k=0}^{i-1} |h|^k)|h|$.

The total number of solutions saved is $(|h|^{i+1} + |s| \sum_{k=0}^{i-1} |h|^k)|h| + |s|$. This value is equal to $|h|^{i+1} + |s| \sum_{k=0}^{i} |h|^k$. If we set $l = i + 1$, this value is equal to $|h|^l + |s| \sum_{k=0}^{l-1} |h|^k$. Therefore, the induction holds and there are $|h|^i + |s| \sum_{k=0}^{i-1} |h|^k$ solutions saved for the *ith* iteration. □

# Appendix C

# Constraints for the ILP Formulation

The following constraints have to be fulfilled for a valid assignment.

### Implementation Constraint

Each component can only be assigned to one implementation at a time. This constraint is implemented as:

$$\forall i \in P_N \; assignment[hw, i] + assignment[sw, i] = 1$$

Because integer linear programming restricts the *assignment* values to integers, we are assured only one implementation can be chosen.

### Source and Sink Nodes Constraint

The source and sink components are distinguished members of the pipeline, as they can only be implemented in software. Therefore, they need to be constrained in this manner:

$$assignment[sw, source] = 1$$

$$assignment[sw, sink] = 1$$

**Cover Constraint**

All of the components need to be assigned to an implementation. Therefore, all of the elements of the *assignment* matrix can be summed for a total of $N$, where $N$ is the pipeline size:

$$\sum_{i \in I} \sum_{j \in P_N} assignment[i, j] = N$$

**Area Constraints**

All of the hardware subsequences in *assignment* need to have an area less than the total area constraint. To start a hardware subsequence there is a shift from software to hardware. At the end of the hardware subsequence, there is a shift from hardware to software. During the hardware subsequence all of the interfaces are hardware to hardware transitions. Therefore, hardware subsequences can be found by looking for cases where the above three conditions occur using the *interface* variable. Once a hardware subsequence is found, its area is compared against the total area constraint. Formally:

*a hardware subsequence from component i to component j is possible*
*if* $\sum_{k=i+1}^{j} area\_cost[hw, k] \leq TAC$

This translates into the constraint:

$$\begin{aligned} TAC \quad \geq \quad & (interface[swxhw, i] + interface[hwxsw, j] + \\ & \sum_{k=i+1}^{j} interface[hwxhw, i]) - k + j) * \sum_{k=i+1}^{j} area\_cost[hw, k] \end{aligned}$$

When the interface conditions are met, then the constraint reduces to whether the sum of the subsequence's area costs are less than the total area constraint.

If any of the entries in the *interface* matrix are zero, then the parenthetical statement on the right hand side of the constraint is zero or negative, which always satisfy the constraint.

There is one problem with this constraint. It should really be an "if and only if" constraint. To make that change, we will add this constraint:

$$TAC * (1 - interface[swxhw, i] - interface[hwxsw, i] - < \sum_{k=i+1}^{j} area\_cost[hw, k] \sum_{k=i+1}^{j} interface[hwxhw, i] + k - j)$$

**Interface Constraints**

There are several constraints on the *interface* variable. There are four rows in the *interface* matrix that signify the type of interface between two components: hardware/hardware, hardware/software, software/hardware and software/software. Each of these rows has three constraints. We describe all four cases using the notation X/Y to indicate the interface types, where $X, Y \in I$.

The X/Y row in the *interface* matrix is dependent on this formulation:

> *Component i and Component i+1 have a "X/Y" interface if and only if Component i is implemented in X and Component i+1 is implemented in Y.*

This statement translates into three constraints:

$$assignment[X, i] + assignment[Y, i + 1] - 1 \leq interface[X \times Y, i]$$

$$assignment[X, i] \leq interface[X \times Y, i]$$

$$assignment[Y, i + 1] \leq interface[X \times Y, i]$$

### HaveHW Constraint

The *haveHW* constraint is dependent on this formulation:

> *haveHW is one if and only if one of the components in the pipeline is implemented in hardware*

When there exists a hardware assignment, then there must be at least one entry in the software to hardware row that is one. This formulation is split into two constraints:

$$\sum_{i=0}^{N} interface[swxhw, i] \geq haveHW$$

$$\forall i \in \{1...N-1\}\ haveHW \geq interface[swxhw, i]$$

### Non-Negativity Constraints

There are three non-negativity constraints: one for each variable.

$$\forall i \in I,\ j \in P_N\ assignment[i, j] \geq 0$$

$$\forall i \in I^2,\ j \in I\ interface[i, j] \geq 0$$

$$haveHW \geq 0$$

# Appendix D

# Tables of Results

## D.1 The Pipeline Assignment Problem Chapter Results

### D.1.1 Optimal Time Results

| Exhaustive Search Runtimes Per TAC | | | |
|---|---|---|---|
| Pipeline Size | TAC1 | TAC2 | TAC3 |
| 1 | 3 | 4 | 4 |
| 2 | 7 | 29 | 34 |
| 3 | 10 | 139 | 234 |
| 4 | 23 | 660 | 1,324 |
| 5 | 56 | 5,527 | 18,853 |

| Dynamic Programming Runtimes Per TAC | | | |
|---|---|---|---|
| Pipeline Size | TAC1 | TAC2 | TAC3 |
| 1 | 3 | 4 | 4 |
| 2 | 8 | 14 | 22 |
| 3 | 17 | 62 | 94 |
| 4 | 37 | 144 | 195 |
| 5 | 47 | 230 | 361 |
| 6 | 50 | 305 | 507 |
| 7 | 67 | 401 | 777 |
| 8 | 85 | 569 | 1,144 |
| 9 | 98 | 699 | 1,521 |
| 10 | 124 | 798 | 1,863 |
| 11 | 152 | 915 | 2,184 |
| 12 | 172 | 1,064 | 2,825 |
| 13 | 180 | 1,237 | 3,402 |
| 14 | 206 | 1,384 | 3,885 |
| 15 | 222 | 1,530 | 4,377 |
| 16 | 255 | 1,795 | 5,497 |
| 17 | 282 | 2,018 | 6,514 |
| 18 | 278 | 2,050 | 6,573 |
| 19 | 336 | 2,323 | 8,061 |
| 20 | 343 | 2,386 | 8,130 |

### D.1.2 Heuristic Results

| Greedy | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 2 | 0 | 1 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 2 | 3 | 2 | 1.0000 | 1.0054 | 1.0055 |
| 3 | 2 | 4 | 4 | 1.0000 | 1.0416 | 1.0438 |
| 4 | 5 | 6 | 5 | 1.0000 | 1.0111 | 1.0184 |
| 5 | 5 | 17 | 7 | 1.0000 | 1.2527 | 1.2744 |
| 6 | 7 | 12 | 10 | 1.0000 | 1.2140 | 1.2344 |
| 7 | 10 | 11 | 10 | 1.0000 | 1.4323 | 1.4721 |
| 8 | 14 | 17 | 13 | 1.0000 | 1.4277 | 1.4677 |
| 9 | 20 | 17 | 19 | 1.0000 | 1.4232 | 1.4698 |
| 10 | 21 | 26 | 22 | 1.0000 | 1.9241 | 1.9969 |
| 11 | 19 | 24 | 24 | 1.0000 | 1.6411 | 1.6957 |
| 12 | 23 | 23 | 24 | 1.0000 | 1.7604 | 1.8492 |
| 13 | 26 | 28 | 31 | 1.0000 | 1.8276 | 1.9144 |
| 14 | 23 | 32 | 44 | 1.0000 | 1.6513 | 1.7187 |
| 15 | 33 | 37 | 31 | 1.0000 | 1.8876 | 1.9685 |
| 16 | 30 | 46 | 44 | 1.0000 | 1.9027 | 2.0058 |
| 17 | 42 | 44 | 44 | 1.0000 | 2.3886 | 2.5444 |
| 18 | 40 | 46 | 43 | 1.0000 | 2.2282 | 2.3447 |
| 19 | 41 | 46 | 51 | 1.0000 | 1.8528 | 1.9687 |
| 20 | 48 | 54 | 50 | 1.0000 | 2.0227 | 2.1247 |

| Random | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 1 | 2 | 2 | 1.0000 | 10.1165 | 8.4774 |
| 2 | 3 | 5 | 7 | 1.0000 | 5.2049 | 5.9177 |
| 3 | 10 | 5 | 7 | 1.0000 | 1.9215 | 1.9381 |
| 4 | 20 | 32 | 29 | 1.0000 | 1.7612 | 1.8951 |
| 5 | 28 | 42 | 37 | 1.0000 | 1.3988 | 1.3959 |
| 6 | 33 | 37 | 44 | 1.0000 | 1.1900 | 1.2668 |
| 7 | 35 | 46 | 48 | 1.0000 | 1.3946 | 1.3585 |
| 8 | 41 | 49 | 55 | 1.0000 | 1.6168 | 1.7546 |
| 9 | 31 | 53 | 46 | 1.0000 | 1.5027 | 1.4948 |
| 10 | 62 | 73 | 74 | 1.0000 | 1.6759 | 1.6679 |
| 11 | 73 | 83 | 73 | 1.0000 | 1.5327 | 1.4803 |
| 12 | 68 | 73 | 77 | 1.0000 | 1.6512 | 1.6216 |
| 13 | 93 | 128 | 126 | 1.0000 | 1.8180 | 1.8570 |
| 14 | 74 | 94 | 67 | 1.0000 | 1.3344 | 1.2850 |
| 15 | 92 | 103 | 100 | 1.0000 | 1.4809 | 1.5731 |
| 16 | 62 | 96 | 92 | 1.0000 | 1.6672 | 1.7412 |
| 17 | 114 | 131 | 131 | 1.0000 | 1.5914 | 1.7969 |
| 18 | 121 | 116 | 104 | 1.0000 | 1.6061 | 1.7284 |
| 19 | 145 | 133 | 115 | 1.0000 | 1.4714 | 1.4529 |
| 20 | 149 | 155 | 166 | 1.0000 | 1.3895 | 1.6595 |

| All Software | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 1 | 1 | 0 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 1 | 1 | 2 | 1.0000 | 1.0077 | 1.0078 |
| 3 | 2 | 2 | 1 | 1.0000 | 1.0406 | 1.0429 |
| 4 | 2 | 1 | 2 | 1.0000 | 1.0111 | 1.0184 |
| 5 | 2 | 2 | 5 | 1.0000 | 1.3256 | 1.3503 |
| 6 | 2 | 4 | 3 | 1.0000 | 1.2909 | 1.3158 |
| 7 | 4 | 3 | 5 | 1.0000 | 1.4965 | 1.5434 |
| 8 | 4 | 9 | 5 | 1.0000 | 1.5961 | 1.6447 |
| 9 | 5 | 6 | 4 | 1.0000 | 1.6310 | 1.6904 |
| 10 | 5 | 6 | 6 | 1.0000 | 2.0261 | 2.1062 |
| 11 | 5 | 10 | 7 | 1.0000 | 1.6795 | 1.7365 |
| 12 | 16 | 11 | 8 | 1.0000 | 1.9698 | 2.0705 |
| 13 | 8 | 11 | 9 | 1.0000 | 1.9837 | 2.0833 |
| 14 | 14 | 8 | 10 | 1.0000 | 1.6513 | 1.7187 |
| 15 | 18 | 14 | 10 | 1.0000 | 2.0283 | 2.1212 |
| 16 | 12 | 11 | 17 | 1.0000 | 2.2970 | 2.4310 |
| 17 | 12 | 13 | 17 | 1.0000 | 2.6258 | 2.8018 |
| 18 | 12 | 13 | 13 | 1.0000 | 2.5902 | 2.7409 |
| 19 | 19 | 15 | 18 | 1.0000 | 2.0090 | 2.1395 |
| 20 | 20 | 17 | 15 | 1.0000 | 2.1259 | 2.2363 |

| All Hardware | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 1 | 2 | 3 | 1.0000 | 15.2395 | 15.2395 |
| 2 | 2 | 1 | 2 | 1.0000 | 5.8548 | 5.8549 |
| 3 | 2 | 3 | 4 | 1.0000 | 2.3479 | 2.3496 |
| 4 | 3 | 7 | 3 | 1.0000 | 2.4929 | 2.5005 |
| 5 | 3 | 18 | 5 | 1.0000 | 1.7391 | 1.7544 |
| 6 | 3 | 10 | 6 | 1.0000 | 3.9611 | 3.9756 |
| 7 | 5 | 18 | 6 | 1.0000 | 1.3308 | 1.3601 |
| 8 | 5 | 12 | 8 | 1.0000 | 1.4888 | 1.5177 |
| 9 | 5 | 23 | 7 | 1.0000 | 1.2093 | 1.2404 |
| 10 | 8 | 19 | 16 | 1.0000 | 1.2087 | 1.2472 |
| 11 | 7 | 26 | 11 | 1.0000 | 1.2206 | 1.2529 |
| 12 | 11 | 31 | 11 | 1.0000 | 1.2241 | 1.2723 |
| 13 | 9 | 30 | 19 | 1.0000 | 1.2820 | 1.3264 |
| 14 | 9 | 38 | 14 | 1.0000 | 1.1984 | 1.2355 |
| 15 | 11 | 17 | 27 | 1.0000 | 1.2933 | 1.3370 |
| 16 | 12 | 17 | 16 | 1.0000 | 1.3293 | 1.3880 |
| 17 | 13 | 19 | 22 | 1.0000 | 1.2511 | 1.3181 |
| 18 | 13 | 24 | 19 | 1.0000 | 1.3050 | 1.3654 |
| 19 | 14 | 27 | 21 | 1.0000 | 1.2116 | 1.2658 |
| 20 | 19 | 22 | 31 | 1.0000 | 1.2395 | 1.2886 |

### D.1.3 Local Search Fixed Time Limit Results

| Local Search Greedy, 1-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 8 | 3 | 3 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 10 | 6 | 4 | 1.0000 | 1.0054 | 1.0054 |
| 3 | 12 | 8 | 10 | 1.0000 | 1.0416 | 1.0425 |
| 4 | 16 | 17 | 23 | 1.0000 | 1.0111 | 1.0142 |
| 5 | 17 | 25 | 26 | 1.0000 | 1.2286 | 1.2097 |
| 6 | 34 | 30 | 33 | 1.0000 | 1.1984 | 1.1930 |
| 7 | 24 | 32 | 40 | 1.0000 | 1.4323 | 1.4564 |
| 8 | 28 | 57 | 62 | 1.0000 | 1.3765 | 1.3948 |
| 9 | 38 | 63 | 69 | 1.0000 | 1.3998 | 1.4347 |
| 10 | 39 | 77 | 80 | 1.0000 | 1.8849 | 1.8805 |
| 11 | 51 | 69 | 85 | 1.0000 | 1.6280 | 1.6755 |
| 12 | 51 | 104 | 114 | 1.0000 | 1.7224 | 1.7064 |
| 13 | 58 | 119 | 131 | 1.0000 | 1.7684 | 1.7746 |
| 14 | 76 | 97 | 117 | 1.0000 | 1.6513 | 1.7101 |
| 15 | 71 | 120 | 145 | 1.0000 | 1.8876 | 1.9521 |
| 16 | 87 | 158 | 210 | 1.0000 | 1.8729 | 1.9466 |
| 17 | 99 | 172 | 219 | 1.0000 | 2.3595 | 2.4644 |
| 18 | 90 | 205 | 223 | 1.0000 | 2.2059 | 2.2771 |
| 19 | 125 | 207 | 238 | 1.0000 | 1.8331 | 1.8990 |
| 20 | 134 | 203 | 239 | 1.0000 | 2.0147 | 2.1082 |

| Local Search Greedy, 1-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 5 | 13 | 11 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 5 | 198 | 331 | 1.0000 | 1.0010 | 1.0010 |
| 3 | 7 | 492 | 503 | 1.0000 | 1.0019 | 1.0019 |
| 4 | 18 | 504 | 507 | 1.0000 | 1.0006 | 1.0005 |
| 5 | 12 | 504 | 505 | 1.0000 | 1.0043 | 1.0043 |
| 6 | 19 | 506 | 508 | 1.0000 | 1.0022 | 1.0055 |
| 7 | 28 | 509 | 508 | 1.0000 | 1.0047 | 1.0083 |
| 8 | 29 | 509 | 509 | 1.0000 | 1.0273 | 1.0173 |
| 9 | 33 | 512 | 508 | 1.0000 | 1.0791 | 1.0246 |
| 10 | 38 | 509 | 513 | 1.0000 | 1.2573 | 1.0521 |
| 11 | 55 | 507 | 506 | 1.0000 | 1.3064 | 1.0810 |
| 12 | 52 | 514 | 510 | 1.0000 | 1.3783 | 1.0996 |
| 13 | 57 | 509 | 512 | 1.0000 | 1.4626 | 1.1616 |
| 14 | 63 | 511 | 514 | 1.0000 | 1.5049 | 1.2570 |
| 15 | 70 | 508 | 516 | 1.0000 | 1.6460 | 1.3930 |
| 16 | 87 | 507 | 510 | 1.0000 | 1.7107 | 1.4509 |
| 17 | 89 | 510 | 511 | 1.0000 | 2.1214 | 1.7388 |
| 18 | 94 | 508 | 511 | 1.0000 | 2.0345 | 1.7458 |
| 19 | 111 | 510 | 517 | 1.0000 | 1.7603 | 1.6450 |
| 20 | 118 | 506 | 510 | 1.0000 | 1.9353 | 1.7952 |

| Local Search Greedy, 2-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 504 | 503 | 505 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 11 | 13 | 11 | 1.0000 | 1.0054 | 1.0032 |
| 3 | 13 | 22 | 36 | 1.0000 | 1.0406 | 1.0376 |
| 4 | 18 | 39 | 46 | 1.0000 | 1.0111 | 1.0109 |
| 5 | 20 | 74 | 92 | 1.0000 | 1.2146 | 1.1783 |
| 6 | 32 | 99 | 121 | 1.0000 | 1.1973 | 1.1730 |
| 7 | 37 | 135 | 168 | 1.0000 | 1.4234 | 1.4365 |
| 8 | 43 | 225 | 237 | 1.0000 | 1.3715 | 1.3893 |
| 9 | 51 | 262 | 289 | 1.0000 | 1.3953 | 1.4289 |
| 10 | 67 | 313 | 388 | 1.0000 | 1.8821 | 1.8606 |
| 11 | 65 | 378 | 424 | 1.0000 | 1.6270 | 1.6554 |
| 12 | 89 | 456 | 477 | 1.0000 | 1.7044 | 1.6869 |
| 13 | 92 | 507 | 513 | 1.0000 | 1.7641 | 1.7524 |
| 14 | 102 | 504 | 505 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 111 | 507 | 504 | 1.0000 | 1.8741 | 1.9265 |
| 16 | 127 | 507 | 505 | 1.0000 | 1.8626 | 1.9323 |
| 17 | 141 | 504 | 503 | 1.0000 | 2.3521 | 2.4389 |
| 18 | 161 | 508 | 504 | 1.0000 | 2.2029 | 2.2563 |
| 19 | 173 | 503 | 505 | 1.0000 | 1.8316 | 1.8839 |
| 20 | 193 | 503 | 506 | 1.0000 | 2.0142 | 2.0940 |

| Local Search Greedy, 2-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 504 | 504 | 504 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 504 | 505 | 505 | 1.0000 | 1.0010 | 1.0010 |
| 3 | 504 | 502 | 504 | 1.0000 | 1.0019 | 1.0021 |
| 4 | 502 | 502 | 502 | 1.0000 | 1.0007 | 1.0005 |
| 5 | 503 | 503 | 506 | 1.0000 | 1.0064 | 1.0098 |
| 6 | 503 | 502 | 505 | 1.0000 | 1.0433 | 1.0167 |
| 7 | 503 | 504 | 504 | 1.0000 | 1.2247 | 1.0539 |
| 8 | 503 | 503 | 504 | 1.0000 | 1.2604 | 1.1516 |
| 9 | 503 | 506 | 505 | 1.0000 | 1.3604 | 1.2271 |
| 10 | 503 | 504 | 506 | 1.0000 | 1.7939 | 1.4388 |
| 11 | 504 | 503 | 503 | 1.0000 | 1.5913 | 1.6155 |
| 12 | 503 | 504 | 504 | 1.0000 | 1.7044 | 1.6838 |
| 13 | 504 | 503 | 505 | 1.0000 | 1.7641 | 1.7524 |
| 14 | 504 | 504 | 506 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 503 | 508 | 503 | 1.0000 | 1.8741 | 1.9260 |
| 16 | 504 | 504 | 504 | 1.0000 | 1.8626 | 1.9323 |
| 17 | 504 | 503 | 504 | 1.0000 | 2.3521 | 2.4388 |
| 18 | 504 | 504 | 508 | 1.0000 | 2.2029 | 2.2562 |
| 19 | 503 | 503 | 504 | 1.0000 | 1.8316 | 1.8839 |
| 20 | 503 | 505 | 505 | 1.0000 | 2.0142 | 2.0940 |

| Local Search Greedy, 3-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 504 | 504 | 503 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 505 | 505 | 505 | 1.0000 | 1.0054 | 1.0032 |
| 3 | 16 | 47 | 45 | 1.0000 | 1.0406 | 1.0374 |
| 4 | 19 | 78 | 85 | 1.0000 | 1.0111 | 1.0109 |
| 5 | 25 | 150 | 185 | 1.0000 | 1.2146 | 1.1780 |
| 6 | 38 | 224 | 265 | 1.0000 | 1.1973 | 1.1730 |
| 7 | 47 | 339 | 398 | 1.0000 | 1.4234 | 1.4362 |
| 8 | 58 | 480 | 487 | 1.0000 | 1.3715 | 1.3893 |
| 9 | 66 | 507 | 503 | 1.0000 | 1.3953 | 1.4289 |
| 10 | 82 | 503 | 502 | 1.0000 | 1.8821 | 1.8604 |
| 11 | 87 | 507 | 503 | 1.0000 | 1.6270 | 1.6553 |
| 12 | 113 | 503 | 508 | 1.0000 | 1.7044 | 1.6869 |
| 13 | 116 | 505 | 506 | 1.0000 | 1.7641 | 1.7524 |
| 14 | 151 | 505 | 505 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 165 | 503 | 504 | 1.0000 | 1.8741 | 1.9260 |
| 16 | 168 | 507 | 508 | 1.0000 | 1.8626 | 1.9323 |
| 17 | 197 | 505 | 504 | 1.0000 | 2.3521 | 2.4388 |
| 18 | 210 | 509 | 505 | 1.0000 | 2.2029 | 2.2562 |
| 19 | 252 | 507 | 504 | 1.0000 | 1.8316 | 1.8839 |
| 20 | 267 | 504 | 504 | 1.0000 | 2.0142 | 2.0940 |

| Local Search Greedy, 3-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 504 | 504 | 503 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 505 | 505 | 504 | 1.0000 | 1.0010 | 1.0010 |
| 3 | 506 | 503 | 507 | 1.0000 | 1.0019 | 1.0021 |
| 4 | 503 | 502 | 503 | 1.0000 | 1.0009 | 1.0005 |
| 5 | 502 | 503 | 504 | 1.0000 | 1.0794 | 1.0228 |
| 6 | 505 | 505 | 508 | 1.0000 | 1.1285 | 1.0443 |
| 7 | 504 | 504 | 503 | 1.0000 | 1.3606 | 1.2270 |
| 8 | 505 | 506 | 507 | 1.0000 | 1.3715 | 1.3834 |
| 9 | 503 | 506 | 507 | 1.0000 | 1.3953 | 1.4289 |
| 10 | 503 | 504 | 503 | 1.0000 | 1.8821 | 1.8604 |
| 11 | 503 | 503 | 507 | 1.0000 | 1.6270 | 1.6553 |
| 12 | 503 | 503 | 504 | 1.0000 | 1.7044 | 1.6869 |
| 13 | 503 | 504 | 505 | 1.0000 | 1.7641 | 1.7524 |
| 14 | 503 | 503 | 503 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 503 | 504 | 504 | 1.0000 | 1.8741 | 1.9260 |
| 16 | 503 | 504 | 504 | 1.0000 | 1.8626 | 1.9323 |
| 17 | 503 | 507 | 504 | 1.0000 | 2.3521 | 2.4388 |
| 18 | 503 | 505 | 506 | 1.0000 | 2.2029 | 2.2567 |
| 19 | 504 | 504 | 506 | 1.0000 | 1.8316 | 1.8839 |
| 20 | 503 | 504 | 505 | 1.0000 | 2.0142 | 2.0940 |

| Local Search Random, 1-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 2 | 6 | 2 | 1.0000 | 8.2859 | 8.2859 |
| 2 | 5 | 13 | 9 | 1.0000 | 4.6723 | 4.8754 |
| 3 | 15 | 25 | 31 | 1.0000 | 1.5199 | 1.4024 |
| 4 | 24 | 34 | 44 | 1.0000 | 1.5586 | 1.4155 |
| 5 | 34 | 39 | 52 | 1.0000 | 1.6702 | 1.6817 |
| 6 | 34 | 57 | 69 | 1.0000 | 3.9867 | 1.4336 |
| 7 | 44 | 71 | 91 | 1.0000 | 1.3728 | 1.3927 |
| 8 | 56 | 93 | 116 | 1.0000 | 1.5830 | 1.5669 |
| 9 | 82 | 123 | 155 | 1.0000 | 1.3122 | 1.4294 |
| 10 | 80 | 139 | 181 | 1.0000 | 1.4316 | 1.5093 |
| 11 | 96 | 161 | 195 | 1.0000 | 1.3005 | 1.3025 |
| 12 | 114 | 204 | 246 | 1.0000 | 1.7107 | 1.5789 |
| 13 | 123 | 244 | 270 | 1.0000 | 1.4692 | 1.5458 |
| 14 | 157 | 274 | 298 | 1.0000 | 1.4750 | 1.4725 |
| 15 | 145 | 307 | 368 | 1.0000 | 1.7458 | 1.6949 |
| 16 | 185 | 319 | 404 | 1.0000 | 1.7144 | 1.7302 |
| 17 | 230 | 361 | 445 | 1.0000 | 1.9276 | 1.8246 |
| 18 | 210 | 375 | 458 | 1.0000 | 1.8594 | 1.6079 |
| 19 | 258 | 448 | 497 | 1.0000 | 1.4247 | 1.6621 |
| 20 | 277 | 445 | 503 | 1.0000 | 1.5270 | 1.3832 |

| Local Search Random, 1-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 7 | 10 | 13 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 12 | 200 | 334 | 1.0000 | 1.0672 | 1.5970 |
| 3 | 13 | 478 | 504 | 1.0000 | 1.2384 | 1.2501 |
| 4 | 24 | 504 | 504 | 1.0000 | 1.9196 | 1.8255 |
| 5 | 33 | 504 | 507 | 1.0000 | 1.1405 | 1.2278 |
| 6 | 37 | 505 | 508 | 1.0000 | 3.6697 | 1.1538 |
| 7 | 50 | 507 | 507 | 1.0000 | 1.1043 | 1.0669 |
| 8 | 52 | 508 | 508 | 1.0000 | 1.2691 | 1.0646 |
| 9 | 84 | 509 | 506 | 1.0000 | 1.1724 | 1.0658 |
| 10 | 76 | 511 | 509 | 1.0000 | 1.1734 | 1.0591 |
| 11 | 88 | 511 | 508 | 1.0000 | 1.2843 | 1.1276 |
| 12 | 132 | 507 | 510 | 1.0000 | 1.5000 | 1.1953 |
| 13 | 131 | 508 | 507 | 1.0000 | 1.5000 | 1.2805 |
| 14 | 159 | 510 | 508 | 1.0000 | 1.3248 | 1.2550 |
| 15 | 169 | 514 | 509 | 1.0000 | 1.4986 | 1.5106 |
| 16 | 196 | 513 | 514 | 1.0000 | 1.7658 | 1.5684 |
| 17 | 194 | 511 | 511 | 1.0000 | 1.8573 | 2.0222 |
| 18 | 211 | 529 | 519 | 1.0000 | 1.8087 | 1.6713 |
| 19 | 294 | 515 | 515 | 1.0000 | 1.5036 | 1.4001 |
| 20 | 260 | 521 | 516 | 1.0000 | 1.6500 | 1.3777 |

| Local Search Random, 2-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 503 | 504 | 504 | 1.0000 | 3.2825 | 5.1366 |
| 2 | 15 | 29 | 26 | 1.0000 | 1.1706 | 2.0710 |
| 3 | 15 | 47 | 55 | 1.0000 | 1.5837 | 1.7946 |
| 4 | 33 | 93 | 84 | 1.0000 | 1.9535 | 2.0320 |
| 5 | 32 | 138 | 149 | 1.0000 | 1.6808 | 1.3530 |
| 6 | 55 | 203 | 222 | 1.0000 | 3.9318 | 3.8431 |
| 7 | 66 | 266 | 299 | 1.0000 | 1.4724 | 1.3411 |
| 8 | 86 | 372 | 372 | 1.0000 | 1.5009 | 1.6813 |
| 9 | 102 | 433 | 437 | 1.0000 | 1.4669 | 1.5150 |
| 10 | 115 | 461 | 487 | 1.0000 | 1.4988 | 1.4346 |
| 11 | 120 | 501 | 503 | 1.0000 | 1.3066 | 1.3375 |
| 12 | 126 | 505 | 504 | 1.0000 | 1.4273 | 1.5277 |
| 13 | 170 | 504 | 507 | 1.0000 | 1.6821 | 1.5897 |
| 14 | 178 | 504 | 506 | 1.0000 | 1.2302 | 1.2949 |
| 15 | 203 | 503 | 504 | 1.0000 | 1.5844 | 1.6053 |
| 16 | 248 | 505 | 507 | 1.0000 | 1.6847 | 1.6302 |
| 17 | 262 | 511 | 507 | 1.0000 | 1.7847 | 1.8146 |
| 18 | 266 | 505 | 507 | 1.0000 | 1.7909 | 2.0821 |
| 19 | 323 | 509 | 507 | 1.0000 | 1.5128 | 1.2830 |
| 20 | 341 | 517 | 518 | 1.0000 | 1.5537 | 1.8111 |

| Local Search Random, 2-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 504 | 503 | 503 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 503 | 505 | 504 | 1.0000 | 1.2136 | 1.7711 |
| 3 | 505 | 506 | 511 | 1.0000 | 1.1457 | 1.1704 |
| 4 | 503 | 503 | 503 | 1.0000 | 1.3342 | 1.9561 |
| 5 | 503 | 503 | 504 | 1.0000 | 1.2201 | 1.1161 |
| 6 | 503 | 503 | 505 | 1.0000 | 3.7846 | 3.6722 |
| 7 | 503 | 504 | 505 | 1.0000 | 1.2370 | 1.1479 |
| 8 | 504 | 503 | 506 | 1.0000 | 1.2322 | 1.1735 |
| 9 | 502 | 506 | 504 | 1.0000 | 1.3445 | 1.2797 |
| 10 | 503 | 506 | 505 | 1.0000 | 1.4273 | 1.4202 |
| 11 | 503 | 504 | 505 | 1.0000 | 1.4059 | 1.5384 |
| 12 | 504 | 504 | 509 | 1.0000 | 1.6007 | 1.5177 |
| 13 | 503 | 504 | 506 | 1.0000 | 1.5140 | 1.5956 |
| 14 | 505 | 509 | 508 | 1.0000 | 1.4689 | 1.4757 |
| 15 | 503 | 506 | 506 | 1.0000 | 1.6849 | 1.7790 |
| 16 | 504 | 507 | 504 | 1.0000 | 1.6731 | 1.7480 |
| 17 | 503 | 510 | 512 | 1.0000 | 1.8060 | 1.7535 |
| 18 | 504 | 511 | 512 | 1.0000 | 1.6927 | 1.8465 |
| 19 | 503 | 517 | 516 | 1.0000 | 1.5916 | 1.7259 |
| 20 | 503 | 522 | 513 | 1.0000 | 1.7870 | 1.6890 |

| Local Search Random, 3-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 504 | 504 | 504 | 1.0000 | 6.4898 | 8.0211 |
| 2 | 504 | 504 | 503 | 1.0000 | 1.9236 | 2.3823 |
| 3 | 30 | 75 | 77 | 1.0000 | 1.3652 | 1.4397 |
| 4 | 44 | 140 | 151 | 1.0000 | 1.3864 | 1.2890 |
| 5 | 46 | 265 | 277 | 1.0000 | 1.5284 | 1.6709 |
| 6 | 68 | 358 | 395 | 1.0000 | 1.2702 | 1.3646 |
| 7 | 81 | 449 | 483 | 1.0000 | 1.4704 | 1.4924 |
| 8 | 84 | 502 | 503 | 1.0000 | 1.6583 | 1.6513 |
| 9 | 104 | 508 | 504 | 1.0000 | 1.3385 | 1.3439 |
| 10 | 127 | 504 | 505 | 1.0000 | 1.5978 | 1.6393 |
| 11 | 150 | 505 | 504 | 1.0000 | 1.4144 | 1.4635 |
| 12 | 186 | 503 | 506 | 1.0000 | 1.6339 | 1.7345 |
| 13 | 178 | 503 | 504 | 1.0000 | 1.5525 | 1.6334 |
| 14 | 232 | 502 | 503 | 1.0000 | 1.3620 | 1.4077 |
| 15 | 241 | 502 | 506 | 1.0000 | 1.7448 | 1.5780 |
| 16 | 297 | 512 | 506 | 1.0000 | 1.6316 | 1.8498 |
| 17 | 344 | 506 | 503 | 1.0000 | 1.6315 | 1.9584 |
| 18 | 320 | 504 | 507 | 1.0000 | 1.7463 | 1.8813 |
| 19 | 359 | 508 | 510 | 1.0000 | 1.4997 | 1.7270 |
| 20 | 374 | 509 | 512 | 1.0000 | 1.6687 | 1.6411 |

| Local Search Random, 3-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 503 | 504 | 504 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 504 | 503 | 508 | 1.0000 | 1.1474 | 1.7434 |
| 3 | 505 | 504 | 505 | 1.0000 | 1.1859 | 1.1791 |
| 4 | 503 | 503 | 502 | 1.0000 | 1.1870 | 1.2994 |
| 5 | 503 | 502 | 503 | 1.0000 | 1.1709 | 1.1496 |
| 6 | 503 | 504 | 503 | 1.0000 | 3.8210 | 3.7360 |
| 7 | 503 | 505 | 503 | 1.0000 | 1.2183 | 1.3036 |
| 8 | 503 | 503 | 503 | 1.0000 | 1.5547 | 1.3761 |
| 9 | 504 | 503 | 507 | 1.0000 | 1.4277 | 1.2884 |
| 10 | 503 | 509 | 504 | 1.0000 | 1.6923 | 1.5649 |
| 11 | 504 | 507 | 505 | 1.0000 | 1.3656 | 1.4011 |
| 12 | 504 | 505 | 506 | 1.0000 | 1.5249 | 1.5787 |
| 13 | 503 | 507 | 507 | 1.0000 | 1.6035 | 1.5096 |
| 14 | 503 | 506 | 506 | 1.0000 | 1.5133 | 1.4834 |
| 15 | 503 | 505 | 505 | 1.0000 | 1.5796 | 1.7752 |
| 16 | 504 | 507 | 508 | 1.0000 | 1.5322 | 1.5604 |
| 17 | 504 | 506 | 510 | 1.0000 | 2.0174 | 2.0336 |
| 18 | 503 | 511 | 508 | 1.0000 | 1.5948 | 1.7189 |
| 19 | 504 | 515 | 518 | 1.0000 | 1.5009 | 1.7239 |
| 20 | 503 | 523 | 517 | 1.0000 | 1.7314 | 1.7196 |

| Local Search Software, 1-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 0 | 2 | 1 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 6 | 4 | 1 | 1.0000 | 1.0077 | 1.0077 |
| 3 | 10 | 10 | 7 | 1.0000 | 1.0406 | 1.0416 |
| 4 | 9 | 10 | 19 | 1.0000 | 1.0111 | 1.0142 |
| 5 | 13 | 18 | 19 | 1.0000 | 1.3256 | 1.3199 |
| 6 | 20 | 22 | 28 | 1.0000 | 1.2909 | 1.3141 |
| 7 | 16 | 25 | 28 | 1.0000 | 1.4965 | 1.5276 |
| 8 | 19 | 41 | 41 | 1.0000 | 1.5961 | 1.6193 |
| 9 | 29 | 44 | 53 | 1.0000 | 1.6310 | 1.6699 |
| 10 | 38 | 56 | 67 | 1.0000 | 2.0261 | 2.0814 |
| 11 | 32 | 56 | 74 | 1.0000 | 1.6795 | 1.7276 |
| 12 | 45 | 67 | 94 | 1.0000 | 1.9698 | 2.0411 |
| 13 | 47 | 77 | 96 | 1.0000 | 1.9837 | 2.0499 |
| 14 | 51 | 95 | 93 | 1.0000 | 1.6513 | 1.7101 |
| 15 | 58 | 94 | 128 | 1.0000 | 2.0283 | 2.1099 |
| 16 | 61 | 113 | 141 | 1.0000 | 2.2970 | 2.3929 |
| 17 | 70 | 120 | 171 | 1.0000 | 2.6258 | 2.7721 |
| 18 | 75 | 129 | 187 | 1.0000 | 2.5902 | 2.7023 |
| 19 | 92 | 149 | 190 | 1.0000 | 2.0090 | 2.1191 |
| 20 | 99 | 164 | 190 | 1.0000 | 2.1259 | 2.2304 |

| Local Search Software, 1-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 4 | 6 | 7 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 3 | 200 | 329 | 1.0000 | 1.0010 | 1.0010 |
| 3 | 6 | 489 | 507 | 1.0000 | 1.0019 | 1.0019 |
| 4 | 18 | 505 | 505 | 1.0000 | 1.0006 | 1.0005 |
| 5 | 13 | 506 | 508 | 1.0000 | 1.0043 | 1.0043 |
| 6 | 21 | 507 | 510 | 1.0000 | 1.0022 | 1.0055 |
| 7 | 19 | 508 | 508 | 1.0000 | 1.0047 | 1.0087 |
| 8 | 18 | 505 | 508 | 1.0000 | 1.0311 | 1.0192 |
| 9 | 26 | 513 | 510 | 1.0000 | 1.1163 | 1.0277 |
| 10 | 26 | 508 | 509 | 1.0000 | 1.2693 | 1.0579 |
| 11 | 38 | 510 | 509 | 1.0000 | 1.3196 | 1.0781 |
| 12 | 45 | 511 | 509 | 1.0000 | 1.4631 | 1.1177 |
| 13 | 60 | 507 | 507 | 1.0000 | 1.5804 | 1.1894 |
| 14 | 63 | 506 | 511 | 1.0000 | 1.4977 | 1.2618 |
| 15 | 59 | 513 | 514 | 1.0000 | 1.7399 | 1.3793 |
| 16 | 59 | 510 | 508 | 1.0000 | 2.0114 | 1.6353 |
| 17 | 72 | 508 | 515 | 1.0000 | 2.3201 | 1.9543 |
| 18 | 80 | 507 | 510 | 1.0000 | 2.3503 | 2.0549 |
| 19 | 82 | 515 | 518 | 1.0000 | 1.8864 | 1.8182 |
| 20 | 103 | 509 | 508 | 1.0000 | 2.0212 | 1.8153 |

| Local Search Software, 2-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 504 | 504 | 504 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 5 | 11 | 13 | 1.0000 | 1.0077 | 1.0055 |
| 3 | 12 | 20 | 38 | 1.0000 | 1.0406 | 1.0376 |
| 4 | 12 | 44 | 43 | 1.0000 | 1.0111 | 1.0109 |
| 5 | 17 | 64 | 72 | 1.0000 | 1.3197 | 1.2891 |
| 6 | 37 | 88 | 102 | 1.0000 | 1.2895 | 1.2687 |
| 7 | 38 | 123 | 153 | 1.0000 | 1.4950 | 1.4987 |
| 8 | 40 | 195 | 206 | 1.0000 | 1.5929 | 1.6066 |
| 9 | 41 | 241 | 275 | 1.0000 | 1.6283 | 1.6487 |
| 10 | 53 | 293 | 377 | 1.0000 | 2.0248 | 2.0530 |
| 11 | 62 | 367 | 427 | 1.0000 | 1.6786 | 1.7071 |
| 12 | 67 | 450 | 473 | 1.0000 | 1.9646 | 2.0053 |
| 13 | 81 | 504 | 507 | 1.0000 | 1.9804 | 2.0143 |
| 14 | 89 | 503 | 503 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 101 | 504 | 504 | 1.0000 | 2.0266 | 2.0798 |
| 16 | 109 | 504 | 506 | 1.0000 | 2.2949 | 2.3666 |
| 17 | 125 | 505 | 506 | 1.0000 | 2.6241 | 2.7468 |
| 18 | 130 | 506 | 503 | 1.0000 | 2.5885 | 2.6808 |
| 19 | 145 | 505 | 503 | 1.0000 | 2.0068 | 2.1020 |
| 20 | 161 | 505 | 504 | 1.0000 | 2.1259 | 2.2108 |

| Local Search Software, 2-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 504 | 503 | 504 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 505 | 506 | 508 | 1.0000 | 1.0010 | 1.0010 |
| 3 | 502 | 502 | 502 | 1.0000 | 1.0019 | 1.0021 |
| 4 | 503 | 504 | 505 | 1.0000 | 1.0007 | 1.0005 |
| 5 | 504 | 506 | 505 | 1.0000 | 1.0052 | 1.0081 |
| 6 | 504 | 503 | 503 | 1.0000 | 1.0742 | 1.0144 |
| 7 | 503 | 504 | 504 | 1.0000 | 1.2437 | 1.0546 |
| 8 | 504 | 504 | 505 | 1.0000 | 1.3792 | 1.2446 |
| 9 | 504 | 504 | 503 | 1.0000 | 1.5418 | 1.3033 |
| 10 | 503 | 502 | 503 | 1.0000 | 1.8868 | 1.5542 |
| 11 | 504 | 503 | 504 | 1.0000 | 1.6182 | 1.6042 |
| 12 | 504 | 506 | 504 | 1.0000 | 1.9646 | 2.0022 |
| 13 | 503 | 505 | 511 | 1.0000 | 1.9804 | 2.0143 |
| 14 | 503 | 503 | 504 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 503 | 504 | 504 | 1.0000 | 2.0266 | 2.0793 |
| 16 | 504 | 504 | 504 | 1.0000 | 2.2949 | 2.3666 |
| 17 | 503 | 501 | 506 | 1.0000 | 2.6241 | 2.7467 |
| 18 | 503 | 504 | 507 | 1.0000 | 2.5885 | 2.6808 |
| 19 | 504 | 504 | 506 | 1.0000 | 2.0068 | 2.1020 |
| 20 | 504 | 504 | 505 | 1.0000 | 2.1259 | 2.2108 |

| Local Search Software, 3-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 503 | 503 | 504 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 504 | 503 | 504 | 1.0000 | 1.0077 | 1.0055 |
| 3 | 21 | 33 | 42 | 1.0000 | 1.0406 | 1.0374 |
| 4 | 25 | 68 | 83 | 1.0000 | 1.0111 | 1.0109 |
| 5 | 27 | 135 | 158 | 1.0000 | 1.3197 | 1.2888 |
| 6 | 33 | 223 | 257 | 1.0000 | 1.2895 | 1.2687 |
| 7 | 42 | 313 | 388 | 1.0000 | 1.4950 | 1.4985 |
| 8 | 55 | 473 | 488 | 1.0000 | 1.5929 | 1.6066 |
| 9 | 70 | 503 | 503 | 1.0000 | 1.6283 | 1.6487 |
| 10 | 72 | 507 | 506 | 1.0000 | 2.0248 | 2.0527 |
| 11 | 82 | 509 | 504 | 1.0000 | 1.6786 | 1.7070 |
| 12 | 98 | 508 | 504 | 1.0000 | 1.9646 | 2.0053 |
| 13 | 113 | 504 | 509 | 1.0000 | 1.9804 | 2.0140 |
| 14 | 118 | 506 | 504 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 138 | 504 | 505 | 1.0000 | 2.0266 | 2.0793 |
| 16 | 159 | 505 | 503 | 1.0000 | 2.2949 | 2.3666 |
| 17 | 172 | 507 | 505 | 1.0000 | 2.6241 | 2.7467 |
| 18 | 185 | 503 | 503 | 1.0000 | 2.5885 | 2.6808 |
| 19 | 221 | 506 | 503 | 1.0000 | 2.0068 | 2.1020 |
| 20 | 233 | 514 | 503 | 1.0000 | 2.1259 | 2.2108 |

| Local Search Software, 3-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 503 | 504 | 503 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 503 | 503 | 506 | 1.0000 | 1.0010 | 1.0010 |
| 3 | 504 | 508 | 506 | 1.0000 | 1.0019 | 1.0021 |
| 4 | 505 | 503 | 503 | 1.0000 | 1.0008 | 1.0005 |
| 5 | 503 | 503 | 504 | 1.0000 | 1.1165 | 1.0256 |
| 6 | 504 | 504 | 504 | 1.0000 | 1.1999 | 1.0444 |
| 7 | 504 | 503 | 505 | 1.0000 | 1.4076 | 1.2605 |
| 8 | 503 | 506 | 503 | 1.0000 | 1.5929 | 1.6006 |
| 9 | 503 | 503 | 503 | 1.0000 | 1.6283 | 1.6487 |
| 10 | 503 | 503 | 508 | 1.0000 | 2.0248 | 2.0527 |
| 11 | 503 | 508 | 503 | 1.0000 | 1.6786 | 1.7071 |
| 12 | 503 | 504 | 505 | 1.0000 | 1.9646 | 2.0053 |
| 13 | 503 | 504 | 505 | 1.0000 | 1.9804 | 2.0143 |
| 14 | 503 | 506 | 504 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 503 | 505 | 503 | 1.0000 | 2.0266 | 2.0793 |
| 16 | 503 | 504 | 504 | 1.0000 | 2.2949 | 2.3666 |
| 17 | 504 | 504 | 507 | 1.0000 | 2.6241 | 2.7467 |
| 18 | 503 | 504 | 508 | 1.0000 | 2.5885 | 2.6808 |
| 19 | 504 | 504 | 506 | 1.0000 | 2.0068 | 2.1020 |
| 20 | 503 | 507 | 504 | 1.0000 | 2.1259 | 2.2108 |

| Local Search Hardware, 1-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 2 | 1 | 2 | 1.0000 | 15.2395 | 15.2395 |
| 2 | 3 | 5 | 10 | 1.0000 | 5.8548 | 5.8460 |
| 3 | 9 | 10 | 15 | 1.0000 | 2.3469 | 2.3382 |
| 4 | 14 | 24 | 38 | 1.0000 | 2.4913 | 2.4771 |
| 5 | 17 | 31 | 64 | 1.0000 | 1.7375 | 1.7291 |
| 6 | 16 | 47 | 74 | 1.0000 | 3.9605 | 3.9559 |
| 7 | 20 | 67 | 115 | 1.0000 | 1.3274 | 1.3343 |
| 8 | 26 | 90 | 157 | 1.0000 | 1.4871 | 1.4923 |
| 9 | 35 | 104 | 200 | 1.0000 | 1.2079 | 1.2193 |
| 10 | 30 | 136 | 244 | 1.0000 | 1.2076 | 1.2204 |
| 11 | 43 | 150 | 283 | 1.0000 | 1.2191 | 1.2322 |
| 12 | 37 | 188 | 341 | 1.0000 | 1.2218 | 1.2488 |
| 13 | 50 | 213 | 413 | 1.0000 | 1.2802 | 1.3038 |
| 14 | 54 | 227 | 447 | 1.0000 | 1.1974 | 1.2196 |
| 15 | 72 | 284 | 469 | 1.0000 | 1.2907 | 1.3188 |
| 16 | 67 | 320 | 497 | 1.0000 | 1.3286 | 1.3679 |
| 17 | 69 | 338 | 511 | 1.0000 | 1.2506 | 1.2964 |
| 18 | 78 | 400 | 512 | 1.0000 | 1.3028 | 1.3458 |
| 19 | 99 | 394 | 507 | 1.0000 | 1.2105 | 1.2507 |
| 20 | 105 | 432 | 509 | 1.0000 | 1.2367 | 1.2749 |

| Local Search Hardware, 1-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 2 | 7 | 4 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 5 | 183 | 360 | 1.0000 | 1.8381 | 1.8369 |
| 3 | 4 | 476 | 507 | 1.0000 | 1.3816 | 1.4037 |
| 4 | 11 | 507 | 508 | 1.0000 | 2.3503 | 2.3425 |
| 5 | 8 | 506 | 506 | 1.0000 | 1.5246 | 1.5196 |
| 6 | 16 | 504 | 509 | 1.0000 | 3.8219 | 3.8168 |
| 7 | 18 | 509 | 507 | 1.0000 | 1.1611 | 1.1635 |
| 8 | 24 | 508 | 510 | 1.0000 | 1.3262 | 1.3346 |
| 9 | 26 | 509 | 510 | 1.0000 | 1.0974 | 1.1155 |
| 10 | 35 | 513 | 515 | 1.0000 | 1.0864 | 1.1172 |
| 11 | 36 | 511 | 514 | 1.0000 | 1.1164 | 1.1515 |
| 12 | 36 | 518 | 513 | 1.0000 | 1.1267 | 1.1818 |
| 13 | 46 | 514 | 523 | 1.0000 | 1.1993 | 1.2481 |
| 14 | 62 | 513 | 519 | 1.0000 | 1.1375 | 1.1778 |
| 15 | 64 | 521 | 524 | 1.0000 | 1.2325 | 1.2762 |
| 16 | 75 | 523 | 524 | 1.0000 | 1.2704 | 1.3365 |
| 17 | 70 | 529 | 518 | 1.0000 | 1.2039 | 1.2772 |
| 18 | 89 | 531 | 527 | 1.0000 | 1.2633 | 1.3281 |
| 19 | 95 | 516 | 525 | 1.0000 | 1.1808 | 1.2370 |
| 20 | 90 | 520 | 530 | 1.0000 | 1.2120 | 1.2597 |

| Local Search Hardware, 2-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 503 | 504 | 504 | 1.0000 | 15.2395 | 15.2395 |
| 2 | 3 | 23 | 24 | 1.0000 | 5.5615 | 5.5543 |
| 3 | 8 | 53 | 63 | 1.0000 | 2.2175 | 2.2129 |
| 4 | 13 | 104 | 123 | 1.0000 | 2.4109 | 2.4131 |
| 5 | 20 | 196 | 223 | 1.0000 | 1.6649 | 1.6747 |
| 6 | 24 | 297 | 355 | 1.0000 | 3.9029 | 3.9130 |
| 7 | 30 | 433 | 477 | 1.0000 | 1.2730 | 1.2986 |
| 8 | 41 | 498 | 505 | 1.0000 | 1.4374 | 1.4622 |
| 9 | 45 | 503 | 505 | 1.0000 | 1.1678 | 1.1963 |
| 10 | 52 | 505 | 506 | 1.0000 | 1.1629 | 1.1984 |
| 11 | 66 | 504 | 505 | 1.0000 | 1.1803 | 1.2095 |
| 12 | 63 | 508 | 505 | 1.0000 | 1.1834 | 1.2287 |
| 13 | 78 | 507 | 509 | 1.0000 | 1.2394 | 1.2819 |
| 14 | 84 | 508 | 508 | 1.0000 | 1.1652 | 1.2007 |
| 15 | 111 | 505 | 508 | 1.0000 | 1.2570 | 1.2988 |
| 16 | 118 | 508 | 509 | 1.0000 | 1.2911 | 1.3478 |
| 17 | 128 | 507 | 507 | 1.0000 | 1.2157 | 1.2800 |
| 18 | 132 | 512 | 508 | 1.0000 | 1.2704 | 1.3290 |
| 19 | 156 | 508 | 514 | 1.0000 | 1.1825 | 1.2352 |
| 20 | 169 | 509 | 516 | 1.0000 | 1.2111 | 1.2592 |

| Local Search Hardware, 2-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 504 | 503 | 504 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 504 | 507 | 505 | 1.0000 | 1.8381 | 1.8369 |
| 3 | 504 | 507 | 507 | 1.0000 | 1.3816 | 1.5382 |
| 4 | 503 | 503 | 504 | 1.0000 | 2.3503 | 2.3437 |
| 5 | 504 | 503 | 504 | 1.0000 | 1.5600 | 1.5577 |
| 6 | 503 | 503 | 504 | 1.0000 | 3.8374 | 3.8477 |
| 7 | 503 | 504 | 505 | 1.0000 | 1.2114 | 1.2461 |
| 8 | 503 | 503 | 505 | 1.0000 | 1.4040 | 1.4498 |
| 9 | 504 | 509 | 508 | 1.0000 | 1.1618 | 1.1963 |
| 10 | 504 | 505 | 510 | 1.0000 | 1.1629 | 1.1984 |
| 11 | 503 | 506 | 509 | 1.0000 | 1.1803 | 1.2095 |
| 12 | 503 | 506 | 512 | 1.0000 | 1.1834 | 1.2287 |
| 13 | 504 | 506 | 514 | 1.0000 | 1.2394 | 1.2819 |
| 14 | 504 | 509 | 512 | 1.0000 | 1.1652 | 1.2007 |
| 15 | 504 | 513 | 513 | 1.0000 | 1.2570 | 1.2990 |
| 16 | 504 | 516 | 514 | 1.0000 | 1.2911 | 1.3478 |
| 17 | 503 | 517 | 522 | 1.0000 | 1.2157 | 1.2803 |
| 18 | 503 | 513 | 515 | 1.0000 | 1.2704 | 1.3294 |
| 19 | 503 | 522 | 521 | 1.0000 | 1.1827 | 1.2373 |
| 20 | 503 | 524 | 535 | 1.0000 | 1.2128 | 1.2597 |

| Local Search Hardware, 3-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 504 | 504 | 504 | 1.0000 | 15.2395 | 15.2395 |
| 2 | 505 | 504 | 504 | 1.0000 | 5.5615 | 5.5543 |
| 3 | 14 | 91 | 109 | 1.0000 | 2.2175 | 2.2129 |
| 4 | 23 | 208 | 245 | 1.0000 | 2.4109 | 2.4131 |
| 5 | 27 | 412 | 451 | 1.0000 | 1.6649 | 1.6747 |
| 6 | 31 | 504 | 505 | 1.0000 | 3.9029 | 3.9130 |
| 7 | 42 | 505 | 504 | 1.0000 | 1.2730 | 1.2986 |
| 8 | 52 | 504 | 507 | 1.0000 | 1.4374 | 1.4622 |
| 9 | 62 | 504 | 504 | 1.0000 | 1.1678 | 1.1963 |
| 10 | 68 | 503 | 506 | 1.0000 | 1.1629 | 1.1984 |
| 11 | 93 | 508 | 508 | 1.0000 | 1.1803 | 1.2095 |
| 12 | 93 | 510 | 506 | 1.0000 | 1.1834 | 1.2287 |
| 13 | 115 | 506 | 509 | 1.0000 | 1.2394 | 1.2819 |
| 14 | 125 | 510 | 510 | 1.0000 | 1.1652 | 1.2007 |
| 15 | 137 | 507 | 510 | 1.0000 | 1.2570 | 1.2988 |
| 16 | 165 | 507 | 508 | 1.0000 | 1.2911 | 1.3478 |
| 17 | 181 | 511 | 508 | 1.0000 | 1.2157 | 1.2800 |
| 18 | 186 | 512 | 508 | 1.0000 | 1.2704 | 1.3290 |
| 19 | 228 | 507 | 517 | 1.0000 | 1.1825 | 1.2354 |
| 20 | 251 | 509 | 515 | 1.0000 | 1.2111 | 1.2589 |

| Local Search Hardware, 3-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 503 | 504 | 504 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 504 | 505 | 503 | 1.0000 | 1.8381 | 1.8369 |
| 3 | 503 | 503 | 503 | 1.0000 | 1.3816 | 1.5701 |
| 4 | 503 | 502 | 502 | 1.0000 | 2.3512 | 2.3439 |
| 5 | 503 | 503 | 504 | 1.0000 | 1.5819 | 1.5833 |
| 6 | 504 | 504 | 504 | 1.0000 | 3.8722 | 3.8891 |
| 7 | 504 | 505 | 510 | 1.0000 | 1.2669 | 1.2986 |
| 8 | 503 | 504 | 505 | 1.0000 | 1.4374 | 1.4622 |
| 9 | 503 | 504 | 505 | 1.0000 | 1.1678 | 1.1963 |
| 10 | 505 | 506 | 507 | 1.0000 | 1.1629 | 1.1984 |
| 11 | 503 | 506 | 509 | 1.0000 | 1.1803 | 1.2095 |
| 12 | 503 | 508 | 512 | 1.0000 | 1.1834 | 1.2287 |
| 13 | 503 | 509 | 512 | 1.0000 | 1.2394 | 1.2819 |
| 14 | 503 | 510 | 513 | 1.0000 | 1.1652 | 1.2007 |
| 15 | 503 | 510 | 516 | 1.0000 | 1.2570 | 1.2990 |
| 16 | 504 | 508 | 516 | 1.0000 | 1.2911 | 1.3481 |
| 17 | 503 | 513 | 517 | 1.0000 | 1.2157 | 1.2800 |
| 18 | 503 | 514 | 522 | 1.0000 | 1.2710 | 1.3294 |
| 19 | 504 | 522 | 525 | 1.0000 | 1.1836 | 1.2367 |
| 20 | 503 | 522 | 529 | 1.0000 | 1.2125 | 1.2597 |

### D.1.4 Local Search Adaptive Time Limit Results

| Local Search Greedy, 1-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 2 | 6 | 6 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 4 | 4 | 10 | 1.0000 | 1.0054 | 1.0054 |
| 3 | 8 | 10 | 12 | 1.0000 | 1.0416 | 1.0425 |
| 4 | 14 | 18 | 22 | 1.0000 | 1.0111 | 1.0142 |
| 5 | 12 | 24 | 24 | 1.0000 | 1.2286 | 1.2097 |
| 6 | 24 | 27 | 29 | 1.0000 | 1.1984 | 1.1930 |
| 7 | 20 | 36 | 42 | 1.0000 | 1.4323 | 1.4564 |
| 8 | 32 | 52 | 56 | 1.0000 | 1.3765 | 1.3948 |
| 9 | 35 | 59 | 66 | 1.0000 | 1.3998 | 1.4347 |
| 10 | 39 | 79 | 79 | 1.0000 | 1.8849 | 1.8805 |
| 11 | 41 | 75 | 87 | 1.0000 | 1.6280 | 1.6755 |
| 12 | 58 | 98 | 108 | 1.0000 | 1.7224 | 1.7064 |
| 13 | 62 | 117 | 136 | 1.0000 | 1.7684 | 1.7746 |
| 14 | 68 | 100 | 123 | 1.0000 | 1.6513 | 1.7101 |
| 15 | 74 | 124 | 151 | 1.0000 | 1.8876 | 1.9521 |
| 16 | 87 | 162 | 215 | 1.0000 | 1.8729 | 1.9466 |
| 17 | 92 | 186 | 233 | 1.0000 | 2.3595 | 2.4644 |
| 18 | 102 | 199 | 245 | 1.0000 | 2.2059 | 2.2771 |
| 19 | 126 | 218 | 244 | 1.0000 | 1.8331 | 1.8990 |
| 20 | 125 | 208 | 250 | 1.0000 | 2.0147 | 2.1082 |

| Local Search Greedy, 1-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 4 | 13 | 10 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 3 | 90 | 145 | 1.0000 | 1.0010 | 1.0010 |
| 3 | 11 | 302 | 309 | 1.0000 | 1.0019 | 1.0019 |
| 4 | 28 | 405 | 409 | 1.0000 | 1.0006 | 1.0005 |
| 5 | 12 | 504 | 507 | 1.0000 | 1.0043 | 1.0043 |
| 6 | 28 | 605 | 605 | 1.0000 | 1.0022 | 1.0052 |
| 7 | 21 | 705 | 710 | 1.0000 | 1.0047 | 1.0068 |
| 8 | 29 | 805 | 807 | 1.0000 | 1.0066 | 1.0128 |
| 9 | 32 | 908 | 911 | 1.0000 | 1.0101 | 1.0118 |
| 10 | 39 | 1007 | 1018 | 1.0000 | 1.0574 | 1.0251 |
| 11 | 46 | 1109 | 1120 | 1.0000 | 1.1013 | 1.0252 |
| 12 | 54 | 1211 | 1214 | 1.0000 | 1.1625 | 1.0384 |
| 13 | 55 | 1311 | 1314 | 1.0000 | 1.2180 | 1.0479 |
| 14 | 66 | 1410 | 1412 | 1.0000 | 1.2748 | 1.0571 |
| 15 | 76 | 1511 | 1516 | 1.0000 | 1.3399 | 1.0694 |
| 16 | 84 | 1615 | 1614 | 1.0000 | 1.3976 | 1.1572 |
| 17 | 101 | 1720 | 1714 | 1.0000 | 1.7048 | 1.1976 |
| 18 | 99 | 1815 | 1825 | 1.0000 | 1.6371 | 1.2119 |
| 19 | 121 | 1920 | 1927 | 1.0000 | 1.5353 | 1.1596 |
| 20 | 119 | 2015 | 2025 | 1.0000 | 1.6430 | 1.2719 |

| Local Search Greedy, 2-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 110 | 109 | 109 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 5 | 16 | 11 | 1.0000 | 1.0054 | 1.0032 |
| 3 | 8 | 22 | 32 | 1.0000 | 1.0406 | 1.0376 |
| 4 | 17 | 37 | 49 | 1.0000 | 1.0111 | 1.0109 |
| 5 | 25 | 79 | 88 | 1.0000 | 1.2146 | 1.1783 |
| 6 | 34 | 99 | 124 | 1.0000 | 1.1973 | 1.1730 |
| 7 | 34 | 133 | 171 | 1.0000 | 1.4234 | 1.4365 |
| 8 | 39 | 229 | 268 | 1.0000 | 1.3715 | 1.3893 |
| 9 | 57 | 303 | 324 | 1.0000 | 1.3953 | 1.4289 |
| 10 | 69 | 359 | 466 | 1.0000 | 1.8821 | 1.8606 |
| 11 | 67 | 397 | 503 | 1.0000 | 1.6270 | 1.6554 |
| 12 | 82 | 586 | 693 | 1.0000 | 1.7044 | 1.6869 |
| 13 | 89 | 690 | 811 | 1.0000 | 1.7641 | 1.7524 |
| 14 | 109 | 691 | 836 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 120 | 891 | 1080 | 1.0000 | 1.8741 | 1.9265 |
| 16 | 128 | 1097 | 1271 | 1.0000 | 1.8626 | 1.9323 |
| 17 | 144 | 1280 | 1441 | 1.0000 | 2.3521 | 2.4389 |
| 18 | 153 | 1379 | 1570 | 1.0000 | 2.2029 | 2.2563 |
| 19 | 184 | 1618 | 1756 | 1.0000 | 1.8316 | 1.8839 |
| 20 | 196 | 1828 | 1937 | 1.0000 | 2.0142 | 2.0940 |

| Local Search Greedy, 2-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 109 | 107 | 109 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 208 | 208 | 207 | 1.0000 | 1.0010 | 1.0010 |
| 3 | 308 | 307 | 306 | 1.0000 | 1.0019 | 1.0021 |
| 4 | 405 | 405 | 407 | 1.0000 | 1.0007 | 1.0005 |
| 5 | 504 | 502 | 504 | 1.0000 | 1.0052 | 1.0105 |
| 6 | 601 | 603 | 603 | 1.0000 | 1.0433 | 1.0131 |
| 7 | 701 | 701 | 703 | 1.0000 | 1.1286 | 1.0360 |
| 8 | 801 | 802 | 803 | 1.0000 | 1.2096 | 1.0688 |
| 9 | 901 | 902 | 903 | 1.0000 | 1.2952 | 1.1217 |
| 10 | 1002 | 1003 | 1006 | 1.0000 | 1.6259 | 1.3043 |
| 11 | 1102 | 1102 | 1103 | 1.0000 | 1.5095 | 1.3510 |
| 12 | 1202 | 1202 | 1203 | 1.0000 | 1.6336 | 1.3043 |
| 13 | 1302 | 1303 | 1305 | 1.0000 | 1.6882 | 1.4081 |
| 14 | 1402 | 1403 | 1404 | 1.0000 | 1.6195 | 1.4922 |
| 15 | 1502 | 1505 | 1506 | 1.0000 | 1.8185 | 1.6290 |
| 16 | 1602 | 1608 | 1610 | 1.0000 | 1.8165 | 1.7287 |
| 17 | 1702 | 1708 | 1705 | 1.0000 | 2.2713 | 2.3357 |
| 18 | 1803 | 1807 | 1811 | 1.0000 | 2.1791 | 2.2292 |
| 19 | 1903 | 1905 | 1908 | 1.0000 | 1.8293 | 1.8839 |
| 20 | 2003 | 2004 | 2007 | 1.0000 | 2.0142 | 2.0939 |

| Local Search Greedy, 3-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 108 | 109 | 109 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 206 | 207 | 207 | 1.0000 | 1.0054 | 1.0032 |
| 3 | 16 | 41 | 46 | 1.0000 | 1.0406 | 1.0374 |
| 4 | 20 | 70 | 88 | 1.0000 | 1.0111 | 1.0109 |
| 5 | 30 | 167 | 176 | 1.0000 | 1.2146 | 1.1780 |
| 6 | 37 | 228 | 272 | 1.0000 | 1.1973 | 1.1730 |
| 7 | 44 | 346 | 419 | 1.0000 | 1.4234 | 1.4362 |
| 8 | 59 | 553 | 584 | 1.0000 | 1.3715 | 1.3893 |
| 9 | 67 | 766 | 822 | 1.0000 | 1.3953 | 1.4289 |
| 10 | 78 | 973 | 998 | 1.0000 | 1.8821 | 1.8604 |
| 11 | 97 | 1098 | 1100 | 1.0000 | 1.6270 | 1.6552 |
| 12 | 111 | 1204 | 1202 | 1.0000 | 1.7044 | 1.6867 |
| 13 | 131 | 1302 | 1302 | 1.0000 | 1.7641 | 1.7521 |
| 14 | 134 | 1402 | 1402 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 166 | 1505 | 1505 | 1.0000 | 1.8741 | 1.9260 |
| 16 | 181 | 1603 | 1603 | 1.0000 | 1.8626 | 1.9323 |
| 17 | 210 | 1702 | 1703 | 1.0000 | 2.3521 | 2.4388 |
| 18 | 218 | 1803 | 1803 | 1.0000 | 2.2029 | 2.2557 |
| 19 | 243 | 1904 | 1903 | 1.0000 | 1.8316 | 1.8839 |
| 20 | 264 | 2010 | 2003 | 1.0000 | 2.0142 | 2.0938 |

| Local Search Greedy, 3-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 109 | 109 | 108 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 208 | 213 | 210 | 1.0000 | 1.0010 | 1.0010 |
| 3 | 310 | 306 | 306 | 1.0000 | 1.0019 | 1.0021 |
| 4 | 405 | 405 | 404 | 1.0000 | 1.0009 | 1.0005 |
| 5 | 502 | 502 | 502 | 1.0000 | 1.0807 | 1.0228 |
| 6 | 603 | 602 | 602 | 1.0000 | 1.1104 | 1.0423 |
| 7 | 701 | 701 | 701 | 1.0000 | 1.3606 | 1.1751 |
| 8 | 801 | 801 | 805 | 1.0000 | 1.3373 | 1.2321 |
| 9 | 901 | 904 | 902 | 1.0000 | 1.3784 | 1.4205 |
| 10 | 1001 | 1001 | 1001 | 1.0000 | 1.8821 | 1.8604 |
| 11 | 1102 | 1102 | 1103 | 1.0000 | 1.6270 | 1.6552 |
| 12 | 1202 | 1202 | 1202 | 1.0000 | 1.7044 | 1.6867 |
| 13 | 1302 | 1304 | 1302 | 1.0000 | 1.7641 | 1.7521 |
| 14 | 1402 | 1402 | 1404 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 1502 | 1503 | 1506 | 1.0000 | 1.8741 | 1.9260 |
| 16 | 1602 | 1604 | 1608 | 1.0000 | 1.8626 | 1.9323 |
| 17 | 1702 | 1703 | 1708 | 1.0000 | 2.3521 | 2.4388 |
| 18 | 1802 | 1807 | 1804 | 1.0000 | 2.2029 | 2.2560 |
| 19 | 1903 | 1904 | 1904 | 1.0000 | 1.8316 | 1.8839 |
| 20 | 2003 | 2004 | 2005 | 1.0000 | 2.0142 | 2.0939 |

| Local Search Random, 1-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 9 | 6 | 8 | 1.0000 | 6.1436 | 6.1917 |
| 2 | 16 | 16 | 14 | 1.0000 | 2.1272 | 2.0544 |
| 3 | 21 | 31 | 38 | 1.0000 | 1.3805 | 1.4077 |
| 4 | 17 | 28 | 29 | 1.0000 | 2.1467 | 2.1822 |
| 5 | 32 | 53 | 53 | 1.0000 | 1.4260 | 1.4177 |
| 6 | 43 | 62 | 84 | 1.0000 | 3.8151 | 3.8567 |
| 7 | 50 | 80 | 85 | 1.0000 | 1.2828 | 1.3474 |
| 8 | 60 | 95 | 115 | 1.0000 | 1.5777 | 1.6576 |
| 9 | 76 | 126 | 142 | 1.0000 | 1.4841 | 1.5118 |
| 10 | 89 | 142 | 181 | 1.0000 | 1.5982 | 1.5438 |
| 11 | 126 | 180 | 193 | 1.0000 | 1.5216 | 1.6027 |
| 12 | 107 | 181 | 226 | 1.0000 | 1.4545 | 1.5260 |
| 13 | 118 | 233 | 285 | 1.0000 | 1.4946 | 1.4616 |
| 14 | 127 | 239 | 315 | 1.0000 | 1.4222 | 1.3794 |
| 15 | 164 | 288 | 361 | 1.0000 | 1.5525 | 1.7983 |
| 16 | 201 | 342 | 416 | 1.0000 | 1.7488 | 1.7146 |
| 17 | 240 | 375 | 479 | 1.0000 | 1.9978 | 2.1367 |
| 18 | 193 | 391 | 500 | 1.0000 | 1.8093 | 1.7729 |
| 19 | 258 | 510 | 590 | 1.0000 | 1.5989 | 1.7330 |
| 20 | 286 | 471 | 612 | 1.0000 | 1.4749 | 1.5016 |

| Local Search Random, 1-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 2 | 6 | 6 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 11 | 97 | 147 | 1.0000 | 1.0010 | 1.5109 |
| 3 | 13 | 286 | 312 | 1.0000 | 1.1731 | 1.2045 |
| 4 | 25 | 407 | 407 | 1.0000 | 2.0012 | 1.5809 |
| 5 | 37 | 507 | 510 | 1.0000 | 1.1879 | 1.3908 |
| 6 | 43 | 604 | 606 | 1.0000 | 1.2348 | 1.1600 |
| 7 | 46 | 705 | 708 | 1.0000 | 1.0493 | 1.0191 |
| 8 | 76 | 807 | 808 | 1.0000 | 1.0509 | 1.1040 |
| 9 | 73 | 906 | 912 | 1.0000 | 1.0477 | 1.0625 |
| 10 | 92 | 1008 | 1011 | 1.0000 | 1.0523 | 1.0272 |
| 11 | 101 | 1116 | 1112 | 1.0000 | 1.0612 | 1.0417 |
| 12 | 117 | 1212 | 1208 | 1.0000 | 1.1345 | 1.0527 |
| 13 | 145 | 1315 | 1315 | 1.0000 | 1.2740 | 1.0929 |
| 14 | 119 | 1416 | 1420 | 1.0000 | 1.1669 | 1.0931 |
| 15 | 153 | 1513 | 1524 | 1.0000 | 1.2975 | 1.1600 |
| 16 | 192 | 1618 | 1621 | 1.0000 | 1.4446 | 1.1425 |
| 17 | 213 | 1715 | 1718 | 1.0000 | 1.3549 | 1.1904 |
| 18 | 254 | 1822 | 1815 | 1.0000 | 1.5485 | 1.3145 |
| 19 | 269 | 1922 | 1923 | 1.0000 | 1.4193 | 1.3021 |
| 20 | 268 | 2020 | 2023 | 1.0000 | 1.4338 | 1.1854 |

| Local Search Random, 2-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 109 | 110 | 109 | 1.0000 | 4.9320 | 4.9875 |
| 2 | 9 | 29 | 25 | 1.0000 | 1.4046 | 1.2996 |
| 3 | 21 | 51 | 67 | 1.0000 | 1.6496 | 1.8511 |
| 4 | 37 | 89 | 91 | 1.0000 | 2.0057 | 1.8681 |
| 5 | 28 | 145 | 156 | 1.0000 | 1.6023 | 1.6168 |
| 6 | 54 | 197 | 194 | 1.0000 | 1.3332 | 1.3243 |
| 7 | 62 | 269 | 309 | 1.0000 | 1.3107 | 1.2198 |
| 8 | 70 | 435 | 445 | 1.0000 | 1.6471 | 1.6309 |
| 9 | 98 | 491 | 568 | 1.0000 | 1.4568 | 1.4873 |
| 10 | 129 | 690 | 734 | 1.0000 | 1.5816 | 1.5552 |
| 11 | 124 | 854 | 833 | 1.0000 | 1.3423 | 1.3358 |
| 12 | 135 | 998 | 995 | 1.0000 | 1.3281 | 1.3981 |
| 13 | 166 | 1173 | 1137 | 1.0000 | 1.4840 | 1.6919 |
| 14 | 214 | 1201 | 1279 | 1.0000 | 1.3377 | 1.3969 |
| 15 | 239 | 1321 | 1414 | 1.0000 | 1.6538 | 1.8205 |
| 16 | 236 | 1494 | 1523 | 1.0000 | 1.7688 | 1.6539 |
| 17 | 270 | 1660 | 1685 | 1.0000 | 1.7618 | 1.8067 |
| 18 | 308 | 1790 | 1799 | 1.0000 | 1.9055 | 1.8219 |
| 19 | 311 | 1908 | 1905 | 1.0000 | 1.4817 | 1.4131 |
| 20 | 370 | 2005 | 2005 | 1.0000 | 1.5055 | 1.5250 |

| Local Search Random, 2-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 109 | 110 | 109 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 208 | 211 | 208 | 1.0000 | 1.1474 | 1.2336 |
| 3 | 307 | 311 | 306 | 1.0000 | 1.2610 | 1.3419 |
| 4 | 404 | 405 | 406 | 1.0000 | 1.7933 | 1.7395 |
| 5 | 504 | 503 | 504 | 1.0000 | 1.3682 | 1.4488 |
| 6 | 601 | 602 | 605 | 1.0000 | 1.2235 | 1.1572 |
| 7 | 701 | 701 | 702 | 1.0000 | 1.2008 | 1.1267 |
| 8 | 801 | 802 | 803 | 1.0000 | 1.3054 | 1.3820 |
| 9 | 902 | 903 | 902 | 1.0000 | 1.1539 | 1.1538 |
| 10 | 1001 | 1003 | 1005 | 1.0000 | 1.4652 | 1.2823 |
| 11 | 1102 | 1104 | 1105 | 1.0000 | 1.2341 | 1.2724 |
| 12 | 1202 | 1204 | 1206 | 1.0000 | 1.4443 | 1.2303 |
| 13 | 1302 | 1310 | 1304 | 1.0000 | 1.5409 | 1.3431 |
| 14 | 1402 | 1406 | 1404 | 1.0000 | 1.3308 | 1.3180 |
| 15 | 1502 | 1511 | 1505 | 1.0000 | 1.5372 | 1.4540 |
| 16 | 1602 | 1607 | 1607 | 1.0000 | 1.6670 | 1.8297 |
| 17 | 1702 | 1710 | 1712 | 1.0000 | 1.8310 | 1.9193 |
| 18 | 1803 | 1814 | 1810 | 1.0000 | 1.7632 | 1.6469 |
| 19 | 1903 | 1913 | 1908 | 1.0000 | 1.5964 | 1.6115 |
| 20 | 2003 | 2007 | 2015 | 1.0000 | 1.7602 | 1.5760 |

| Local Search Random, 3-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 109 | 110 | 109 | 1.0000 | 6.4785 | 6.4883 |
| 2 | 207 | 208 | 207 | 1.0000 | 1.5458 | 2.8852 |
| 3 | 35 | 66 | 76 | 1.0000 | 1.4737 | 1.7049 |
| 4 | 34 | 134 | 153 | 1.0000 | 2.1157 | 1.9957 |
| 5 | 49 | 249 | 276 | 1.0000 | 1.6322 | 1.5352 |
| 6 | 61 | 387 | 416 | 1.0000 | 1.3085 | 1.3586 |
| 7 | 79 | 551 | 577 | 1.0000 | 1.4433 | 1.5541 |
| 8 | 104 | 704 | 749 | 1.0000 | 1.4142 | 1.3438 |
| 9 | 98 | 887 | 892 | 1.0000 | 1.3145 | 1.3619 |
| 10 | 139 | 1003 | 1003 | 1.0000 | 1.3927 | 1.4677 |
| 11 | 137 | 1103 | 1103 | 1.0000 | 1.4000 | 1.4186 |
| 12 | 184 | 1203 | 1204 | 1.0000 | 1.4649 | 1.4659 |
| 13 | 191 | 1303 | 1302 | 1.0000 | 1.5913 | 1.5644 |
| 14 | 218 | 1405 | 1402 | 1.0000 | 1.4686 | 1.4989 |
| 15 | 230 | 1503 | 1506 | 1.0000 | 1.5209 | 1.6787 |
| 16 | 295 | 1605 | 1604 | 1.0000 | 1.7594 | 1.5692 |
| 17 | 318 | 1707 | 1704 | 1.0000 | 1.8844 | 1.9019 |
| 18 | 375 | 1804 | 1805 | 1.0000 | 1.9646 | 1.9968 |
| 19 | 377 | 1909 | 1914 | 1.0000 | 1.7138 | 1.7275 |
| 20 | 430 | 2004 | 2006 | 1.0000 | 1.5970 | 1.8136 |

| Local Search Random, 3-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 109 | 109 | 109 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 208 | 210 | 208 | 1.0000 | 1.0010 | 1.1474 |
| 3 | 307 | 305 | 306 | 1.0000 | 1.2736 | 1.3079 |
| 4 | 405 | 405 | 405 | 1.0000 | 1.5475 | 1.5479 |
| 5 | 503 | 503 | 503 | 1.0000 | 1.4913 | 1.3799 |
| 6 | 602 | 605 | 606 | 1.0000 | 3.9041 | 3.6743 |
| 7 | 701 | 703 | 702 | 1.0000 | 1.3423 | 1.2977 |
| 8 | 801 | 802 | 801 | 1.0000 | 1.3431 | 1.3011 |
| 9 | 901 | 904 | 903 | 1.0000 | 1.4081 | 1.3557 |
| 10 | 1001 | 1004 | 1002 | 1.0000 | 1.5119 | 1.6262 |
| 11 | 1101 | 1109 | 1102 | 1.0000 | 1.4977 | 1.4289 |
| 12 | 1202 | 1204 | 1203 | 1.0000 | 1.3007 | 1.5158 |
| 13 | 1302 | 1303 | 1308 | 1.0000 | 1.6815 | 1.5274 |
| 14 | 1402 | 1403 | 1407 | 1.0000 | 1.4398 | 1.4447 |
| 15 | 1502 | 1507 | 1504 | 1.0000 | 1.5910 | 1.8003 |
| 16 | 1602 | 1608 | 1609 | 1.0000 | 1.6395 | 1.5414 |
| 17 | 1702 | 1711 | 1710 | 1.0000 | 1.8605 | 1.7009 |
| 18 | 1803 | 1809 | 1816 | 1.0000 | 1.7357 | 2.0322 |
| 19 | 1903 | 1907 | 1908 | 1.0000 | 1.4587 | 1.7022 |
| 20 | 2003 | 2015 | 2010 | 1.0000 | 1.3709 | 1.7289 |

| Local Search Software, 1-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 2 | 2 | 1 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 3 | 6 | 3 | 1.0000 | 1.0077 | 1.0077 |
| 3 | 5 | 7 | 10 | 1.0000 | 1.0406 | 1.0416 |
| 4 | 13 | 11 | 16 | 1.0000 | 1.0111 | 1.0142 |
| 5 | 12 | 19 | 17 | 1.0000 | 1.3256 | 1.3199 |
| 6 | 16 | 19 | 23 | 1.0000 | 1.2909 | 1.3141 |
| 7 | 19 | 24 | 30 | 1.0000 | 1.4965 | 1.5276 |
| 8 | 19 | 39 | 45 | 1.0000 | 1.5961 | 1.6193 |
| 9 | 24 | 47 | 61 | 1.0000 | 1.6310 | 1.6699 |
| 10 | 36 | 46 | 63 | 1.0000 | 2.0261 | 2.0814 |
| 11 | 39 | 59 | 80 | 1.0000 | 1.6795 | 1.7276 |
| 12 | 49 | 64 | 85 | 1.0000 | 1.9698 | 2.0411 |
| 13 | 42 | 73 | 99 | 1.0000 | 1.9837 | 2.0499 |
| 14 | 50 | 82 | 101 | 1.0000 | 1.6513 | 1.7101 |
| 15 | 58 | 102 | 122 | 1.0000 | 2.0283 | 2.1099 |
| 16 | 73 | 112 | 149 | 1.0000 | 2.2970 | 2.3929 |
| 17 | 84 | 115 | 189 | 1.0000 | 2.6258 | 2.7721 |
| 18 | 85 | 131 | 185 | 1.0000 | 2.5902 | 2.7023 |
| 19 | 99 | 144 | 188 | 1.0000 | 2.0090 | 2.1191 |
| 20 | 96 | 156 | 193 | 1.0000 | 2.1259 | 2.2304 |

| Local Search Software, 1-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 1 | 6 | 13 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 4 | 88 | 142 | 1.0000 | 1.0010 | 1.0010 |
| 3 | 6 | 305 | 307 | 1.0000 | 1.0019 | 1.0019 |
| 4 | 7 | 407 | 408 | 1.0000 | 1.0006 | 1.0005 |
| 5 | 12 | 505 | 506 | 1.0000 | 1.0043 | 1.0043 |
| 6 | 11 | 604 | 604 | 1.0000 | 1.0022 | 1.0052 |
| 7 | 16 | 704 | 705 | 1.0000 | 1.0047 | 1.0068 |
| 8 | 32 | 805 | 807 | 1.0000 | 1.0100 | 1.0132 |
| 9 | 30 | 905 | 910 | 1.0000 | 1.0101 | 1.0124 |
| 10 | 31 | 1013 | 1012 | 1.0000 | 1.0575 | 1.0259 |
| 11 | 38 | 1106 | 1116 | 1.0000 | 1.1015 | 1.0265 |
| 12 | 40 | 1212 | 1217 | 1.0000 | 1.2008 | 1.0393 |
| 13 | 42 | 1317 | 1314 | 1.0000 | 1.2761 | 1.0581 |
| 14 | 50 | 1419 | 1412 | 1.0000 | 1.2703 | 1.0568 |
| 15 | 66 | 1510 | 1523 | 1.0000 | 1.3901 | 1.0864 |
| 16 | 65 | 1623 | 1616 | 1.0000 | 1.5716 | 1.1873 |
| 17 | 67 | 1724 | 1722 | 1.0000 | 1.8307 | 1.2827 |
| 18 | 77 | 1811 | 1827 | 1.0000 | 1.8574 | 1.3286 |
| 19 | 87 | 1923 | 1922 | 1.0000 | 1.6401 | 1.2242 |
| 20 | 93 | 2026 | 2020 | 1.0000 | 1.7111 | 1.2645 |

| Local Search Software, 2-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 109 | 109 | 109 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 6 | 20 | 9 | 1.0000 | 1.0077 | 1.0055 |
| 3 | 12 | 18 | 29 | 1.0000 | 1.0406 | 1.0376 |
| 4 | 19 | 37 | 43 | 1.0000 | 1.0111 | 1.0109 |
| 5 | 21 | 68 | 69 | 1.0000 | 1.3197 | 1.2891 |
| 6 | 21 | 88 | 108 | 1.0000 | 1.2895 | 1.2687 |
| 7 | 29 | 120 | 155 | 1.0000 | 1.4950 | 1.4987 |
| 8 | 45 | 210 | 212 | 1.0000 | 1.5929 | 1.6066 |
| 9 | 49 | 245 | 283 | 1.0000 | 1.6283 | 1.6487 |
| 10 | 56 | 308 | 427 | 1.0000 | 2.0248 | 2.0530 |
| 11 | 60 | 365 | 478 | 1.0000 | 1.6786 | 1.7071 |
| 12 | 72 | 526 | 654 | 1.0000 | 1.9646 | 2.0053 |
| 13 | 77 | 631 | 801 | 1.0000 | 1.9804 | 2.0143 |
| 14 | 84 | 683 | 826 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 110 | 849 | 1073 | 1.0000 | 2.0266 | 2.0798 |
| 16 | 111 | 1025 | 1242 | 1.0000 | 2.2949 | 2.3666 |
| 17 | 131 | 1214 | 1437 | 1.0000 | 2.6241 | 2.7468 |
| 18 | 139 | 1336 | 1562 | 1.0000 | 2.5885 | 2.6808 |
| 19 | 149 | 1578 | 1726 | 1.0000 | 2.0068 | 2.1020 |
| 20 | 165 | 1757 | 1890 | 1.0000 | 2.1259 | 2.2108 |

| Local Search Software, 2-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 109 | 108 | 109 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 208 | 208 | 208 | 1.0000 | 1.0010 | 1.0010 |
| 3 | 307 | 307 | 307 | 1.0000 | 1.0019 | 1.0021 |
| 4 | 405 | 405 | 407 | 1.0000 | 1.0007 | 1.0005 |
| 5 | 505 | 504 | 504 | 1.0000 | 1.0052 | 1.0097 |
| 6 | 603 | 604 | 604 | 1.0000 | 1.0538 | 1.0128 |
| 7 | 701 | 701 | 702 | 1.0000 | 1.1388 | 1.0393 |
| 8 | 801 | 801 | 802 | 1.0000 | 1.3320 | 1.1131 |
| 9 | 901 | 902 | 904 | 1.0000 | 1.4314 | 1.1272 |
| 10 | 1002 | 1002 | 1004 | 1.0000 | 1.6996 | 1.3057 |
| 11 | 1102 | 1103 | 1104 | 1.0000 | 1.5514 | 1.3582 |
| 12 | 1202 | 1202 | 1206 | 1.0000 | 1.8354 | 1.4823 |
| 13 | 1303 | 1304 | 1304 | 1.0000 | 1.8671 | 1.5211 |
| 14 | 1402 | 1403 | 1404 | 1.0000 | 1.6195 | 1.4922 |
| 15 | 1502 | 1506 | 1506 | 1.0000 | 1.9430 | 1.7372 |
| 16 | 1602 | 1605 | 1611 | 1.0000 | 2.1769 | 1.9052 |
| 17 | 1702 | 1706 | 1705 | 1.0000 | 2.5094 | 2.5745 |
| 18 | 1803 | 1807 | 1808 | 1.0000 | 2.5446 | 2.5986 |
| 19 | 1903 | 1905 | 1904 | 1.0000 | 2.0068 | 2.1020 |
| 20 | 2003 | 2003 | 2007 | 1.0000 | 2.1259 | 2.2107 |

| Local Search Software, 3-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 109 | 109 | 109 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 208 | 208 | 207 | 1.0000 | 1.0077 | 1.0055 |
| 3 | 18 | 48 | 46 | 1.0000 | 1.0406 | 1.0374 |
| 4 | 24 | 73 | 83 | 1.0000 | 1.0111 | 1.0109 |
| 5 | 27 | 136 | 161 | 1.0000 | 1.3197 | 1.2888 |
| 6 | 30 | 234 | 268 | 1.0000 | 1.2895 | 1.2687 |
| 7 | 39 | 325 | 406 | 1.0000 | 1.4950 | 1.4985 |
| 8 | 56 | 531 | 578 | 1.0000 | 1.5929 | 1.6066 |
| 9 | 63 | 742 | 810 | 1.0000 | 1.6283 | 1.6487 |
| 10 | 74 | 972 | 991 | 1.0000 | 2.0248 | 2.0527 |
| 11 | 81 | 1099 | 1101 | 1.0000 | 1.6786 | 1.7070 |
| 12 | 96 | 1202 | 1207 | 1.0000 | 1.9646 | 2.0051 |
| 13 | 106 | 1304 | 1302 | 1.0000 | 1.9804 | 2.0140 |
| 14 | 129 | 1402 | 1402 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 143 | 1504 | 1504 | 1.0000 | 2.0266 | 2.0792 |
| 16 | 164 | 1604 | 1603 | 1.0000 | 2.2949 | 2.3666 |
| 17 | 181 | 1703 | 1702 | 1.0000 | 2.6241 | 2.7466 |
| 18 | 195 | 1803 | 1805 | 1.0000 | 2.5885 | 2.6803 |
| 19 | 230 | 1904 | 1903 | 1.0000 | 2.0068 | 2.1020 |
| 20 | 236 | 2005 | 2010 | 1.0000 | 2.1259 | 2.2106 |

| Local Search Software, 3-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 108 | 109 | 109 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 207 | 207 | 208 | 1.0000 | 1.0010 | 1.0010 |
| 3 | 307 | 307 | 308 | 1.0000 | 1.0019 | 1.0021 |
| 4 | 405 | 405 | 406 | 1.0000 | 1.0009 | 1.0005 |
| 5 | 501 | 502 | 503 | 1.0000 | 1.1179 | 1.0274 |
| 6 | 603 | 602 | 603 | 1.0000 | 1.1850 | 1.0429 |
| 7 | 701 | 701 | 701 | 1.0000 | 1.4076 | 1.2179 |
| 8 | 801 | 801 | 802 | 1.0000 | 1.4825 | 1.3436 |
| 9 | 901 | 901 | 901 | 1.0000 | 1.6114 | 1.6324 |
| 10 | 1001 | 1005 | 1003 | 1.0000 | 2.0248 | 2.0527 |
| 11 | 1102 | 1107 | 1102 | 1.0000 | 1.6786 | 1.7070 |
| 12 | 1202 | 1202 | 1202 | 1.0000 | 1.9646 | 2.0051 |
| 13 | 1302 | 1302 | 1302 | 1.0000 | 1.9804 | 2.0140 |
| 14 | 1402 | 1402 | 1402 | 1.0000 | 1.6513 | 1.7069 |
| 15 | 1502 | 1502 | 1503 | 1.0000 | 2.0266 | 2.0792 |
| 16 | 1602 | 1604 | 1603 | 1.0000 | 2.2949 | 2.3666 |
| 17 | 1702 | 1703 | 1703 | 1.0000 | 2.6241 | 2.7466 |
| 18 | 1803 | 1805 | 1805 | 1.0000 | 2.5885 | 2.6806 |
| 19 | 1903 | 1905 | 1908 | 1.0000 | 2.0068 | 2.1020 |
| 20 | 2003 | 2004 | 2009 | 1.0000 | 2.1259 | 2.2107 |

| Local Search Hardware, 1-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 1 | 1 | 2 | 1.0000 | 15.2395 | 15.2395 |
| 2 | 6 | 11 | 8 | 1.0000 | 5.8548 | 5.8460 |
| 3 | 14 | 11 | 18 | 1.0000 | 2.3469 | 2.3382 |
| 4 | 9 | 25 | 37 | 1.0000 | 2.4913 | 2.4771 |
| 5 | 17 | 32 | 64 | 1.0000 | 1.7375 | 1.7291 |
| 6 | 20 | 43 | 89 | 1.0000 | 3.9605 | 3.9559 |
| 7 | 18 | 70 | 112 | 1.0000 | 1.3274 | 1.3343 |
| 8 | 34 | 90 | 155 | 1.0000 | 1.4871 | 1.4923 |
| 9 | 33 | 100 | 198 | 1.0000 | 1.2079 | 1.2193 |
| 10 | 31 | 122 | 242 | 1.0000 | 1.2076 | 1.2204 |
| 11 | 38 | 153 | 271 | 1.0000 | 1.2191 | 1.2322 |
| 12 | 41 | 188 | 346 | 1.0000 | 1.2218 | 1.2488 |
| 13 | 49 | 203 | 411 | 1.0000 | 1.2802 | 1.3038 |
| 14 | 50 | 221 | 452 | 1.0000 | 1.1974 | 1.2196 |
| 15 | 55 | 283 | 513 | 1.0000 | 1.2907 | 1.3188 |
| 16 | 70 | 317 | 623 | 1.0000 | 1.3286 | 1.3679 |
| 17 | 70 | 351 | 716 | 1.0000 | 1.2506 | 1.2964 |
| 18 | 84 | 425 | 755 | 1.0000 | 1.3028 | 1.3458 |
| 19 | 85 | 443 | 859 | 1.0000 | 1.2105 | 1.2507 |
| 20 | 96 | 482 | 893 | 1.0000 | 1.2367 | 1.2749 |

| Local Search Hardware, 1-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 1 | 8 | 5 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 2 | 87 | 154 | 1.0000 | 1.8381 | 1.8369 |
| 3 | 11 | 289 | 307 | 1.0000 | 1.3816 | 1.5380 |
| 4 | 6 | 407 | 408 | 1.0000 | 2.3503 | 2.3425 |
| 5 | 10 | 508 | 508 | 1.0000 | 1.5246 | 1.5203 |
| 6 | 20 | 605 | 606 | 1.0000 | 3.8219 | 3.8158 |
| 7 | 20 | 706 | 708 | 1.0000 | 1.1601 | 1.1619 |
| 8 | 20 | 806 | 808 | 1.0000 | 1.3142 | 1.3156 |
| 9 | 27 | 907 | 907 | 1.0000 | 1.0841 | 1.0890 |
| 10 | 32 | 1011 | 1015 | 1.0000 | 1.0500 | 1.0614 |
| 11 | 32 | 1113 | 1114 | 1.0000 | 1.0755 | 1.0982 |
| 12 | 44 | 1215 | 1218 | 1.0000 | 1.0798 | 1.1155 |
| 13 | 50 | 1318 | 1317 | 1.0000 | 1.1465 | 1.1872 |
| 14 | 55 | 1420 | 1432 | 1.0000 | 1.0927 | 1.1314 |
| 15 | 54 | 1520 | 1524 | 1.0000 | 1.1797 | 1.2292 |
| 16 | 75 | 1627 | 1625 | 1.0000 | 1.2163 | 1.2788 |
| 17 | 73 | 1734 | 1734 | 1.0000 | 1.1513 | 1.2168 |
| 18 | 83 | 1821 | 1826 | 1.0000 | 1.2083 | 1.2702 |
| 19 | 97 | 1930 | 1947 | 1.0000 | 1.1386 | 1.1986 |
| 20 | 97 | 2029 | 2035 | 1.0000 | 1.1651 | 1.2195 |

| Local Search Hardware, 2-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 109 | 110 | 109 | 1.0000 | 15.2395 | 15.2395 |
| 2 | 8 | 23 | 22 | 1.0000 | 5.5615 | 5.5543 |
| 3 | 7 | 52 | 65 | 1.0000 | 2.2175 | 2.2129 |
| 4 | 14 | 105 | 123 | 1.0000 | 2.4109 | 2.4131 |
| 5 | 19 | 197 | 223 | 1.0000 | 1.6649 | 1.6747 |
| 6 | 23 | 293 | 363 | 1.0000 | 3.9029 | 3.9130 |
| 7 | 28 | 435 | 539 | 1.0000 | 1.2730 | 1.2986 |
| 8 | 36 | 656 | 750 | 1.0000 | 1.4374 | 1.4622 |
| 9 | 42 | 862 | 897 | 1.0000 | 1.1678 | 1.1963 |
| 10 | 48 | 997 | 1003 | 1.0000 | 1.1629 | 1.1984 |
| 11 | 62 | 1096 | 1103 | 1.0000 | 1.1803 | 1.2095 |
| 12 | 73 | 1207 | 1203 | 1.0000 | 1.1834 | 1.2287 |
| 13 | 76 | 1304 | 1306 | 1.0000 | 1.2394 | 1.2819 |
| 14 | 93 | 1404 | 1406 | 1.0000 | 1.1652 | 1.2007 |
| 15 | 96 | 1504 | 1506 | 1.0000 | 1.2570 | 1.2988 |
| 16 | 124 | 1605 | 1607 | 1.0000 | 1.2911 | 1.3478 |
| 17 | 130 | 1709 | 1711 | 1.0000 | 1.2157 | 1.2800 |
| 18 | 143 | 1807 | 1808 | 1.0000 | 1.2704 | 1.3290 |
| 19 | 166 | 1906 | 1906 | 1.0000 | 1.1825 | 1.2352 |
| 20 | 174 | 2007 | 2011 | 1.0000 | 1.2111 | 1.2589 |

| Local Search Hardware, 2-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 109 | 109 | 108 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 207 | 207 | 210 | 1.0000 | 1.8381 | 1.8369 |
| 3 | 306 | 307 | 306 | 1.0000 | 1.3816 | 1.5382 |
| 4 | 405 | 406 | 407 | 1.0000 | 2.3503 | 2.3439 |
| 5 | 503 | 503 | 504 | 1.0000 | 1.5600 | 1.5584 |
| 6 | 602 | 602 | 605 | 1.0000 | 3.8331 | 3.8370 |
| 7 | 701 | 702 | 704 | 1.0000 | 1.1965 | 1.2198 |
| 8 | 801 | 803 | 805 | 1.0000 | 1.3850 | 1.4122 |
| 9 | 901 | 907 | 905 | 1.0000 | 1.1334 | 1.1629 |
| 10 | 1001 | 1005 | 1006 | 1.0000 | 1.1266 | 1.1788 |
| 11 | 1102 | 1106 | 1110 | 1.0000 | 1.1586 | 1.2028 |
| 12 | 1202 | 1209 | 1208 | 1.0000 | 1.1716 | 1.2247 |
| 13 | 1302 | 1306 | 1312 | 1.0000 | 1.2394 | 1.2819 |
| 14 | 1402 | 1407 | 1411 | 1.0000 | 1.1642 | 1.2007 |
| 15 | 1502 | 1512 | 1518 | 1.0000 | 1.2570 | 1.2988 |
| 16 | 1602 | 1610 | 1614 | 1.0000 | 1.2911 | 1.3478 |
| 17 | 1702 | 1716 | 1718 | 1.0000 | 1.2157 | 1.2800 |
| 18 | 1803 | 1819 | 1820 | 1.0000 | 1.2704 | 1.3290 |
| 19 | 1903 | 1920 | 1928 | 1.0000 | 1.1825 | 1.2352 |
| 20 | 2003 | 2017 | 2022 | 1.0000 | 1.2111 | 1.2589 |

| Local Search Hardware, 3-Opt, Steepest | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 109 | 109 | 109 | 1.0000 | 15.2395 | 15.2395 |
| 2 | 208 | 208 | 207 | 1.0000 | 5.5615 | 5.5543 |
| 3 | 15 | 91 | 110 | 1.0000 | 2.2175 | 2.2129 |
| 4 | 24 | 206 | 246 | 1.0000 | 2.4109 | 2.4131 |
| 5 | 23 | 411 | 451 | 1.0000 | 1.6649 | 1.6747 |
| 6 | 34 | 596 | 603 | 1.0000 | 3.9029 | 3.9130 |
| 7 | 38 | 701 | 701 | 1.0000 | 1.2730 | 1.2986 |
| 8 | 54 | 801 | 801 | 1.0000 | 1.4374 | 1.4622 |
| 9 | 65 | 903 | 902 | 1.0000 | 1.1678 | 1.1963 |
| 10 | 77 | 1002 | 1002 | 1.0000 | 1.1629 | 1.1984 |
| 11 | 92 | 1102 | 1104 | 1.0000 | 1.1803 | 1.2095 |
| 12 | 94 | 1203 | 1205 | 1.0000 | 1.1834 | 1.2287 |
| 13 | 108 | 1303 | 1306 | 1.0000 | 1.2394 | 1.2819 |
| 14 | 123 | 1405 | 1406 | 1.0000 | 1.1652 | 1.2007 |
| 15 | 146 | 1508 | 1509 | 1.0000 | 1.2570 | 1.2988 |
| 16 | 167 | 1606 | 1607 | 1.0000 | 1.2911 | 1.3478 |
| 17 | 189 | 1707 | 1713 | 1.0000 | 1.2157 | 1.2800 |
| 18 | 200 | 1807 | 1808 | 1.0000 | 1.2704 | 1.3290 |
| 19 | 228 | 1907 | 1912 | 1.0000 | 1.1825 | 1.2352 |
| 20 | 245 | 2008 | 2011 | 1.0000 | 1.2111 | 1.2589 |

| Local Search Hardware, 3-Opt, Tabu | | | | | | |
|---|---|---|---|---|---|---|
| | Runtimes | | | Solution Quality | | |
| Pipeline Size | TAC1 | TAC2 | TAC3 | TAC1 | TAC2 | TAC3 |
| 1 | 109 | 109 | 108 | 1.0000 | 1.0000 | 1.0000 |
| 2 | 207 | 207 | 208 | 1.0000 | 1.8381 | 1.8369 |
| 3 | 307 | 306 | 311 | 1.0000 | 1.3816 | 1.5704 |
| 4 | 405 | 408 | 405 | 1.0000 | 2.3516 | 2.3454 |
| 5 | 502 | 502 | 503 | 1.0000 | 1.5833 | 1.5833 |
| 6 | 602 | 602 | 603 | 1.0000 | 3.8609 | 3.8814 |
| 7 | 701 | 702 | 703 | 1.0000 | 1.2422 | 1.2924 |
| 8 | 801 | 802 | 804 | 1.0000 | 1.4321 | 1.4622 |
| 9 | 902 | 903 | 906 | 1.0000 | 1.1678 | 1.1963 |
| 10 | 1002 | 1002 | 1006 | 1.0000 | 1.1629 | 1.1984 |
| 11 | 1102 | 1103 | 1106 | 1.0000 | 1.1803 | 1.2095 |
| 12 | 1202 | 1206 | 1208 | 1.0000 | 1.1834 | 1.2287 |
| 13 | 1302 | 1306 | 1311 | 1.0000 | 1.2394 | 1.2819 |
| 14 | 1402 | 1409 | 1408 | 1.0000 | 1.1652 | 1.2007 |
| 15 | 1502 | 1508 | 1516 | 1.0000 | 1.2570 | 1.2988 |
| 16 | 1602 | 1618 | 1619 | 1.0000 | 1.2911 | 1.3478 |
| 17 | 1702 | 1711 | 1718 | 1.0000 | 1.2157 | 1.2800 |
| 18 | 1802 | 1816 | 1828 | 1.0000 | 1.2704 | 1.3290 |
| 19 | 1903 | 1917 | 1923 | 1.0000 | 1.1825 | 1.2352 |
| 20 | 2003 | 2013 | 2029 | 1.0000 | 1.2111 | 1.2589 |

## D.2 Dynamo Chapter Results

| Pipeline | Solution | Estimated Runtime |
|---|---|---|
| 1 | (SOURCE:sw) → [none] → (minFilter:sw) → [none] → (SINK:sw) | 1111 |
| 2 | (SOURCE:sw) → [none] → (hist4c8b16bin:sw) → [none] → (SINK:sw) | 300 |
| 3 | (SOURCE:sw) → [none] → (maxFilter:sw) → [none] → (hist1c3b16bin:sw) → [none] → (SINK:sw) | 1392 |
| 4 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (SINK:sw) | 2910 |
| 5 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (SINK:sw) | 2920 |
| 6 | (SOURCE:sw) → [none] → (maxFilter:sw) → [none] → (emAnd:sw) → [none] → (minFilter:sw) → [none] → (SINK:sw) | 2260 |
| 7 | (SOURCE:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emOr:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (SINK:sw) | 3226 |
| 8 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out+ comm:out] → (hist4c4b1bin:sw) → [none] → (SINK:sw) | 3188 |
| 9 | (SOURCE:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (hist3c8b1bin:sw) → [none] → (SINK:sw) | 3407 |
| 10 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (hist3c8b1bin:sw) → [none] → (SINK:sw) | 3362 |

| Pipeline | Solution | Estimated Runtime |
|---|---|---|
| 11 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (minFilter:hw:PadminFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (hist1c3b1bin:hw:hist1c3b1binRight.x86) → [comm:out] → (SINK:sw) | 3740 |
| 12 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emOr:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (hist4c8b1bin:hw:hist4c8b1binRight.x86) → [comm:out] → (SINK:sw) | 3733 |
| 13 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (hist3c4b1bin:sw) → [none] → (SINK:sw) | 3884 |
| 14 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (SINK:sw) | 4128 |

| Pipeline | Solution | Estimated Runtime |
|---|---|---|
| 15 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (SINK:sw) | 4538 |
| 16 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (hist3c8b1bin:sw) → [none] → (SINK:sw) | 4182 |
| 17 | (SOURCE:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emOr:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (ed:hw:PadedRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (hist4c8b16bin:hw:hist4c8b16binRight.x86) → [comm:out] → (SINK:sw) | 4374 |
| 18 | (SOURCE:sw) → [none] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (hist3c8b1bin:sw) → [none] → (SINK:sw) | 4579 |

| Pipeline | Solution | Estimated Runtime |
|---|---|---|
| 19 | (SOURCE:sw) → [comm:in + reprg] → (minFilter:hw:PadminFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [none] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out+ comm:out] → (SINK:sw) | 4224 |
| 20 | (SOURCE:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (hist1c3b16bin:hw:hist1c3b16binRight.x86) → [comm:out] → (SINK:sw) | 4854 |
| 21 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (SINK:sw) | 5211 |

| Pipeline | Solution | Estimated Runtime |
|---|---|---|
| 22 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [none] → (emAnd:sw) → [none] → (emOr:sw) → [none] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emOr:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (hist1c3b16bin:sw) → [none] → (SINK:sw) | 4218 |
| 23 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (hist3c8b1bin:sw) → [none] → (SINK:sw) | 5260 |
| 24 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [none] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (hist1c3b16bin:sw) → [none] → (SINK:sw) | 4774 |

| Pipeline | Solution | Estimated Runtime |
|---|---|---|
| 25 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [none] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [none] → (emAnd:sw) → [none] → (SINK:sw) | 4998 |
| 26 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [none] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (hist3c4b1bin:sw) → [none] → (SINK:sw) | 4931 |
| 27 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (hist3c8b1bin:sw) → [none] → (SINK:sw) | 5554 |

| Pipeline | Solution | Estimated Runtime |
|---|---|---|
| 28 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [comm:in + reprg] → (minFilter:hw:PadminFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [comm:in + reprg] → (hist1c3b1bin:hw:hist1c3b1binRight.x86) → [comm:out] → (SINK:sw) | 5732 |
| 29 | (SOURCE:sw) → [none] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (minFilter:hw:PadminFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (SINK:sw) | 5534 |

| Pipeline | Solution | Estimated Runtime |
|---|---|---|
| 30 | (SOURCE:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [none] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (hist3c8b1bin:sw) → [none] → (SINK:sw) | 5385 |
| 31 | (SOURCE:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (ed:hw:PadedRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (hist4c8b16bin:hw:hist4c8b16binRight.x86) → [comm:out] → (SINK:sw) | 6176 |

| Pipeline | Solution | Estimated Runtime |
|---|---|---|
| 32 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (minFilter:hw:PadminFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out+ comm:out] → (hist3c8b16bin:sw) → [none] → (SINK:sw) | 6098 |
| 33 | (SOURCE:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (hist4c4b1bin:sw) → [none] → (SINK:sw) | 6315 |

| Pipeline | Solution | Estimated Runtime |
|---|---|---|
| 34 | (SOURCE:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [none] → (hist3c8b1bin:sw) → [none] → (SINK:sw) | 6501 |
| 35 | (SOURCE:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emOr:sw) → [none] → (emAnd:sw) → [none] → (emOr:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [none] → (emAnd:sw) → [comm:in + reprg] → (hist1c3b16bin:hw:hist1c3b16binRight.x86) → [comm:out] → (SINK:sw) | 6094 |

| Pipeline | Solution | Estimated Runtime |
|---|---|---|
| 36 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (minFilter:hw:PadminFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (hist1c3b16bin:sw) → [none] → (SINK:sw) | 6429 |
| 37 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emOr:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Unpad:out+ comm:out] → (hist3c4b1bin:sw) → [none] → (SINK:sw) | 6827 |

| Pipeline | Solution | Estimated Runtime |
|---|---|---|
| 38 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [none] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [comm:in + reprg] → (hist4c8b16bin:hw:hist4c8b16binRight.x86) → [comm:out] → (SINK:sw) | 6426 |
| 39 | (SOURCE:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emOr:sw) → [Pad:in + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [none] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [none] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Unpad:out + comm:out + comm:in + reprg] → (hist1c3b16bin:hw:hist1c3b16binRight.x86) → [comm:out] → (SINK:sw) | 6443 |

| Pipeline | Solution | Estimated Runtime |
|---|---|---|
| 40 | (SOURCE:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [comm:out] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [comm:out] → (emAnd:sw) → [none] → (emAnd:sw) → [Pad:in + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (maxFilter:hw:maxFilterRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [none] → (emAnd:sw) → [comm:in + reprg] → (mf:hw:PadmfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (ed:hw:edUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Fix:out + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (minFilter:hw:minFilterUnpadRight.x86) → [Pad:in + comm:out + comm:in + reprg] → (mf:hw:mfRight.x86) → [Unpad:out+ comm:out] → (emAnd:sw) → [none] → (SINK:sw) | 7305 |

| Test Number | Predicted Latency with No Overhead | Predicted Latency with Comm Costs | Predicted Latency with Comm and HW Init Costs | Predicted Latency with All Costs | Actual Latency |
|---|---|---|---|---|---|
| 1 | 1111 | 1111 | 1111 | 1111 | 1309 |
| 2 | 300 | 300 | 300 | 300 | 629 |
| 3 | 1392 | 1392 | 1392 | 1392 | 1517 |
| 4 | 360 | 512 | 2876 | 2910 | 3342 |
| 5 | 375 | 504 | 2868 | 2920 | 3169 |
| 6 | 2260 | 2260 | 2260 | 2260 | 2104 |
| 7 | 588 | 792 | 3156 | 3226 | 3482 |
| 8 | 635 | 789 | 3153 | 3188 | 3399 |
| 9 | 769 | 973 | 3337 | 3407 | 3582 |
| 10 | 743 | 947 | 3311 | 3380 | 3571 |
| 11 | 957 | 1271 | 3635 | 3740 | 4348 |
| 12 | 952 | 1265 | 3629 | 3733 | 4049 |
| 13 | 1064 | 1399 | 3763 | 3884 | 4335 |
| 14 | 1135 | 1572 | 3936 | 4128 | 4520 |
| 15 | 1411 | 2002 | 4366 | 4538 | 4789 |
| 16 | 1271 | 1679 | 4043 | 4182 | 5289 |
| 17 | 1435 | 1869 | 4233 | 4374 | 5186 |
| 18 | 1513 | 2041 | 4405 | 4579 | 5198 |
| 19 | 1308 | 1718 | 4082 | 4224 | 4803 |
| 20 | 1701 | 2317 | 4681 | 4854 | 5955 |
| 21 | 1865 | 2622 | 4986 | 5211 | 6165 |
| 22 | 1398 | 1733 | 4097 | 4218 | 4969 |
| 23 | 1960 | 2653 | 5017 | 5260 | 6932 |
| 24 | 1737 | 2255 | 4619 | 4774 | 5677 |
| 25 | 1785 | 2426 | 4790 | 4998 | 6012 |
| 26 | 1804 | 2395 | 4759 | 4931 | 7141 |
| 27 | 2199 | 2983 | 5347 | 5554 | 7246 |
| 28 | 2296 | 3143 | 5507 | 5732 | 7736 |
| 29 | 2093 | 2838 | 5202 | 5534 | 8888 |
| 30 | 2114 | 2757 | 5121 | 5385 | 8560 |
| 31 | 2601 | 3497 | 5861 | 6176 | 9871 |
| 32 | 2549 | 3438 | 5802 | 6098 | 9450 |
| 33 | 2723 | 3653 | 6017 | 6315 | 10173 |
| 34 | 2750 | 3755 | 6119 | 6501 | 12052 |
| 35 | 2575 | 3453 | 5817 | 6094 | 10922 |
| 36 | 2868 | 3765 | 6129 | 6429 | 10027 |
| 37 | 2999 | 4099 | 6463 | 6827 | 11510 |
| 38 | 2810 | 3784 | 6148 | 6426 | 10949 |
| 39 | 2857 | 3781 | 6145 | 6443 | 14383 |
| 40 | 3450 | 4555 | 6919 | 7305 | 12217 |

| Test Number | ARE with No Overhead | ARE with Comm Costs | ARE with Comm and HW Init Costs | ARE with All Costs |
|---|---|---|---|---|
| 1 | 0.1513 | 0.1513 | 0.1513 | 0.1513 |
| 2 | 0.5231 | 0.5231 | 0.5231 | 0.5231 |
| 3 | 0.0824 | 0.0824 | 0.0824 | 0.0824 |
| 4 | 0.8923 | 0.8468 | 0.1394 | 0.1293 |
| 5 | 0.8817 | 0.8410 | 0.0950 | 0.0786 |
| 6 | 0.0741 | 0.0741 | 0.0741 | 0.0741 |
| 7 | 0.8311 | 0.7725 | 0.0936 | 0.0735 |
| 8 | 0.8132 | 0.7679 | 0.0724 | 0.0621 |
| 9 | 0.7853 | 0.7284 | 0.0684 | 0.0489 |
| 10 | 0.7919 | 0.7348 | 0.0728 | 0.0535 |
| 11 | 0.7799 | 0.7077 | 0.1640 | 0.1398 |
| 12 | 0.7649 | 0.6876 | 0.1037 | 0.0780 |
| 13 | 0.7546 | 0.6773 | 0.1319 | 0.1040 |
| 14 | 0.7489 | 0.6522 | 0.1292 | 0.0867 |
| 15 | 0.7054 | 0.5820 | 0.0883 | 0.0524 |
| 16 | 0.7597 | 0.6825 | 0.2356 | 0.2093 |
| 17 | 0.7233 | 0.6396 | 0.1838 | 0.1566 |
| 18 | 0.7089 | 0.6073 | 0.1526 | 0.1191 |
| 19 | 0.7277 | 0.6423 | 0.1501 | 0.1205 |
| 20 | 0.7144 | 0.6109 | 0.2139 | 0.1849 |
| 21 | 0.6975 | 0.5747 | 0.1912 | 0.1547 |
| 22 | 0.7187 | 0.6512 | 0.1755 | 0.1511 |
| 23 | 0.7173 | 0.6173 | 0.2763 | 0.2412 |
| 24 | 0.6940 | 0.6028 | 0.1864 | 0.1591 |
| 25 | 0.7031 | 0.5965 | 0.2033 | 0.1687 |
| 26 | 0.7474 | 0.6646 | 0.3336 | 0.3095 |
| 27 | 0.6965 | 0.5883 | 0.2621 | 0.2335 |
| 28 | 0.7032 | 0.5937 | 0.2881 | 0.2590 |
| 29 | 0.7645 | 0.6807 | 0.4147 | 0.3774 |
| 30 | 0.7530 | 0.6779 | 0.4018 | 0.3709 |
| 31 | 0.7365 | 0.6457 | 0.4062 | 0.3743 |
| 32 | 0.7303 | 0.6362 | 0.3860 | 0.3547 |
| 33 | 0.7323 | 0.6409 | 0.4085 | 0.3792 |
| 34 | 0.7718 | 0.6884 | 0.4923 | 0.4606 |
| 35 | 0.7642 | 0.6838 | 0.4674 | 0.4420 |
| 36 | 0.7140 | 0.6245 | 0.3888 | 0.3588 |
| 37 | 0.7394 | 0.6439 | 0.4385 | 0.4069 |
| 38 | 0.7434 | 0.6544 | 0.4385 | 0.4131 |
| 39 | 0.8014 | 0.7371 | 0.5728 | 0.5520 |
| 40 | 0.7176 | 0.6272 | 0.4337 | 0.4021 |

| Pipe | RIPS Runtime |
|---|---|
| 1 | 50 |
| 2 | 0 |
| 3 | 10 |
| 4 | 20 |
| 5 | 10 |
| 6 | 10 |
| 7 | 20 |
| 8 | 20 |
| 9 | 10 |
| 10 | 10 |
| 11 | 20 |
| 12 | 20 |
| 13 | 20 |
| 14 | 20 |
| 15 | 20 |
| 16 | 20 |
| 17 | 30 |
| 18 | 20 |
| 19 | 20 |
| 20 | 20 |
| 21 | 30 |
| 22 | 30 |
| 23 | 30 |
| 24 | 21 |
| 25 | 30 |
| 26 | 30 |
| 27 | 30 |
| 28 | 30 |
| 29 | 20 |
| 30 | 50 |
| 31 | 40 |
| 32 | 30 |
| 33 | 70 |
| 34 | 30 |
| 35 | 140 |
| 36 | 40 |
| 37 | 30 |
| 38 | 30 |
| 39 | 40 |
| 40 | 40 |

# Bibliography

[1] Emile Aarts and Jan Lenstra. *Local Search in Combinatorial Optimization*. Wiley, 1997.

[2] Shoukat Ali and et al. Greedy heuristics for resource allocation in dynamic distributed real-time heterogeneous computing systems. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, pages 519–530, June 2002.

[3] 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.

[4] Cedric Avanthay, Alain Hertz, and Nicolas Zuffrey. A variable neighborhood search for graph coloring. *European Journal of Operational Research*, 151:379–388, 2003.

[5] Prithviraj Banerjee and Nagaraj Shenoy. A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In *International Symposium on FPGA Custom Computing Machines*, 2000.

[6] Peter Bellows and Brad Hutchings. JHDL - an HDL for reconfigurable systems. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[7] Kiran Bondalapati and Viktor Prasanna. Loop pipelining and optimization for run time reconfiguration. In *Reconfigurable Architectures Workshop*, pages 906–915, May 2000.

[8] Vincenzo Catania, Michele Malgeri, and Marco Russo. Applying fuzzy logic to codesign partitioning. *IEEE Micro*, 17(3):62–70, 1997.

[9] J. Chang and M. Pedram. Codex-dp: Co-design of communicating systems using dynamic programming. Technical Report CENG 98-04, University of Southern California, 1998.

[10] Jui-Ming Chang and Massoud Pedram. Codex-dp: Co-design of communicating systems using dynamic programming. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(7), July 2000.

[11] Marco Chiarandini and Thomas Stutzle. An application of iterated local search to graph coloring. In *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, pages 112–125, 2002.

[12] Pai Chou, Ross Ortega, and Gaetano Boriello. Synthesis of the hardware/software interface in microcontroller-based systems. In *Proceedings of the International Conference on Computer Aided Design*, 1992.

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

[14] Stella da Silva Porto. Heuristic task scheduling algorithms in multiprocessors with heterogeneous architectures: A systematic construction and performance evaluation. Master's thesis, Departamento de Informatica da PUC-RIO, Rio de Janeiro, 1991.

[15] Stella da Silva Porto and Celso Ribeiro. A tabu search approach to task scheduling on heterogeneous processors under precedence constraints. *International Journal of High Speed Computing (IJHSC)*, 7(2):45–71, 1995.

[16] Stella da Silva Porto and Celso Ribeiro. A case study on parallel synchronous implementations of tabu search based on neighborhood decomposition. *Investigacion Operativa*, 5:233–259, 1996.

[17] John Davis and et al. Overview of the Ptolemy project. Technical report, University of California Berkeley, 2001.

[18] Raphael Dorne and Jin-Kao Hao. *Metaheuristics 98: Theory and Applications*, chapter Tabu Search for Graph Coloring, T-coloring and Set T-coloring. Kluwer Academic Publishers, 1998.

[19] Raphael Dorne and Jin-Kao Hao. A new genetic local search algorithm for graph coloring. *Lecture Notes in Computer Science*, pages 745–754, 1998.

[20] Eclipse. Available on the web at http://www.eclipse.org/. Last visited in July 2004.

[21] Eclipsecolorer profiling tool. Available on the web at http://sourceforge.net/projects/eclipsecolorer/. Last visited in July 2004.

[22] Andrzej Ehrenfeucht, Tero Harju, and Grzegorz Rozenberg. *The Theory of 2-Structures: A Framework for Decomposition and Transformation of Graphs*. World Scientific, 1999.

[23] Hersham El-Rewini, Theodore Lewis, and Hesham Ali. *Task Scheduling in Parallel and Distributed Systems*. PTR Prentice Hall, 1994.

[24] Petru Eles, Zebo Peng, Krzysztof Kuchcinski, and Alexa Doboli. Hardware/software partitioning with iterative improvement heuristics. In *International Symposium on System Synthesis (ISSS)*, pages 71–76, 1996.

[25] Dimitris Fotakis, Spyros Likothanassis, and Stamatis Stefanakos. An evolutionary annealing approach to graph coloring. In *Proceedings of Applications of Evolutionary Computing*, pages 120–129. Springer-Verlag, 2001.

[26] Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the streams-C C-to-FPGA compiler: an applications perspective. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field Programmable Gate Arrays*, pages 134–140. ACM Press, 2001.

[27] Daniel Gajski, Frank Vahid, and Sanjiv Narayan. SpecCharts: VHDL front-end for embedded systems. *IEEE Transactions on CAD*, 14(6):694–706, 1995.

[28] Daniel Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. SpecSyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design. *IEEE Transactions on Very Large Scale Integration Systems*, 6(1):84–100, 1998.

[29] Demetris Galatopoullos and Elias Manolakos. Developing parallel applications using the JavaPorts environment. In *Parallel and Distributed Processing*, volume 1586 of *Lecture Notes in Computer Science*, pages 813–828. Elsevier, 1999.

[30] Phillippe Galinier and Jin-Kao Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4):379–397, 1999.

[31] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, CA, 1979.

[32] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, 1993.

[33] Sethavidh Gertphol and et al. A metric and mixed-integer-programming-based approach for resource allocation in dynamic real-time systems. In *Proceedings of International the International Parallel and Distributed Processing Symposium*, April 2002.

[34] Maya Gokhale, Janice Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented FPGA computing in the streams-c high level language. In *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 49–58, 2000.

[35] Rajesh Gupta. *Co-synthesis of Hardware and Software for Digital Embedded Systems*. PhD thesis, Stanford University, 1993.

[36] Rajesh Gupta, Claudionor Coelho Jr., and Giovanni de Micheli. Synthesis and simulation of digital systems containing interacting hardware and software components. In *Design Automation Conference*, pages 225–230, 1992.

[37] Jean-Philippe Hamiez and Jin-Kao Hao. Scatter search for graph coloring. In *Artificial Evolution*, pages 168–179, 2001.

[38] Jeffrey Hammes, Bruce Draper, and Willem Bohm. Sassy: A language and optimizing compiler for image processing on reconfigurable computing systems. In *International Conference on Vision Systems (ICVS)*, pages 83–97, 1999.

[39] Jeffrey Hammes, Bob Rinker, Willem Bohm, Walid Najjar, Bruce Draper, and Ross Beveridge. Cameron: High level language compilation for reconfigurable systems. In *Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 236–244, 1999.

[40] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, March 1987.

[41] Jorg Henkel, Thomas Benner, and Rolf Ernst. Hardware generation and partitioning effects in the COSYMA system. In *Proceedings of the International Workshop on Hardware-Software Codesign*, 1993.

[42] Jorg Henkel and Rolf Ernst. A hardware/software partitioner using a dynamically determined granularity. In *Design Automation Conference*, pages 691–696, 1997.

[43] Jorg Henkel, Rolf Ernst, Ulrich Holtmann, and Thomas Benner. Adaptation of partitioning and high-level synthesis in hardware/software co-synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 96–100, 1994.

[44] Dirk Herrmann, Jorg Henkel, and Rolf Ernst. An approach to the adaptation of estimated cost parameters in the COSYMA system. In *Proceedings Third International Workshop on Hardware/Software Co-Design (Codes/CASHE '94)*, 1994.

[45] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice/Hall International, 1985.

[46] Xiabo Sharon Hu, Garrison Greenwood, and Joseph G. D'Ambrosio. An evolutionary approach to hardware/software partitioning. In Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature—PPSN IV*, pages 900–909. Springer-Verlag. Lecture Notes in Computer Science No. 1141, 1996.

[47] Brad Hutchings and Brent Nelson. Using general-purpose programming languages for FPGA design. In *Design Automation Conference (DAC)*, pages 561–566, 2000.

[48] A. Jantsch, J. Oberg, P. Ellervee, A. Hemani, and H. Tenhunen. A software oriented approach to hardware-software co-design. In *Proceedings of International Conf. on Compiler Construction*, pages 93–102, 1994.

[49] Axel Jantsch, Peter Ellervee, Ahmed Hemani, Johnny Oberg, and Hannu Tenhunen. Hardware/software partitioning and minimizing memory interface traffic. In *Proceedings of the Conference on European Design Automation Conference*, pages 226–231. IEEE Computer Society Press, 1994.

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

[51] Asawaree Kalavade and Edward Lee. The extended partitioning problem: Hardware/software mapping and implementation-bin selection. In *Proceedings of the Sixth International Workshop on Rapid Systems Prototyping*, pages 12–18, 1995.

[52] A. A. Khan, C. McCreary, and M. S. Jones. A comparison of multiprocessor scheduling heuristics. In *International Conference on Parallel Processing*, pages 243–250, 1994.

[53] P. Knudsen and J. Madsen. Pace: A dynamic programming algorithm for hardware-software partitioning. In *Proceedings of Int'l Workshop on Hardware/Software Codesign*, pages 85–92, March 1996.

[54] Peter Knudsen and Jan Madsen. Aspects of system modelling in hardware/software partitioning. In *Seventh IEEE International Workshop on Rapid Systems Prototyping*, pages 18–23, 1996.

[55] Peter Knudsen and Jan Madsen. Graph based communication analysis for hardware/software codesign. In *7th International Workshop on Hardware/Software Codesign, Codes'99*, pages 131–135, 1999.

[56] Benjamin Levine and Herman Schmit. Efficient application representation for HASTE: Hybrid architectures with a single executable. In *Field Programmable Custom Computing Machines (FCCM) conference*, 2003.

[57] Jinfeng Liu, Pai H. Chou, and Nader Bagherzadeh. Combined functional partitioning and communication speed selection for networked voltage-scalable processors. In *Proceedings of the International Symposium on System Synthesis*, pages 14–19, October 2002.

[58] Douglas A. Lyon. *Image Processing in Java*. Prentice Hall PTR, 1999.

[59] Roman Lysecky and Frank Vahid. A configurable logic architecture for dynamic hardware/software partitioning. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition Volume I (DATE'04)*, page 10480. IEEE Computer Society, 2004.

[60] Jan Madsen and et al. LYCOS: The Lyngby cosynthesis system. *Design Automat. Embedded Syst.*, 2(2):195–235, 1997.

[61] Jan Madsen and Bjarne Hald. An approach to interface synthesis. In *the 8th International Symposium of System Synthesis (ISSS)*, pages 16–21, 1995.

[62] Elias Manolakos and Demetris Galatopoullos. JavaPorts: An environment to facilitate parallel computing on a heterogeneous cluster of workstations. *Informatica (Slovenia)*, 23(1), 1999.

[63] C. McCreary. Partitioning for parallelization using graph parsing. Technical Report CSE91-17, Auburn University, 1991.

[64] C. McCreary, J. Thompson, H. Gill, T. Smith, and Y. Zhu. Partitioning and scheduling using graph decomposition. Technical Report CSE93-06, Auburn University, 1993.

[65] Daniel Menasce and Stella da Silva Porto. Processor assignment in heterogeneous parallel architectures. In *International Parallel Processing Symposium*, pages 186–191, March 1992.

[66] Daniel Menasce, Debanjan Saha, Stella da Silva Porto, Virgilio Almeida, and Satish Tripathi. Static and dynamic processor scheduling disciplines in heterogeneous parallel architectures. *Journal of Parallel and Distributed Computing*, 28(1):1–18, 1995.

[67] Ralf Niemann and Peter Marwedel. Hardware/software partitioning using Integer Programming. In *Proccedings of the European Design and Test*

*Conference*, pages 473–480, Paris, France, 1996. IEEE Computer Society Press (Los Alamitos, California).

[68] Ralf Niemann and Peter Marwedel. An algorithm for hardware/software partitioning using mixed Integer Linear Programming. *Design Automation for Embedded Systems*, 2(2):165–193, 1997.

[69] Sze-Wei 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.

[70] M. O'Nils, A. Jantsch, A. Hemani, and H. Tenhunen. Interactive hardware-software partitioning and memory allocation based on data transfer profiling. In *Proceeding of International Conference on Recent Advances in Mechatronics*, pages 447–452, 1995.

[71] Christos Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, Inc., 1998.

[72] Luis Paquete and Thomas Stützle. An experimental investigation of iterated local search for coloring graphs. In Stefano Cagnoni, Jens Gottlieb, Emma Hart, Martin Middendorf, and Gunther Raidl, editors, *Applications of Evolutionary Computing, Proceedings of EvoWorkshops2002: EvoCOP, EvoIASP, EvoSTim*, volume 2279, pages 121–130, Kinsale, Ireland, 3-4 2002. Springer-Verlag.

[73] Roberto Passerone, James Rowson, and Alberto Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Design Automation Conference*, pages 8–13, 1998.

[74] Aviral Shrivastava, Mohit Kumar, Sanjiv Kapoor, Shashi Kumar, and M. Balakrishnan. Optimal hardware/software partitioning for concurrent specification using dynamic programming. In *Proceedings of 13th International Conference on VLSI Design*, pages 110–113, 2000.

[75] Laurie Smith King, Heather Quinn, Miriam Leeser, Demetris Galatopoullos, and Elias Manolakos. Runtime execution of reconfigurable hardware in a Java environment. In *Proccedings of International Conference on Computer Design*, pages 380–385, Austin, TX, 2001.

[76] G. Stitt, R. Lysecky, and F. Vahid. Dynamic hardware/software partitioning: A first approach. In *Design Automation Conference*, pages 250–255, 2003.

[77] Greg Stitt and Frank Vahid. Hardware/software partitioning of software binaries. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 164–170. ACM Press, 2002.

[78] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[79] John R. Taylor. *An Introduction to Error Analysis: The Studies of Uncertainties in Physical Measurements*. University Science Books, 1982.

[80] Frank Vahid and Daniel Gajski. Clustering for improved system-level functional partitioning. In *8th International Symposium on System Synthesis*, pages 28–33, September 1995.

[81] Frank Vahid and Thuy Le. Towards a model for hardware and software functional partitioning. In *International Workshop on Hardware-Software Co-Design*, pages 116–123, 1996.

[82] Theerayod Wiangtong, Peter Y. K. Cheung, and Wayne Luk. Tabu search with intensification strategy for functional partitioning in hardware-software codesign. In *Field-Programmable Custom Computing Machines (FCCM)*, pages 297–298, 2002.

[83] M.-Y. Wu, W. Shu, and J. Gu. Efficient local search for DAG scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):617–627, 2001.

[84] Tao Yang and Apostolos Gerasoulis. PYRROS: Static scheduling and code generation for message passing multiprocessors. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 428–437, Washington, DC, 1992.

[85] Tao Yang and Apostolos Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.