

**ALGORITHMS AND SOFTWARE TOOLS FOR
MAPPING ARBITRARILY CONNECTED TASKS ONTO
AN ASYNCHRONOUS ARRAY OF SIMPLE
PROCESSORS**

By

ERIC WESLEY WORK
B.S.E.E. (University of Washington) March, 2004

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTERS OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Chair, Dr. Bevan M. Baas

Member, Dr. Soheil Ghiasi

Member, Dr. John D. Owens

Committee in charge
2007

© Copyright by ERIC WESLEY WORK 2007
All Rights Reserved

Abstract

Due to advances in VLSI technology large scale parallel arrays are being developed at a rapid rate and growing larger in size with each new chip fabricated. Programming these large scale parallel arrays using fully manual techniques is very difficult as the array size grows larger. In this work we present a framework for mapping arbitrarily connected task graphs onto nearest neighbor dominated 2D-mesh parallel arrays. The contributions are an automated mapping algorithm, for placing and routing applications onto parallel arrays, and an intuitive graphical user interface, for creating applications based on their dataflow. In this work we demonstrate how an automated mapping algorithm is essential for efficiently programming large scale parallel arrays. The *Asynchronous Array of Simple Processors* (AsAP) architecture is used as a test platform for the mapping algorithm, but the mapping algorithm can easily be adapted to other parallel array architectures. The mapping algorithm is time efficient, scalable up to thousands of processors, tolerant of fabrication errors, and can even optimize applications using processor characteristics. Other features include, configurable architectural parameters, customizable user cost functions, and dynamically inserted routing processors which handle intersecting datastreams.

Acknowledgments

This research never would have been possible without the support of many other knowledgeable, talented, and loving people. Very few things in life are done in solitude.

First and foremost I'd like to thank my adviser Professor Bevan M. Baas for his invaluable wisdom and constant encouragement. You have advised me on a variety of academic and personal projects that have taught me practical industry skills that greatly compliment my education. I'm also very grateful for the funding you've been able to offer me for the past two years.

I'd of course like to thank my committee members, Professor Soheil Ghiasi Professor and John D. Owens. Professor Ghiasi, without you I wouldn't have had the solid foundation I needed for my mapping algorithm. Professor Owens, the mapping algorithm never would have been able to handle the large and complex applications that it handles now if it wasn't for your high expectations and positive criticism.

I'd like to thank everyone in the VLSI Computation Laboratory for their support during those rough times both inside and outside the lab. Toney and Wayne, thanks for the great games of Starcraft when we should have been working. Zhiyi, Tinoosh, and Dean, thanks for teaching me about AsAP and helping me put all those CAD tools to good use. Jeremy, thanks for the tips on how to improve the graphical user interface and teaching me about FPGAs.

I'm eternally grateful for the endless support from my wife Lauren. There were many times when my research led to immense frustration and thoughts of giving up. You helped me remain focused and optimistic when I thought I had lost hope.

I'd like to thank my parents for the past 24 years. You have always been there for me and done everything in your power to keep my spirits up and help me pursue my dreams. Mom, I can't thank you enough for allowing me to work on my research full-time until when I was supposed to be working with you.

Finally I'd like to thank Jake, Karl, Nadine, and all of my other close friends for the wonderful memories that I have. We must remember to never lose touch.

This work was supported in part by Intel Corporation, UC MICRO, the National Science Foundation under Grant No. 0430090 and CAREER Award 0546907, SRC, Intelliasys Corporation, ST Microelectronics, SEM, MOSIS, Artisan, and a University of California, Davis, Faculty Research Grant.

It now ends the same way it began, with a single key stroke!

Contents

Abstract	iii
Acknowledgments	iv
List of Figures	viii
List of Tables	xii
1 Introduction	1
1.1 Project Goals	2
1.2 Organization	2
2 Target Platform	5
2.1 AsAP Architecture	5
2.1.1 Input/Output Limitations	6
2.1.2 Programming	7
2.2 AsAP Version 2.0	8
2.3 Conclusion	9
3 Mapping Algorithm	11
3.1 Overview	11
3.1.1 Inputs and Outputs	12
3.2 Placement Phase	15
3.2.1 Simulated Annealing	16
3.2.2 Algorithm Details	17
3.2.3 Modifications	39
3.2.4 Summary	40
3.3 Routing Phase	40
3.3.1 Maze Routing	41
3.3.2 Algorithm Details	41
3.3.3 Modifications	59
3.3.4 Summary	59
3.4 Top-Level	60
3.5 Conclusion	62
4 Implementation	63
4.1 Back-end	63
4.2 Front-end	64
4.2.1 Procedure	65
4.2.2 Interface	67
4.3 XML File Formats	75
4.3.1 Module Files	75

4.3.2	Project Files	77
4.4	Conclusion	78
5	Evaluation Methods	79
5.1	Quality Metrics	79
5.1.1	Communication	80
5.1.2	Area	80
5.1.3	Utilization	82
5.1.4	Runtime	82
5.1.5	Summary	84
5.2	Applications	84
5.2.1	Building Blocks	84
5.2.2	802.11a Wireless Transmitter	85
5.2.3	Viterbi Decoder	86
5.2.4	Fast Fourier Transform	87
5.2.5	Clos Networks	87
5.2.6	Random Graphs	88
5.2.7	Multi-App Application	89
5.3	Conclusion	92
6	Results	93
6.1	Procedure	93
6.2	Efficiency	94
6.2.1	802.11a Wireless Transmitter	95
6.2.2	Viterbi Decoder	96
6.2.3	Fast Fourier Transform	97
6.2.4	Small Clos Network	101
6.2.5	Large Clos Network	102
6.2.6	Summary	107
6.3	Scalability	108
6.3.1	100 Random Nodes	109
6.3.2	250 Random Nodes	109
6.3.3	500 Random Nodes	110
6.3.4	1000 Random Nodes	111
6.3.5	Summary	114
6.4	Fault Tolerance	116
6.4.1	Multi-App Application	117
6.4.2	100 Random Nodes	134
6.4.3	802.11a Wireless Transmitter	150
6.4.4	Multiple Exclusions	154
6.4.5	Summary	157
6.5	Fabrication Differences	158
6.5.1	Value Annotations	159
6.5.2	Custom User Function	165
6.5.3	Numerical Evaluation	166
6.5.4	802.11a Wireless Transmitter	166
6.5.5	Viterbi Decoder	171
6.5.6	Summary	176
6.6	Conclusion	179

7	Related Work	181
7.1	Parallel Processor Arrays	181
7.1.1	RAW	181
7.1.2	Smart Memories	182
7.1.3	iWarp	183
7.1.4	Imagine Stream Processor	184
7.1.5	Explicit Data Graph Execution (TRIPS)	185
7.2	Parallel Programming Tools	185
7.2.1	StreamIt	186
7.2.2	OREGAMI	187
7.2.3	CASCH	188
7.2.4	PYRROS	189
7.2.5	HyperTool	189
7.2.6	Energy-Aware Mapping Algorithm	190
8	Conclusion	193
8.1	Lessons Learned	194
8.2	Future Work	195
A	Readme - Mapping Library	197
B	Readme - AsAP Mapping Tool	209
	Glossary	217
	Bibliography	223

List of Figures

2.1	Block level overview for the first version of AsAP	6
2.2	Block level overview for the second version of AsAP	9
3.1	Flowchart for the mapping algorithm	13
3.2	Visual representation of the various properties for the <i>Graph</i> , <i>Vertex</i> , and <i>Edge</i> variables	15
3.3	Depiction of how path intersections are determined using rectangular bounding boxes	26
3.4	Depiction of the five possible targets for the primary input and the primary output .	27
3.5	Improvements in minimum configuration cost over three iterations of the temperature schedule	35
3.6	The basics of the maze routing algorithm shown visually	43
3.7	Depiction of the two possible column dependencies and the two possible row dependencies	46
4.1	The main window of the AsAP mapping tool	69
4.2	The array window of the AsAP mapping tool	70
4.3	The code window of the AsAP mapping tool	71
4.4	The array settings tab of the mapping dialog for the AsAP mapping tool	72
4.5	The algorithm settings tab of the mapping dialog for the AsAP mapping tool	73
4.6	The other settings tab of the mapping dialog for the AsAP mapping tool	74
4.7	The Document Object Model hierarchy for XML module files	76
4.8	The Document Object Model hierarchy for XML project files	77
5.1	Visual depiction of the difference between rectangular array area and enclosed array area	81
5.2	The basic building blocks used for creating applications	85
5.3	The dataflow graph entered into the mapping tool for the 802.11a wireless transmitter	85
5.4	The dataflow graph entered into the mapping tool for the Viterbi decoder	86
5.5	The dataflow graph entered into the mapping tool for the Fast Fourier Transform . .	87
5.6	The dataflow graph entered into the mapping tool for the small Clos network	88
5.7	The dataflow graph entered into the mapping tool for the large Clos network	88
5.8	Basic constructs used to build the random node applications	89
5.9	Examples of randomly generated graphs that are used for the random node applications	90
5.10	Applications used for constructing the multi-app application	90
5.11	The dataflow graph entered into the mapping tool for the multi-app application . . .	91
6.1	Side-by-side comparison of the hand mapping and the automatic mapping for the 802.11a wireless transmitter	96
6.2	Reduction in optimization cost over time for the 802.11a wireless transmitter using 1000 trials	97
6.3	Side-by-side comparison of the hand mapping and the automatic mapping for the Viterbi decoder	98

6.4	Reduction in optimization cost over time for the Viterbi decoder using 1000 trials . .	98
6.5	Side-by-side comparison of the hand mapping and the automatic mapping for the Fast Fourier Transform	99
6.6	Reduction in optimization cost over time for the Fast Fourier Transform using 1000 trials	100
6.7	Automatic mapping for the Fast Fourier Transform when targeting the second version of AsAP	100
6.8	Side-by-side comparison of the hand mapping and the automatic mapping for the small Clos network	101
6.9	Reduction in optimization cost over time for the small Clos network using 1000 trials	102
6.10	Automatic mapping for the small Clos network when targeting the second version of AsAP	103
6.11	Side-by-side comparison of the hand mapping and the automatic mapping for the large Clos network	105
6.12	Reduction in optimization cost over time for the large Clos network using 1000 trials	106
6.13	Automatic mapping for the large Clos network when targeting the second version of AsAP	106
6.14	Reduction in optimization cost over time for the 100 random node application using 100 trials	110
6.15	Reduction in optimization cost over time for the 250 random node application using 100 trials	111
6.16	Reduction in optimization cost over time for the 500 random node application using 100 trials	112
6.17	Reduction in optimization cost over time for the 1000 random node application using 100 trials	113
6.18	Plot of the application runtime with respect to problem size for the random nodes applications	115
6.19	Plot of the increase in metric quality, relative to the minimum number of nodes, with respect to problem size for the random nodes applications	115
6.20	2D-plot of the minimum rectangular array area when excluding each processor individually for the multi-app application and targeting the first version of AsAP	119
6.21	Histogram of the minimum rectangular array area when excluding each processor individually for the multi-app application and targeting the first version of AsAP . .	120
6.22	Cumulative Distribution Function for the minimum rectangular array area when excluding each processor individually for the multi-app application and targeting the first version of AsAP	121
6.23	2D-plot of the minimum number of routing processors when excluding each processor individually for the multi-app application and targeting the first version of AsAP . .	122
6.24	Histogram of the minimum number of routing processors when excluding each processor individually for the multi-app application and targeting the first version of AsAP	123
6.25	Cumulative Distribution Function for the minimum number of routing processors when excluding each processor individually for the multi-app application and targeting the first version of AsAP	124
6.26	Best automatic mapping for the multi-app application after excluding each processor individually while targeting the first version of AsAP	125
6.27	2D-plot of the minimum rectangular array area when excluding each processor individually for the multi-app application and targeting the second version of AsAP . .	127
6.28	Histogram of the minimum rectangular array area when excluding each processor individually for the multi-app application and targeting the second version of AsAP	128
6.29	Cumulative Distribution Function for the minimum rectangular array area when excluding each processor individually for the multi-app application and targeting the second version of AsAP	129

6.30	2D-plot of the minimum number of long-distance interconnects when excluding each processor individually for the multi-app application and targeting the second version of AsAP	130
6.31	Histogram of the minimum number of long-distance interconnects when excluding each processor individually for the multi-app application and targeting the second version of AsAP	131
6.32	Cumulative Distribution Function for the minimum number of long-distance interconnects when excluding each processor individually for the multi-app application and targeting the second version of AsAP	132
6.33	Best automatic mapping for the multi-app application after excluding each processor individually while targeting the second version of AsAP	133
6.34	2D-plot of the minimum rectangular array area when excluding each processor individually for the 100 random nodes application and targeting the first version of AsAP	135
6.35	Histogram of the minimum rectangular array area when excluding each processor individually for the 100 random nodes application and targeting the first version of AsAP	136
6.36	Cumulative Distribution Function for the minimum rectangular array area when excluding each processor individually for the 100 random nodes application and targeting the first version of AsAP	137
6.37	2D-plot of the minimum number of routing processors when excluding each processor individually for the 100 random nodes application and targeting the first version of AsAP	138
6.38	Histogram of the minimum number of routing processors when excluding each processor individually for the 100 random nodes application and targeting the first version of AsAP	139
6.39	Cumulative Distribution Function for the minimum number of routing processors when excluding each processor individually for the 100 random nodes application and targeting the first version of AsAP	140
6.40	Best automatic mapping for the 100 random nodes application after excluding each processor individually while targeting the first version of AsAP	141
6.41	2D-plot of the minimum rectangular array area when excluding each processor individually for the 100 random nodes application and targeting the second version of AsAP	143
6.42	Histogram of the minimum rectangular array area when excluding each processor individually for the 100 random nodes application and targeting the second version of AsAP	144
6.43	Cumulative Distribution Function for the minimum rectangular array area when excluding each processor individually for the 100 random nodes application and targeting the second version of AsAP	145
6.44	2D-plot of the minimum number of long-distance interconnects when excluding each processor individually for the 100 random nodes application and targeting the second version of AsAP	146
6.45	Histogram of the minimum number of long-distance interconnects when excluding each processor individually for the 100 random nodes application and targeting the second version of AsAP	147
6.46	Cumulative Distribution Function for the minimum number of long-distance interconnects when excluding each processor individually for the 100 random nodes application and targeting the second version of AsAP	148
6.47	Best automatic mapping for the 100 random nodes application after excluding each processor individually while targeting the second version of AsAP	149
6.48	2D-plot of the minimum rectangular array area when excluding each processor individually for the 802.11a wireless transmitter application and targeting the first version of AsAP	151

6.49	Histogram of the minimum rectangular array area when excluding each processor individually for the 802.11a wireless transmitter application and targeting the first version of AsAP	152
6.50	Cumulative Distribution Function for the minimum rectangular array area when excluding each processor individually for the 802.11a wireless transmitter application and targeting the first version of AsAP	153
6.51	Best automatic mapping for the 802.11a wireless transmitter application after excluding each processor individually while targeting the first version of AsAP	154
6.52	Plot of the minimum rectangular array area with sets of 10 excluded processors for the 802.11a wireless transmitter application while targeting the first version of AsAP	155
6.53	Plot of the minimum rectangular array area with sets of 20 excluded processors for the 802.11a wireless transmitter application while targeting the first version of AsAP	156
6.54	Plot of the minimum rectangular array area with sets of 30 excluded processors for the 802.11a wireless transmitter application while targeting the first version of AsAP	156
6.55	Plot of the minimum rectangular array area with respect to the number of excluded processors for the 802.11a wireless transmitter	158
6.56	Maximum frequency value for each processor in the target 10x10 array	161
6.57	Leakage current value for each processor in the target 10x10 array	162
6.58	Load average and activity level for each task in the 802.11a wireless transmitter application	163
6.59	Load average and activity level for each task in the Viterbi decoder application . . .	164
6.60	Automatic mapping without using any annotations for the 802.11a wireless transmitter	167
6.61	Automatic mapping using just speed related annotations for the 802.11a wireless transmitter	168
6.62	Automatic mapping using just power related annotations for the 802.11a wireless transmitter	169
6.63	Automatic mapping using both speed and power related annotations for the 802.11a wireless transmitter	170
6.64	Automatic mapping without using any annotations for the Viterbi decoder	171
6.65	Automatic mapping using just speed related annotations for the Viterbi decoder . .	173
6.66	Automatic mapping using just power related annotations for the Viterbi decoder . .	174
6.67	Automatic mapping using both speed and power related annotations for the Viterbi decoder	175
6.68	Visual comparison between the three annotated mappings for the 802.11a wireless transmitter application	177
6.69	Visual comparison between the three annotated mappings for the Viterbi decoder application	178

List of Tables

3.1	Typefaces and notations used by the pseudo-code listings.	12
3.2	List of properties for the <i>Graph</i> , <i>Vertex</i> , and <i>Edge</i> variables.	14
3.3	List of properties for the <i>Config</i> variable.	14
3.4	List of properties for the <i>Perturb</i> variable.	33
3.5	List of properties for the <i>Gridmap</i> variable.	51
5.1	A breakdown of the runtime for the mapping algorithm while mapping the 802.11a wireless transmitter using the default settings.	83
6.1	Relevant default batch mode configuration parameters	94
6.2	The rectangular array area for the hand mappings and the automatic mappings, when targeting the first version of AsAP	107
6.3	The enclosed array area for the hand mappings and the automatic mappings, when targeting the first version of AsAP	108
6.4	The rectangular array area for the hand mappings and the automatic mappings, when targeting the second version of AsAP	108
6.5	The enclosed array area for the hand mappings and the automatic mappings, when targeting the second version of AsAP	108
6.6	Runtime and increase in metric quality, relative to the minimum number of nodes, with respect to problem size for the random node applications.	114
6.7	The estimated sample latency and the estimated leakage power for automatic mappings both with and without annotations for the 802.11a wireless transmitter application	176
6.8	The estimated sample latency and the estimated leakage power for automatic mappings both with and without annotations for the Viterbi decoder application	176

List of Algorithms

3.1	Placement Phase - PlacementMain	19
3.2	Placement Phase - InitPlacement	21
3.3	Placement Phase - NextCoord	22
3.4	Placement Phase - ConfigCost	24
3.5	Placement Phase - PathRoutable	25
3.6	Placement Phase - InputCost	28
3.7	Placement Phase - OutputCost	29
3.8	Placement Phase - ArrayCost	29
3.9	Placement Phase - InitialTemp	31
3.10	Placement Phase - FinalTemp	32
3.11	Placement Phase - PerturbGraph	35
3.12	Placement Phase - PerturbSetup	36
3.13	Placement Phase - RandVertex	37
3.14	Placement Phase - PerturbApply	38
3.15	Placement Phase - PerturbUndo	39
3.16	Routing Phase - RoutingMain	43
3.17	Routing Phase - InsertSpacing	45
3.18	Routing Phase - EdgeDepends	47
3.19	Routing Phase - ColSplits	48
3.20	Routing Phase - RowSplits	49
3.21	Routing Phase - ShiftArray	50
3.22	Routing Phase - InitGridmap	52
3.23	Routing Phase - Propagate	54
3.24	Routing Phase - QueueNeighbors	55

3.25 Routing Phase - Traceback	56
3.26 Routing Phase - NextNeighbor	57
3.27 Routing Phase - InsertRouter	58
3.28 Routing Phase - Cleanup	59
3.29 AlgorithmMain	62
6.1 UserCost	165

Chapter 1

Introduction

An automated mapping algorithm is essential to efficiently program large scale parallel arrays. An automated mapping algorithm can take into account over a dozen optimization factors at one time. Without an automated mapping algorithm programming large scale parallel arrays is nearly impossible when applications are divided across thousands of processing elements. This work shows the applications for a mapping algorithm using the *Asynchronous Array of Simple Processors* (AsAP) architecture as the test platform [37]. This thesis demonstrates quantitatively through mapped complex applications the benefits of using an automated mapping tool over hand mappings. Some of these benefits are time efficiency, scalability up to thousands of processors, tolerance to fabrication errors, and optimizations using processor characteristics to name a few.

Programming large scale parallel arrays is not quite as straight forward as programming general purpose processors. There are a number of additional steps involved. The first step is to partition an application into a number of parallel or cascaded tasks. For some applications this may be extremely difficult. The next step is to place tasks onto processors. The technique used when placing tasks onto processors depends upon how the application is being optimized and the constraints imposed by the target architecture. Communication channels must also be created to satisfy data dependencies when maximizing nearest neighbor communication. The final step is to schedule the code inside each processor to avoid deadlocks and optimize throughput. The mapping algorithm presented in this work only focuses on assigning tasks to processors and creating communication channels between tasks. Partitioning an application and scheduling code are beyond the scope of this work.

This work is intended to be more than just a mapping algorithm for 2D-mesh nearest

neighbor dominated parallel arrays. It's a framework for mapping applications to large scale parallel arrays in general. The framework consists of two parts, an automated mapping algorithm and an intuitive graphical user interface. The majority of this paper is focused on the mapping algorithm since it contains most of the intellectual contributions. The mapping algorithm primarily targets the AsAP architecture, but it can easily be modified to target other architectures. Even though the focus of this paper is on the mapping algorithm the graphical user interface contains some important contributions. The graphical user interface explores a new technique for programming parallel arrays by allowing applications to be created primarily using their dataflow. I welcome other researchers who wish to improve or extended this framework to other parallel arrays.

1.1 Project Goals

- Map an arbitrary dataflow graph to a 2D-array of processors
- Maximize the use of nearest neighbor connections
- Minimize the area and if possible the perimeter
- Dynamically insert routing processors
- Minimize the use of routing processors
- Allow processors to be excluded from the mapping
- Allow tasks to be assigned to fixed locations
- Handle very small and very large problem sizes
- Optimize mappings using the physical properties of a processor
- Allow a customizable user cost function and user datafile

1.2 Organization

The remainder of this paper is divided into 7 additional chapters. Chapter 2 talks about the AsAP architecture, which is the test platform for the mapping algorithm. Chapter 3 talks about the two phases of the mapping algorithm. Chapter 4 talks about the implementation of the mapping algorithm and the AsAP mapping tool. Chapter 5 talks about the metrics and applications used to evaluate the mapping algorithm. Chapter 6 talks about the results obtained from evaluating the

mapping algorithm using a variety of optimizations. Chapter 7 talks about work related to AsAP and the mapping algorithm. Finally Chapter 8 summarizes this work and discusses future directions for the framework.

Chapter 2

Target Platform

The target platform for this work is the *Asynchronous Array of Simple Processors* (AsAP) architecture. AsAP is a highly parallel chip-multiprocessor architecture designed for efficiently executing DSP applications. Each processing element is a low power, high clock rate, RISC-style processor core, which achieves high energy efficiency while executing computationally intensive DSP kernels. This chapter discusses the details of the AsAP architecture that are most relevant to the mapping problem, mainly the communication infrastructure. The instruction set and general architectural design are not discussed as they are not applicable to the mapping problem.

2.1 AsAP Architecture

The AsAP architecture is able to attain high performance while maintaining high energy efficiency. This is accomplished by using simple processors with reduced area and power requirements. Relative to typical DSP processors, a smaller proportion of AsAP's area is dedicated to memory with most of the area used for the processor core [37]. Each processor is clocked synchronously by an independent local oscillator. Communication between processors is performed asynchronously. By using a *Globally Asynchronous Locally Synchronous* (GALS) clocking style, AsAP is well suited for future technologies because of shorter wire lengths and a simplified clock tree. The first in-depth investigation on using GALS architectures was done by Chapiro [10]. By using independent clock domains for each processor, power can be reduced by turning off processors that are waiting for data or have no work to do [7].

Processors in the first version of AsAP are arranged in a 6x6 array, as seen in Figure 2.1, for a total of 36 processing elements. Each processing element contains a nine stage pipeline with

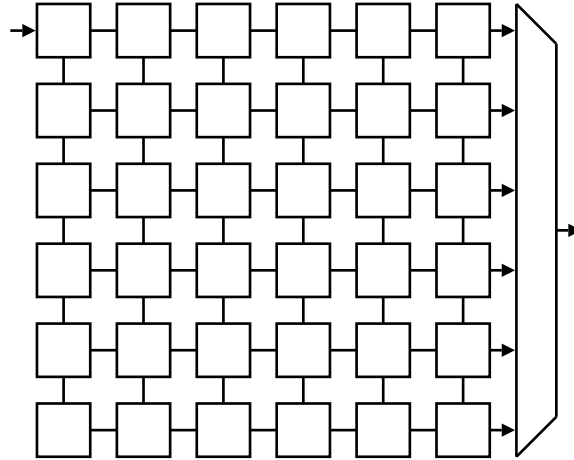


Figure 2.1: Block level overview for the first version of AsAP, which contains 36 processors arranged in a 6x6 array connected using a nearest neighbor 2D-mesh

a 16-bit fixed-point datapath and a 40-bit multiply accumulator. The instruction set consists of 54 32-bit instructions with only one instruction that is algorithm specific, bit-reverse. The array has one input, connected to the top-left processor, and one output, connected to any one of the right edge processors. Each processor has two inputs and four outputs connected in a nearest neighbor 2D-mesh. This version of AsAP allows only nearest neighbor communication, which can be restrictive for some applications, but highly efficient for other applications that map well to the 2D-mesh array [37].

2.1.1 Input/Output Limitations

Communication between processors is performed using dual-clock FIFOs due to AsAP's asynchronous nature. The dual-clock FIFO was designed by Ryan Apperson for his master's thesis titled *A Dual-Clock FIFO for the Reliable Transfer of High-Throughput Data Between Unrelated Clock Domains* [6]. The two processors connected to a dual-clock FIFO operate in entirely unrelated clock domains. Each dual-clock FIFO contains a 32-word SRAM with a configurable amount of reserve space to avoid overflow. During extensive testing no meta-stability problems occurred.

Read and write pointers are exchanged between clock domains to check whether or not the FIFO is full or empty. When sending data the sending processor first increments its write pointer then sends a handshake signal to the receiving processor to increment its write pointer. Similarly when receiving data the receiving processor first increments its read pointer then sends a handshake signal to the sending processor to increment its read pointer. The read and write pointers are gray coded before being sent across the clock boundary to increase reliability. Gray coding is used to

avoid race conditions caused by changing multiple signals simultaneously. The FIFO compares the read and write pointers in each respective clock domain to determine if it's full or empty. This information is sent back to the processor, stalling the local oscillator when a full or empty condition occurs. By generating these condition flags the FIFO automatically performs flow control based on the production and consumption rates of neighboring processors.

There are three possibilities when deciding where to place the FIFO logic. The first possibility is to place half the logic in the sender and half the logic in the receiver. The second possibility is to place all the logic in the sender and just run wires to the receiver. The third possibility is to place all the logic in the receiver and just run wires from the sender. For AsAP the FIFO logic is placed in the receiving side of each connection. This makes the output ports more flexible than the input ports.

The downside to putting the FIFO logic in the receiving processor is that it limits the number of inputs ports. Since area efficiency is a primary concern for AsAP, only two FIFOs were included in each processor. With only two FIFOs each processor can only receive data from two of its four nearest neighbors at a given time. The two FIFOs can operate independently since they are addressed using different instruction sources. This allows both FIFOs to be read with a single instruction. The input direction for each FIFO is set during the configuration phase before the application is executed. The array must be reconfigured or given explicit commands from the host computer in order to change the input ports.

The upside to putting the FIFO logic in the receiving processor is that the output ports are more flexible. The output ports can be changed dynamically during runtime. Each processor can talk to any combination of its four nearest neighbors simultaneously using an output mask. The output mask is 4-bits long with one bit representing each neighbor: north, south, east, and west. Data is broadcast to each processor whose bit is set in the output mask. Multiple destination routing schemes other than broadcasting, such as round-robin, must be done in software using combinations of output masks.

2.1.2 Programming

There are primarily two programming languages available for programming the AsAP array. The first is AsAP-ASM (or AsAP assembly) and the second is AsAP-C, which includes a subset of the C programming language. The current implementation of the AsAP-C compiler lacks the ability to automatically partition code. Therefore code must be manually partitioned into independent

parallel tasks. After code partitioning the application is represented by a set of AsAP-C or AsAP-ASM program kernels, one for each processor. Due to the lack of an automatic partitioner data dependencies between program kernels must also be specified manually by the user. This work helps with the final step of programming AsAP, which is placing tasks onto processors. Before this tool was created the entire process was done manually.

Once the user is satisfied with the simulation results the next step is to run the application on the physical chip to obtain measurements. To program the physical chip the complete application is packed into a binary bitstream consisting of address/data pairs. A utility I wrote called the AsAP Programmer (or `aprogram`) creates this binary bitstream directly from AsAP-ASM code. The AsAP Programmer, running on the host computer, loads the binary bitstream into the array's configuration and instruction memories through an intermediate FPGA. The AsAP Programmer can control each individual processor. This utility can either halt a single processor or the entire array then change the frequency, the input ports, the instruction memory, or other aspects of any processor. With this level of control the array can be remapped and reprogrammed dynamically.

2.2 AsAP Version 2.0

The second version of the AsAP array processor is still in development but its features are worth discussing since it pertains to this work. The most noticeable change is the array size. The second version of AsAP has an array of size 13x13 with a few of the lower processors replaced by hardware-based accelerators. For the sake of this work the array is assumed to be homogeneous with an array of size 16x16 (similar to the original design) for a total of 256 processing elements. There have also been a number of general architectural improvements such as, new min and max instructions, and conditional execution, but these changes do not affect this work. Another pertinent change is the addition of a long-distance point-to-point routing overlay network. With this new routing overlay network some applications can be mapped more efficiently.

Since details for the second version of AsAP have not been finalized I will discuss the implementation that is supported by this work. Figure 2.2 shows the communication infrastructure that is currently planned for the second version of AsAP. Each processor now has 8 output ports. The original 4 ports remain the same and an additional 4 ports have been added, one for each direction. To support the routing overlay network each processor contains a number of switches. These switches can be configured to either consume the data or send the data to another switch inside a neighboring processor. Consuming the data involves routing the data to one of the processor's

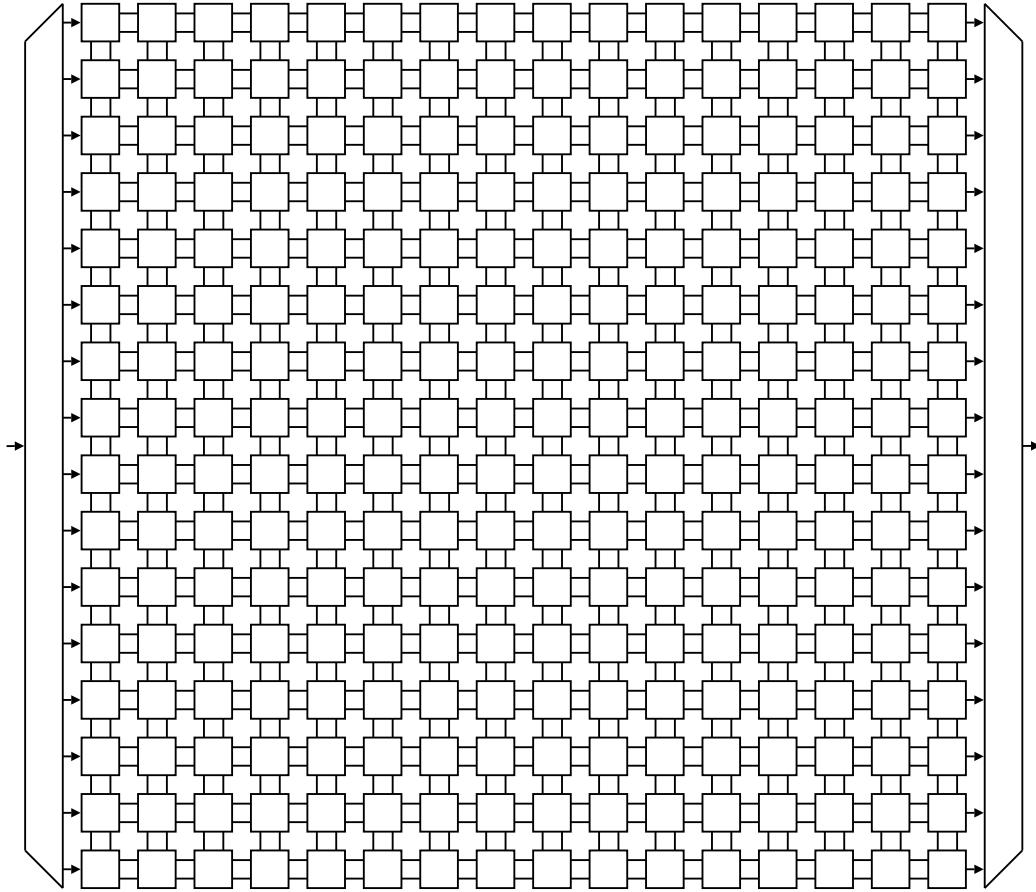


Figure 2.2: Block level overview for the second version of AsAP, which contains 256 processors arranged in a 16x16 array connected using both a nearest neighbor 2D-mesh and a long-distance routing overlay network

two local FIFOs. By configuring a series of switches along a path a long-distance interconnect is created. Details on how these switches are configured in hardware has not yet been determined so it's handed-off to a post-processing tool. Implementing this post-processing tool is left for future work. In addition to the routing overlay network, and the additional output ports, the array can now redirect the input data to any processor along the left edge.

2.3 Conclusion

There are many aspects of the AsAP architecture that present challenges to the mapping problem. The first version of AsAP uses nearest neighbor communication exclusively, which for some applications make inefficient use of the array. For applications that are mostly feed-forward and resemble segmented pipelines this is not much of a problem. For more complex applications routing processors are required. Routing processors are processors whose sole purpose is to for-

ward data between computation processors. Routing processors can often be avoided in the second version of AsAP using the routing overlay network, which leads to some interesting comparisons. Scalability becomes an issue as the number of processors in AsAP approaches several hundred and manually mapping tasks to processors becomes less feasible. Each processor is also somewhat unique due to physical differences, which are created during fabrication. By taking these differences into consideration mappings can be better suited for the target chip not just the target architecture.

Chapter 3

Mapping Algorithm

This chapter discusses the details of the mapping algorithm, which places tasks onto physical processors. The mapping algorithm operates in two phases. The first phase is the placement phase, which assigns tasks to processors while trying to minimize area and maximize nearest neighbor communication. The second phase is the routing phase, which inserts routing processors to complete non-nearest neighbor connections. The primary focus of the mapping algorithm is mapping applications onto the first version of AsAP. The algorithm contains a number of configurable parameters, which allow it to map applications onto the second version of AsAP (and other parallel array architectures), and also improve the mapping quality for difficult applications.

3.1 Overview

Since the mapping algorithm is rather long and complex the pseudo-code has been broken down into a number of listings that resemble functions in a normal programming language. As a result this chapter contains a large number of these pseudo-code listings each documenting a specific part of the mapping algorithm. The listings are grouped into three sections, placement (3.2), routing (3.3), and top-level (3.4). The first two sections discuss in detail the functions required by each phase, the goals, and how optimization parameters were chosen. The final section discusses how the two phases interact and any post-processing that occurs. The typefaces and notations used throughout the pseudo-code listings are shown in Table 3.1. These typefaces and notations help differentiate code elements for easier reading.

The two phases of the mapping algorithm, the placement phase and the routing phase, have been divided into a number of major components. Whether or not these components are enabled

Notation	Description
Keyword	Reserved word
<i>Variable</i>	Variable name
<i>Variable.Property</i>	Variable property
<i>CONSTANT</i>	Constant variable
<i>Variable</i> ← <i>Value</i>	Variable assignment
<i>Array[Index]</i>	Array indexing
Function	Function name
Function (<i>Argument</i>)	Function call
Coordinate (X, Y)	Coordinate object

Table 3.1: Typefaces and notations used by the pseudo-code listings.

depends upon the configuration parameters. Which components get executed and how often they are executed depends upon the application dataflow. The flowchart in Figure 3.1 shows how these components interact and a few of the higher-level decisions that must be made. The mapping algorithm begins its execution at **Start**, accepting two input data structures, and terminates its execution when **Finish** is reached, producing one output data structure. The various pseudo-code listings throughout this chapter have been grouped into categories that correspond to the major components in the flowchart.

3.1.1 Inputs and Outputs

The mapping algorithm requires two input data structures or arguments. The first data structure is the application’s dataflow graph and the second data structure contains the configuration parameters for the mapping algorithm. These two data structures are stored globally and are available for the lifetime of the algorithm. The application’s dataflow graph is accessed throughout the algorithm using the *Graph* variable, which holds information about task inter-dependencies using a series of edges and vertices. Each edge and each vertex contain a number of properties as well as the graph itself. These properties are listed in Table 3.2 as well as shown visually in Figure 3.2. The algorithm configuration parameters are accessed throughout the algorithm using the *Config* variable, which determines the optimizations that are performed and the general structure for the final mapping. The properties for this variable are listed in Table 3.3. The way each configuration parameter affects the mapping algorithm is deferred until it’s used within the pseudo-code listings. The common defaults used for each configuration parameter and each compile-time constant are based on mapping the applications detailed in Chapter 5, executing on Intel Xeon 2.0 GHz processors.

The output from the mapping algorithm is a new data structure similar to the input data structure which contained the application’s dataflow graph. The primary difference between the

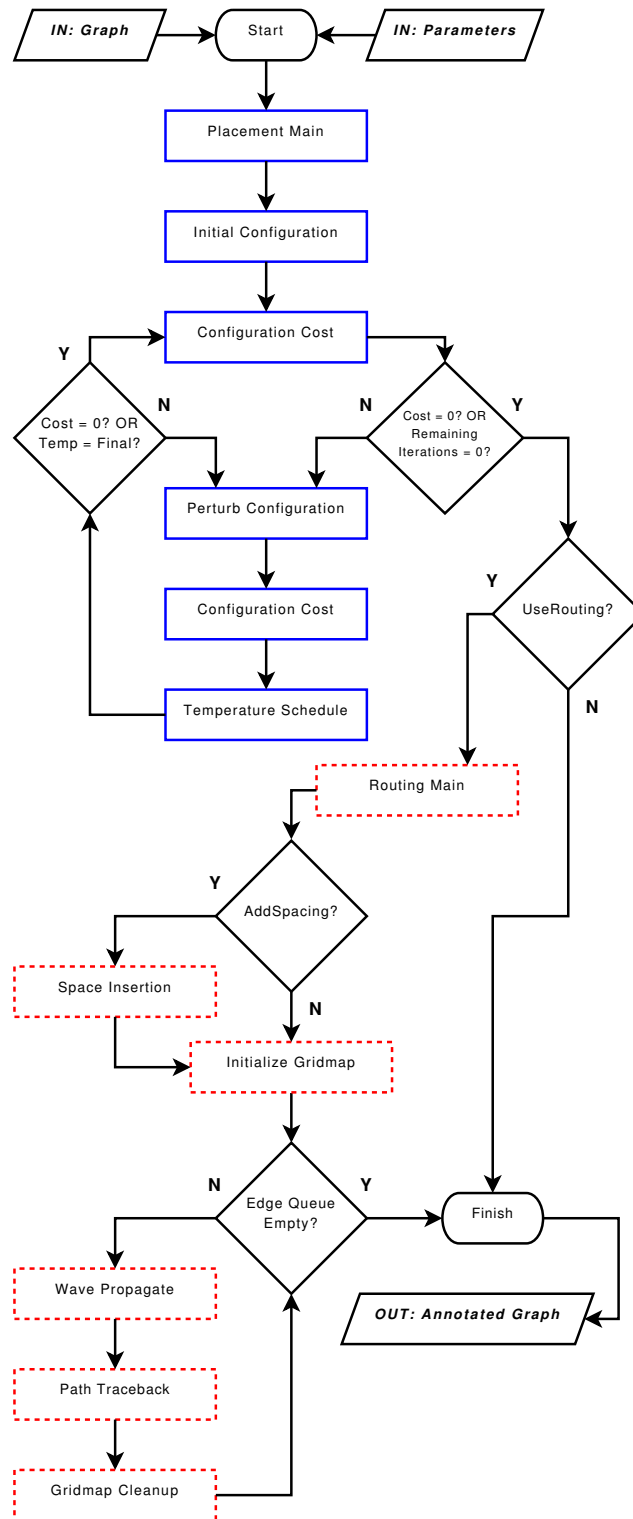


Figure 3.1: Flowchart for the mapping algorithm showing the relationship between major components. The placement phase is colored with solid blue and the routing phase is colored with dotted red.

Property	Description
<i>Graph.Size</i>	The dimensions of the annotated dataflow graph, <i>Graph</i>
<i>Vertex.Category</i>	The type of task object that <i>Vertex</i> represents
<i>Vertex.Coordinate</i>	The processor coordinate assigned to <i>Vertex</i>
<i>Vertex.NoTouch</i>	Flag that determines if <i>Vertex</i> is movable
<i>Vertex.Cost</i>	Temporary cost for prioritizing <i>Vertex</i> during random selections
<i>Edge.Source</i>	Vertex located at the tail-side of the directed edge, <i>Edge</i>
<i>Edge.Target</i>	Vertex located at the head-side of the directed edge, <i>Edge</i>

Table 3.2: List of properties for the *Graph*, *Vertex*, and *Edge* variables.

Property	Description
<i>Config.Input</i>	The input <i>Vertex</i> for the application
<i>Config.InputType</i>	The target edge for the input <i>Vertex</i>
<i>Config.Output</i>	The output <i>Vertex</i> for the application
<i>Config.OutputType</i>	The target edge for the output <i>Vertex</i>
<i>Config.Size</i>	The desired dimensions for the final mapping
<i>Config.QuickPlace</i>	Flag that shortens the placement phase runtime
<i>Config.UseRouting</i>	Flag that enables the routing phase
<i>Config.AddSpacing</i>	Flag that inserts additional space before routing
<i>Config.ExpandType</i>	Expansion sequence type for the initial placement
<i>Config.NumIters</i>	Number of placement phase iterations
<i>Config.MaxRoutes</i>	Maximum number of intersections / router
<i>Config.RandSeed</i>	Initial seed for random number generation
<i>Config.SpaceThreshold</i>	Edge to node percentage ratio for inserting space
<i>Config.CostExcludeMatch</i>	Cost for assigning tasks to excluded locations
<i>Config.CostChannelLength</i>	Cost for using non-nearest neighbor connections
<i>Config.CostInputOutput</i>	Cost for having I/O not along the target edge
<i>Config.CostArraySize</i>	Cost for exceeding the desired dimensions
<i>Config.ExcludeList</i>	List of locations not to assign to tasks
<i>Config.FixedList</i>	List of vertices and their desired locations

Table 3.3: List of properties for the *Config* variable.

input dataflow graph and the output dataflow graph is that the output dataflow graph is annotated with processor locations for each vertex. This means that the coordinate field for each vertex has been assigned a unique processor location. In addition to the annotations new routing vertices have been added to the dataflow graph (if the routing phase was enabled). These new routing vertices are also assigned unique processor locations. It's also worth mentioning that the edges and vertices contain data pointers, which are not listed in Table 3.2 and help programmers synchronize their custom data structures to the dataflow graph object.

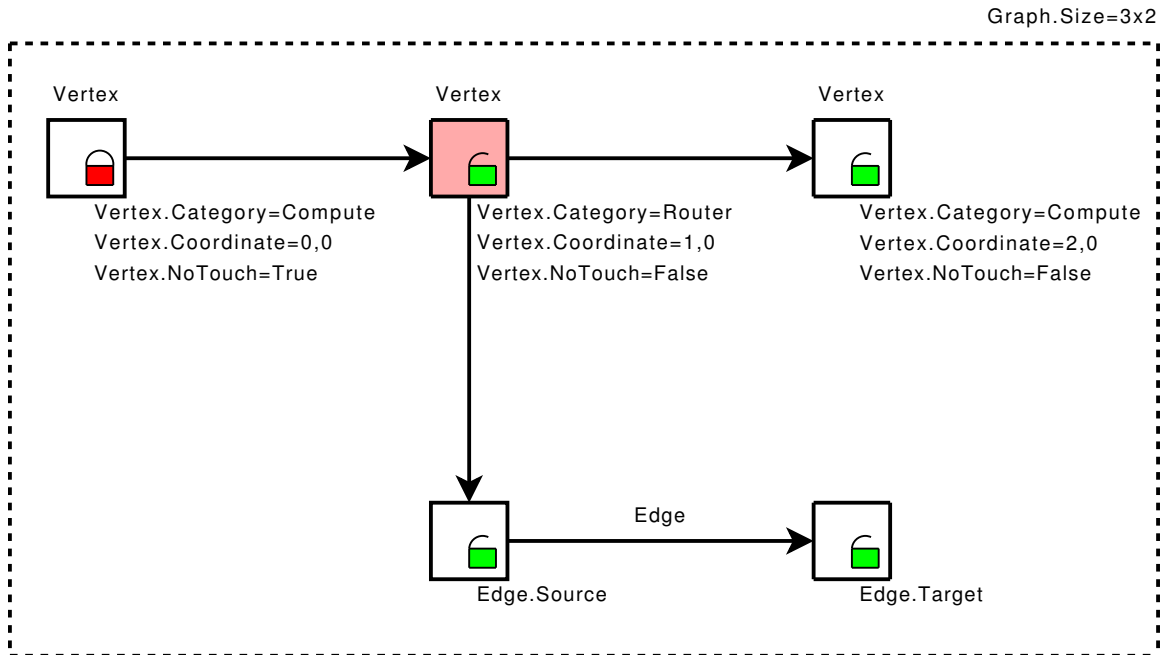


Figure 3.2: Visual representation of the various properties for the *Graph*, *Vertex*, and *Edge* variables. The lock in the bottom-right corner indicates whether or not the vertex is movable. A red closed lock indicates that the vertex is unmovable, while a green open lock indicates that the vertex is movable. The color of the vertex indicates the type of processor represented by the vertex. A vertex colored in white represents a computation processor. A vertex shaded in red represents a routing processor. Arrow heads indicate the direction data is transmitted across an edge, or connection.

3.2 Placement Phase

The placement phase of the mapping algorithm performs the bulk of the work. Much of the basic structure used in the implementation of the placement phase came from the book *Algorithms for VLSI Design Automation* by Gerez [13]. This phase is not only responsible for assigning tasks to processors but it also contains a majority of the optimizations. The placement phase is even responsible for recognizing free processors in key locations to simplify the routing phase. Simulated annealing was chosen as the best candidate for the base algorithm because of its flexibility and performance. This section will discuss the simulated annealing algorithm and the modifications made to solve the mapping problem for nearest neighbor dominated parallel arrays.

Simulated annealing is just one of many possible placement algorithms used by researchers to solve these types of problems. One of these algorithms is the genetic algorithm, which is similar to simulated annealing in that it emulates a physical process, evolution. New configurations are created from parent configurations by producing so called offspring configurations. Only the strongest configurations survive between generations. The genetic algorithm is generally more complex than

simulated annealing and results are often equal in terms of quality. Another possible algorithm is force-directed placement, based on Hook's law ($F = -kx$), where node positions are determined by edge weights. It would have been extremely difficult to implement the various optimizations required by this work if force-directed placement was used. Numerical optimization techniques, such as integer linear programming (ILP), are popular for smaller problem sizes. For these algorithms the problem is described as a set of linear equations then an advanced solver is used to find an optimal solution. Numerical optimization techniques unfortunately aren't scalable. The above mentioned algorithms and other lesser known algorithms have been summarized by Shahookar and Mazumder in their paper *VLSI Cell Placement Techniques* [31].

3.2.1 Simulated Annealing

A number of problems in computer science are classified as NP-complete meaning that the time required to find an optimal solution increases exponentially as the problem size increases. The mapping problem discussed here is classified as NP-complete. To solve these types of problems heuristics are needed to find near-optimal solutions in an acceptable amount of time which does not increase exponentially with the problem size. There are a number of strategies for solving these types of problems but one of the more popular approaches, especially in VLSI, is using iterative improvement [31]. Iterative improvement is a framework more than an algorithm, which can be tailored to solve some NP-complete problems.

Simulated annealing is one form of iterative improvement, modeled after the physical phenomenon of annealing. Kirkpatrick noticed, while exploring statistical mechanics, that as the temperature of a material cooled the most probable atomic configuration was the one with the lowest energy state [19]. Materials that crystallize close to the zero energy state are stronger since the bonds between elements are less likely to break, in other words a near-optimal state. If the pseudo energy state of a configuration can be quantified accurately and the right cooling schedule is used a near-optimal solution can be found. This energy state is typically referred to as the configuration cost, which is a numerical value that can easily be compared. This method has been demonstrated on the traveling salesman problem and many other placement and routing problems from VLSI with good results [19].

The basic principle behind simulated annealing is to perturb a configuration, determine the new configuration's cost, make a decision whether or not to accept the new configuration, and do this iteratively until the configuration converges to a low cost point. To get the algorithm started an initial

configuration is constructed. A new configuration is then created by performing a perturbation on the initial configuration. These two configurations are then compared to see whether or not the new configuration is acceptable. The new configuration is accepted if the configuration cost decreases. A problem that typically occurs when blindly accepting only lower cost configurations is that the algorithm could quickly converge to a local minimum. To overcome this problem simulated annealing will occasionally accept configurations with a higher cost given some probability. The probability of acceptance is determined by the cost difference and the temperature. The smaller the cost difference the more likely the new configuration is accepted. Also the higher the temperature the more likely the new configuration is accepted. The temperature starts out very high where acceptance is likely and the configuration continues to be perturbed until the ground state is reached (zero cost configuration) or the temperature has cooled to a cut-off point.

3.2.2 Algorithm Details

To simulate the annealing process each physical property is represented by a function. Some examples are the perturbation function, which simulates molecular movements, and the cost function, which simulates the energy contained within the molecular bonds. Every function in the simulated annealing algorithm must be tailored to the problem being solved, some more than others. The perturbation and cost functions are examples of highly customized functions while the temperature schedule is an example of a function that remains mostly unchanged. The remainder of this section will discuss how these different functions are combined and optimized for placing tasks onto processors.

Framework

The `PlacementMain` function listed in Algorithm 3.1 contains the basic simulated annealing framework and hence acts as an entry point for the placement phase. This function does very little work in itself and relies on other subfunctions to do the heavy lifting. The main purpose of this function is to setup the mapping, monitor its progress, and terminate the mapping when certain criteria are met. Nevertheless a few optimizations are applied in this function.

This function starts by calling `InitPlacement` to create the initial configuration. After creating the initial configuration all vertices are checked to ensure that the user has not fixed the location of every vertex. Since the initial configuration is the starting point it is also initially the lowest cost configuration. The function then prepares the temperature schedule by calling

`InitialTemp` and `FinalTemp`, which adjusts the schedule based on the difficulty of the problem. The outer loop is then started, which performs the complete annealing process numerous times. Each iteration of the outer loop uses the lowest cost configuration from the previous iteration as a starting point, except the first iteration which uses the initial configuration. Entering the inner loop, the temperature starts at its initial value and `PerturbGraph` is continuously called until the cost becomes zero or the temperature decreases below its final value. The number of times the configuration is perturbed at each temperature step is based on the number of vertices in the graph. Larger problems naturally require more moves to shift around all the vertices.

This function uses two configuration parameters. The first parameter is *Config.QuickPlace*, which decreases the runtime of the placement phase to obtain area and routing estimates. Enabling this parameter decreases the number of perturbs at each temperature step by half. This parameter also enables stall detection, which breaks the inner loop when no improvements are detected for a number of iterations. The second parameter is *Config.NumIters*, which controls how many times the simulated annealing process is executed. This parameter has a significant impact on both the mapping quality and the runtime of the algorithm. By using a higher number for this parameter more of the solution space is explored, which increases the quality but also increases the runtime. After some trial and error the quality shows almost no improvement after the 3rd iteration and sometimes the quality peaks after the 1st iteration.

There are a few constants used by this function. The constant *PERTURB_ITER_BASE* determines the number of perturbs to perform at each temperature step. If this value is too low then the mapping quality will be very poor. If this value is too high then the runtime will be very long. Through trial and error a value of 10 to 20 works best with 15 being the value chosen. Values above 20 could be beneficial for CPUs faster than a 2.0 GHz Xeon processor. The constant *TEMP_DECAY_RATE* determines how quickly the temperature cools which in turn determines the number of temperature steps. The cooling rate can sometimes be a complex function involving changing decay rates but typically a value of around 0.90 is used [31], with a value of 0.85 being used for this work. The cooling rate must be less than 1.0 but also close to 1.0 so that most of the time is spent in the lower temperature range where the majority of the improvements occur. The constant *STALL_CUTOFF_FACTOR* determines how quickly stalls are detected. If this value is too high then the estimation will be very poor since not enough perturbations are performed. If this value is too low then the amount of time saved will be almost nothing. A value of 5 was chosen so that at least 1/5th of the inner loop is executed but time is still saved. A value of 4 would also be

Algorithm 3.1 Placement Phase - PlacementMain

PlacementMain() : assigns vertices in *Graph* to processors in the target array

```

InitPlacement()
if Vertex.NoTouch = true for every Vertex in Graph then
  return
end if
let  $Cost_{min} \leftarrow \text{ConfigCost}(Graph)$ 
let  $Temp_{initial} \leftarrow \text{InitialTemp}(Cost_{min})$ 
let  $Temp_{final} \leftarrow \text{FinalTemp}(Cost_{min})$ 
let  $TempSteps \leftarrow \lceil \log(Temp_{final} / Temp_{initial}) / \log(TEMP\_DECAY\_RATE) \rceil$ 
let  $StallMax \leftarrow TempSteps / STALL\_CUTOFF\_FACTOR$ 
let  $NumPerturbs \leftarrow \text{number of vertices in } Graph \times PERTURB\_ITER\_BASE$ 
if Config.QuickPlace = true then
  let  $NumPerturbs \leftarrow NumPerturbs / 2$ 
end if
for  $Iter \leftarrow 1$  to Config.NumIters do
  if  $Cost_{min} = 0$  then
    return
  end if
  let  $NewGraph \leftarrow Graph$ 
  let  $NewCost \leftarrow Cost_{min}$ 
  let  $StallCount \leftarrow 0$ 
  let  $StallCost \leftarrow Cost_{min}$ 
  let  $Temp \leftarrow Temp_{initial}$ 
  while  $Temp > Temp_{final}$  and  $Cost_{min} > 0$  do
     $PerturbGraph(NewGraph, NewCost, Temp, NumPerturbs, Cost_{min})$ 
    if Config.QuickPlace = true then
      if  $Cost_{min} < StallCost$  then
        let  $StallCount \leftarrow 0$ 
        let  $StallCost \leftarrow Cost_{min}$ 
      end if
      increment  $StallCount$ 
      if  $StallCount \geq StallMax$  then
        break loop
      end if
    end if
    let  $Temp \leftarrow Temp \times TEMP\_DECAY\_RATE$ 
  end while
end for

```

acceptable if the estimate needed to be a little more accurate.

Initial Configuration

The `InitPlacement` and `NextCoord` functions, listed in Algorithm 3.2 and Algorithm 3.3 respectively, are responsible for creating the initial configuration. The `InitPlacement` function returns a configuration with every task in the application assigned to a unique processor location. There are a couple of ways to implement an initial configuration function. A very simple approach is to randomly initialize every element [31]. The method used in this work to initialize the configuration is to sequentially initialize the elements starting from the input vertex, which is beneficial when placing long-chains of connected tasks that form software pipelines. Given more time, I would have liked to try an initial placement routine based on graph planarization. I believe using a form of graph planarization would noticeably improve the mapping quality.

The `InitPlacement` function begins by clearing the graph dimensions and vertex properties so that the target array appears empty. Next a 2D-array, called the marker array, is created to indicate which processors in the target array are allocated. The function starts the allocation process with the fixed location list so that each vertex in this list is assigned its desired location. Next each vertex in this list is tagged as unmovable and its assigned location is marked as unavailable in the marker array. Next each location in the excluded list is set as unavailable in the marker array. The remaining vertices are then sorted using Depth First Search (DFS) where the root of the tree is the primary input vertex. By using DFS long chains are placed consecutively in the target array. Vertices are then sequentially assigned to locations starting at location (0, 0) skipping any unavailable locations in the marker array. As tasks are being assigned locations the size of the target array is updated so that the graph dimensions will be correct when the function returns.

The `NextCoord` function, called by `InitPlacement`, determines the sequence of locations assigned to vertices. There are two possible sequences that can be used. The first sequence starts by traveling down the first column until the bottom edge is reached, then moving to the next column and traveling up the column until the top edge is reached. This snake-like pattern is continued until all tasks are assigned locations. The second sequence starts by traveling right across the first row until the right edge is reached, then moving to the next row and traveling left across the row until the left edge is reached. This snake-like pattern again continues until all tasks are assigned locations. The sequence type used is configurable.

These two functions respond to a few configuration parameters. The *Config.Size* parameter

Algorithm 3.2 Placement Phase - InitPlacement

```

InitPlacement() : assigns coordinates to the vertices in Graph using a sequential pattern
let Graph.Size  $\leftarrow$  Coordinate (-1, -1)
for each Vertex in Graph do
  let Vertex.Category  $\leftarrow$  PROCESSOR
  let Vertex.Coordinate  $\leftarrow$  Coordinate (-1, -1)
  let Vertex.NoTouch  $\leftarrow$  false
end for
let Marker  $\leftarrow$  2D-array of size Config.Size.X  $\times$  Config.Size.Y containing Boolean values
for each Vertex, Coord in Config.FixedList do
  let Vertex.Coordinate  $\leftarrow$  Coord
  let Vertex.NoTouch  $\leftarrow$  true
  let Marker[Coord]  $\leftarrow$  true
  let Graph.Size  $\leftarrow$  updated bounding box
end for
for each Coord in Config.ExcludeList do
  let Marker[Coord]  $\leftarrow$  true
end for
let Order  $\leftarrow$  unassigned vertices in Graph sorted using DFS, starting at Config.Input
let Coord  $\leftarrow$  Coordinate (0, 0)
while Order is not empty do
  if Marker[Coord] = false then
    let Vertex  $\leftarrow$  next vertex from Order
    let Vertex.Coordinate  $\leftarrow$  Coord
    let Graph.Size  $\leftarrow$  updated bounding box
  end if
  NextCoord(Coord)
end while

```

determines the desired size for the final array assuming no routing processors are needed. This parameter is used to create the marker array and also determines where the two location sequences wrap. The *Config.ExpandType* parameter determines which location sequence type is used, either vertical expansion or horizontal expansion. For most applications the sequence type doesn't really matter. Vertical expansion is helpful for long chains where the output is on the right edge. Horizontal expansion is helpful for problems that are much smaller than the target array but the output is on the right. The *Config.FixedList* parameter is a list of vertex and coordinate pairs. This list is used to ensure that tasks are placed in their desired location before another task has a chance to occupy it. The *Config.ExcludeList* parameter is a list of locations within the target array that should never have tasks assigned to them. This list is used to ensure that the marker array blocks the locations in this list correctly. The final parameter *Config.Input* determines which vertex in the dataflow graph is the input.

The constants *PROCESSOR*, *VERTICAL*, and *HORIZONTAL* are simply enumerations used to distinguish the type of an object or operation.

Algorithm 3.3 Placement Phase - NextCoord

NextCoord(*Coord*) : calculates the next coordinate after *Coord* in the expansion sequence

```

if Config.ExpandType = VERTICAL then
  if Coord.X is even then
    let Coord.Y  $\leftarrow$  Coord.Y + 1
  else
    let Coord.Y  $\leftarrow$  Coord.Y - 1
  end if
  if Coord.Y < 0 then
    let Coord.Y  $\leftarrow$  0
    let Coord.X  $\leftarrow$  Coord.X + 1
  end if
  if Coord.Y > Config.Size.Y - 1 then
    let Coord.Y  $\leftarrow$  Config.Size.Y - 1
    let Coord.X  $\leftarrow$  Coord.X + 1
  end if
else
  if Coord.Y is even then
    let Coord.X  $\leftarrow$  Coord.X + 1
  else
    let Coord.X  $\leftarrow$  Coord.X - 1
  end if
  if Coord.X < 0 then
    let Coord.X  $\leftarrow$  0
    let Coord.Y  $\leftarrow$  Coord.Y + 1
  end if
  if Coord.X > Config.Size.X - 1 then
    let Coord.X  $\leftarrow$  Config.Size.X - 1
    let Coord.Y  $\leftarrow$  Coord.Y + 1
  end if
end if

```

Configuration Cost

The functions `ConfigCost`, `PathRoutable`, `InputCost`, `OutputCost`, and `ArrayCost`, listed in Algorithm 3.4, Algorithm 3.5, Algorithm 3.6, Algorithm 3.7, and Algorithm 3.8 respectively, calculate the configuration cost. This operation is the most complex operation in the mapping algorithm. These functions must evaluate the configuration and return some numerical value that rates the goodness of the configuration. Since estimating the configuration cost is a complex operation, the process is broken down into several parts. Each part evaluates one attribute of the configuration. The results from each part are summed together to make up the configuration cost. If these functions are not properly tuned it will be difficult to determine if a perturbation improves or deteriorates the configuration. What makes this process so difficult to tune properly is deciding the correct weights to apply to each attribute to ensure they are prioritized correctly.

The `ConfigCost` function is responsible for combining the various costs as well as calculating communication related costs. To start a 2D-array is created, called the marker array, that indicates which processors already have tasks assigned to them. Next the location for each vertex in the graph is marked as unavailable in the marker array. Then each location in the excluded location list is marked as unavailable. If any location in the excluded location list was previously marked as unavailable a constant is added to the total cost since these locations should not have tasks assigned to them. Next communication related costs are calculated. Nearest neighbor connections are ideal so they can be skipped since their cost contribution would be zero. The cost for using a long-distance connection is linearly related to the Manhattan distance between the source and target vertices. Although if a simple route is found between the source and target, after calling `PathRoutable`, then this cost increase is reduced. Costs associated with the other attributes of the configuration are calculated by the functions `InputCost`, `OutputCost`, and `ArrayCost`. The configuration cost can also be augmented by an user cost function if implemented. The final cost returned by this function accounts for all the attributes of the configuration.

In a previous version of the mapping algorithm there was a path intersection detection feature. The communication cost was increased if the rectangular bounding boxes for two edges intersected. Figure 3.3 depicts how path intersections are determined. After testing this feature on a few applications the mapping quality didn't improve and the runtime took a big hit so this feature was removed. Another attempt was made using a more accurate line intersection algorithm, using basic algebra, but this method actually decreased the mapping quality. Intersecting lines were removed as expected, but the lines instead started becoming parallel making it difficult to route since many connections traveled along the same row or column.

The `PathRoutable` function checks the coordinates along a path against the marker array (from the `ConfigCost` function) to see if simple routes can be formed without any obstructions. There are three types of routable paths. The first type requires that the source and target coordinates share the same column number. If so the coordinates of intermediate rows along the shared column are inserted into the path queue. The second type requires that the source and target coordinates share the same row number. If so the coordinates of intermediate columns along the shared row are inserted into the path queue. The third type requires that the source and target coordinates are diagonal and that the X and Y coordinates both have a difference of 1 (also called diagonal nearest neighbor). If so the two coordinates that are adjacent to both the source and target are inserted into the path queue. After a possible path has been determined, and its coordinates have

Algorithm 3.4 Placement Phase - ConfigCost

```

ConfigCost(NewGraph) = Cost : evaluates the configuration NewGraph and returns the
configuration cost in Cost

let Cost  $\leftarrow$  0
let Marker  $\leftarrow$  2D-array of size NewGraph.Size.X  $\times$  NewGraph.Size.Y containing Boolean values
for each Vertex in NewGraph do
  let Coord  $\leftarrow$  Vertex.Coordinate
  let Marker[Coord]  $\leftarrow$  true
end for
for each Coord in Config.ExcludeList do
  if Marker[Coord] = true then
    let Cost  $\leftarrow$  Cost + Config.CostExcludeMatch
  end if
  let Marker[Coord]  $\leftarrow$  true
end for
for each Edge in NewGraph do
  if Edge.Source and Edge.Target are nearest neighbors then
    next iteration
  end if
  let Length  $\leftarrow$  (distance from Edge.Source to Edge.Target) - 1
  let Costadd  $\leftarrow$  Config.CostChannelLength
  let Costadd  $\leftarrow$  Costadd  $\times$  Length
  if Config.UseRouting = true then
    if PathRoutable(Marker, Edge) = true then
      let Costadd  $\leftarrow$  Costadd / PATH_ROUTABLE_FACTOR
    end if
  end if
  let Cost  $\leftarrow$  Cost + Costadd
end for
let Cost  $\leftarrow$  Cost + InputCost(NewGraph)
let Cost  $\leftarrow$  Cost + OutputCost(NewGraph)
let Cost  $\leftarrow$  Cost + ArrayCost(NewGraph)
if UserCost is defined then
  let Cost  $\leftarrow$  Cost + UserCost(NewGraph)
end if

```

been inserted into the path queue, each coordinate along the path is checked to see if it is available in the marker array. For paths that traverse a single column or row, a single match will make the path unroutable. For diagonal paths there are two possibilities so both coordinates must match for the path to be unroutable. Even though a simple path can be found it doesn't necessarily mean the path is routable since there could be a conflict with another route. Also, if a simple path can not be found this doesn't necessarily mean the path is unroutable.

The *InputCost* and *OutputCost* functions, which are closely related, calculate costs associated with the primary input and primary output, respectively. There are five possible targets for these two vertices. The first possibility is along the left edge (which is typically used by the input)

Algorithm 3.5 Placement Phase - PathRoutable

PathRoutable(*Marker*, *Edge*) = *Routable* : checks coordinates in *Marker* that are along *Edge* and sets *Routable* to **true** if the coordinates are unblocked

```

let SingleMatch ← true
let Source ← Edge.Source
let Target ← Edge.Target
let Path ← queue that contains coordinates
if Source.X = Target.X then
  if Target.Y > Source.Y then
    let Step ← +1
  else
    let Step ← -1
  end if
  for Y ← (Source.Y + Step) to (Target.Y - Step) do
    insert Coordinate (Source.X, Y) into queue Path
  end for
else if Source.Y = Target.Y then
  if Target.X > Source.X then
    let Step ← +1
  else
    let Step ← -1
  end if
  for X ← (Source.X + Step) to (Target.X - Step) do
    insert Coordinate (X, Source.Y) into queue Path
  end for
else if Source is diagonal nearest neighbor to Target then
  insert Coordinate (Target.X, Source.Y) into queue Path
  insert Coordinate (Source.X, Target.Y) into queue Path
  let SingleMatch ← false
else
  return false
end if
let NumMatches ← 0
for each Coord in Path do
  if Marker[Coord] = true then
    increment NumMatches
  end if
end for
if SingleMatch = true then
  return (NumMatches = 0)
end if
return (NumMatches < length of queue Path)

```

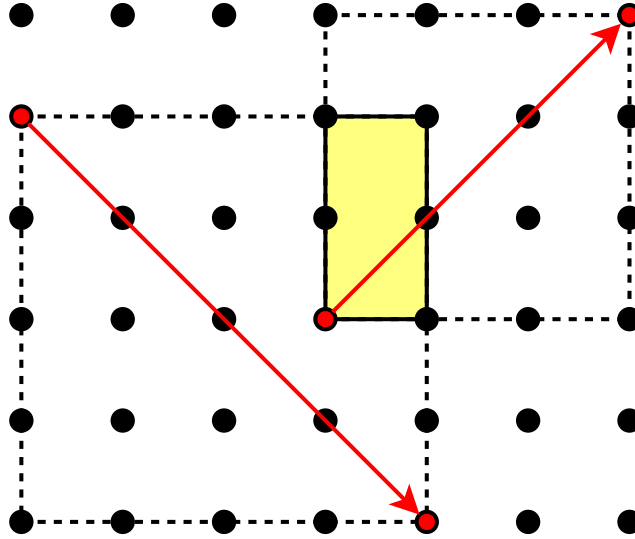


Figure 3.3: Depiction of how path intersections are determined using rectangular bounding boxes. In this example the red lines would be considered intersecting within the shaded yellow region.

where the cost is dependent upon the distance the X coordinate is from 0. The second possibility is along the right edge (which is typically used by the output) where the cost is dependent upon the distance the X coordinate is from the desired right edge. The desired right edge is either the current array width or the desired array width, whichever is larger. The third possibility is along the top edge where the cost is dependent upon the distance the Y coordinate is from 0. The fourth possibility is along the bottom edge where the cost is dependent upon the distance the Y coordinate is from the desired bottom edge. The desired bottom edge is either the current array height or the desired array height, whichever is larger. The fifth possibility is to completely ignore the vertex, which means the cost increase is zero no matter where the vertex is positioned. The five possible targets are depicted in Figure 3.4. Whatever cost increase is decided upon is multiplied by a constant and squared before being returned where it is combined with the total configuration cost.

The `ArrayCost` function calculates the cost associated with the size of the array. There are two components to this cost. The array size in the X dimension and the array size in the Y dimension. First the current array size in the X dimension is compared to the desired array size in the X dimension. If the current X dimension is larger than the desired X dimension then the cost is increased. The cost increase is exponential where the base is a small constant and the power is the difference between the X dimensions. Next the current array size in the Y dimension is compared to the desired array size in the Y dimension. If the current Y dimension is larger than the desired Y dimension then the cost is increased. The cost increase is again exponential where the base is a small constant and the power is the difference between the Y dimensions. When the current array

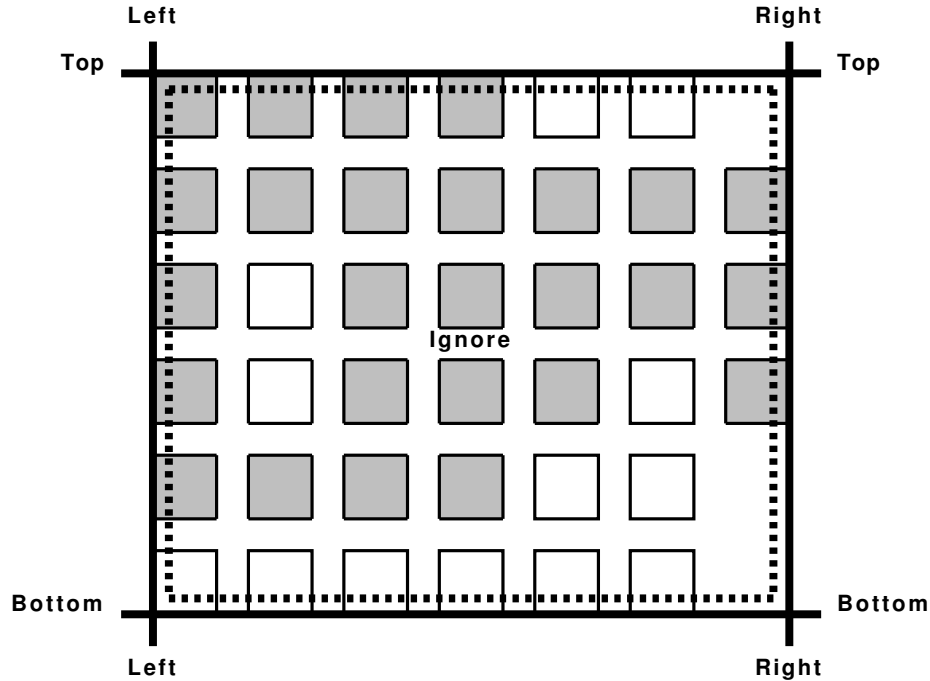


Figure 3.4: Depiction of the five possible targets for the primary input and the primary output. The current mapping for this application is shown in gray and unassigned processors in the target array are shown in white. In this example the desired array height is larger than the current array height, but the desired array width is less than the current array width.

size is smaller than the desired array size the cost increase is zero.

A large number of parameters are used in determining the configuration cost. The *Config.ExcludeList* parameter is a list of locations within the target array that should never have tasks assigned to them. Tasks can temporarily be assigned to these excluded locations, to help move tasks around the array, but there is a large penalty for using these excluded locations. This list is also used to properly initialize the marker array, which is used for routable path calculations. The *Config.UseRouting* parameter is used to enable or disable routable path calculations, which can impact the runtime. If there are no plans to insert routing processors then there is no need to distinguish between routable and unroutable paths. The *Config.Size* parameter contains the desired size for the target array. This parameter is primarily used for calculating array size related costs, but it is also used for finding the desired right and bottom edges for primary input and primary output related costs. The *Config.Input* parameter determines which vertex is the primary input. The *Config.InputType* parameter determines the desired edge for the primary input. The *Config.Output* parameter determines which vertex is the primary output. The *Config.OutputType* parameter determines the desired edge for the primary output.

Algorithm 3.6 Placement Phase - InputCost

```

InputCost(NewGraph) = Cost : calculates the cost associated with the input vertex for
NewGraph and returns the cost in Cost

let Costadd  $\leftarrow$  0
let RightEdge  $\leftarrow$  max(NewGraph.Size.X - 1, Config.Size.X - 1)
let BottomEdge  $\leftarrow$  max(NewGraph.Size.Y - 1, Config.Size.Y - 1)
let Coord  $\leftarrow$  coordinate for vertex Config.Input of NewGraph
if Config.InputType = LEFT then
  if Coord.X > 0 then
    let Costadd  $\leftarrow$  Config.CostInputOutput
    let Costadd  $\leftarrow$  Costadd  $\times$  Coord.X
  end if
else if Config.InputType = RIGHT then
  if Coord.X < RightEdge then
    let Costadd  $\leftarrow$  Config.CostInputOutput
    let Costadd  $\leftarrow$  Costadd  $\times$  (RightEdge - Coord.X)
  end if
else if Config.InputType = TOP then
  if Coord.Y > 0 then
    let Costadd  $\leftarrow$  Config.CostInputOutput
    let Costadd  $\leftarrow$  Costadd  $\times$  Coord.Y
  end if
else if Config.InputType = BOTTOM then
  if Coord.Y < BottomEdge then
    let Costadd  $\leftarrow$  Config.CostInputOutput
    let Costadd  $\leftarrow$  Costadd  $\times$  (BottomEdge - Coord.Y)
  end if
end if
return (Costadd  $\times$  Costadd)

```

There are four additional parameters which serve a different purpose than the previously listed parameters. These four parameters assign weights to the different cost attributes. This is where most of the fine tuning takes place. The first parameter is *Config.CostChannelLength*, which accounts for the presence of non-nearest neighbor connections. Longer connections are less ideal so the cost for using them increases linearly with the connection length. This cost weight serves as somewhat of a basis for the other cost weights. A value of 20 was chosen, which is small enough to avoid frequent overflows, but large enough that a number of integer fractions can be created. The second parameter is *Config.CostExcludeMatch*, which accounts for tasks assigned to excluded locations. For each matching location the cost is increased by a constant amount since a match is either a true or false event. Since using excluded locations is allowed but highly undesirable this cost weight is very large relative to other cost weights. After some trial and error this value needs to be at least 5 times larger than the basis with a value of 100 being chosen. The third parameter is *Config.CostInputOutput*, which accounts for the distance the input and output vertices are from

Algorithm 3.7 Placement Phase - OutputCost

OutputCost(*NewGraph*) = *Cost* : calculates the cost associated with the output vertex for *NewGraph* and returns the cost in *Cost*

let $Cost_{add} \leftarrow 0$
let $RightEdge \leftarrow \max(NewGraph.Size.X - 1, Config.Size.X - 1)$
let $BottomEdge \leftarrow \max(NewGraph.Size.Y - 1, Config.Size.Y - 1)$
let *Coord* \leftarrow coordinate for vertex *Config.Output* of *NewGraph*
if *Config.OutputType* = *LEFT* **then**
 if *Coord.X* > 0 **then**
 let $Cost_{add} \leftarrow Config.CostInputOutput$
 let $Cost_{add} \leftarrow Cost_{add} \times Coord.X$
 end if
else if *Config.OutputType* = *RIGHT* **then**
 if *Coord.X* < *RightEdge* **then**
 let $Cost_{add} \leftarrow Config.CostInputOutput$
 let $Cost_{add} \leftarrow Cost_{add} \times (RightEdge - Coord.X)$
 end if
else if *Config.OutputType* = *TOP* **then**
 if *Coord.Y* > 0 **then**
 let $Cost_{add} \leftarrow Config.CostInputOutput$
 let $Cost_{add} \leftarrow Cost_{add} \times Coord.Y$
 end if
else if *Config.OutputType* = *BOTTOM* **then**
 if *Coord.Y* < *BottomEdge* **then**
 let $Cost_{add} \leftarrow Config.CostInputOutput$
 let $Cost_{add} \leftarrow Cost_{add} \times (BottomEdge - Coord.Y)$
 end if
end if
return ($Cost_{add} \times Cost_{add}$)

Algorithm 3.8 Placement Phase - ArrayCost

ArrayCost(*NewGraph*) = *Cost* : calculates the cost associated with the array size for *NewGraph* and returns the cost in *Cost*

let $Cost_{addX} \leftarrow 0$
let $Cost_{addY} \leftarrow 0$
if *NewGraph.Size.X* > *Config.Size.X* **then**
 let $Cost_{base} \leftarrow Config.CostArraySize$
 let $Cost_{exp} \leftarrow NewGraph.Size.X - Config.Size.X$
 let $Cost_{addX} \leftarrow \text{pow}(Cost_{base}, Cost_{exp})$
end if
if *NewGraph.Size.Y* > *Config.Size.Y* **then**
 let $Cost_{base} \leftarrow Config.CostArraySize$
 let $Cost_{exp} \leftarrow NewGraph.Size.Y - Config.Size.Y$
 let $Cost_{addY} \leftarrow \text{pow}(Cost_{base}, Cost_{exp})$
end if
return ($Cost_{addX} + Cost_{addY}$)

their desired edges. Since the cost for these attributes is only included once the value is squared to signify its importance. Trial and error has shown that this attribute can grow rather quickly so a value less than the basis was needed with a value of 10 being chosen. The fourth and final parameter is *Config.CostArraySize*, which accounts for the size of the array. It is important that this attribute (which is often the most important attribute) doesn't get overwhelmed by other attributes so it is emphasized using an exponential increase. As expected, this attribute grows very quickly so a value of 5 was chosen, which is much smaller than the basis. One side effect of using an exponential increase with a small base is that when the dimensions are over by only a small amount the cost increase is relatively small compared to other attributes. This makes the array boundaries less rigid but has been shown to produce better mappings for some applications because of the extra row or column when positioning the primary input and primary output.

Only a few constants are used in determining the configuration cost since most of the tuning is done by the configuration parameters. The constant *PATH_ROUTABLE_FACTOR* is used to reduce the communication cost to a fraction of its original value when an easily routable path can be found. If this value is too high the reduced cost will be very close to zero meaning that a routable path is nearly identical to being nearest neighbor. If this value is too low the benefit of finding routable paths would be questionable. This would be similar to what happens when the *Config.UseRouting* parameter is set to false, except that when using this parameter the runtime is also reduced. After some trial and error a value of 4 to 5 prioritizes routable paths best, with 5 being the value chosen. The constants *LEFT*, *RIGHT*, *TOP*, and *BOTTOM* are simply enumerations used to distinguish the type of operation being performed.

Temperature Schedule

The temperature schedule is broken down into three parts, the initial temperature, the final temperature, and the cooling rate. The cooling rate has already been discussed in the `PlacementMain` function on page 18. The initial and final temperature points can be determined in a number of ways. One method is to use constant temperature points [19]. Another method is to calculate these temperature points [14]. For this work the initial and final temperature points are calculated. The advantage to using calculated temperature points is that the temperature schedule is aware of the problem's difficulty level. The initial and final temperature points are calculated by the functions `InitialTemp` and `FinalTemp`, listed in Algorithm 3.9 and Algorithm 3.10 respectively. The way the temperature schedule is chosen has a significant impact on the runtime of the algorithm. When the

number of temperature steps increases more perturbations are executed and as a result the runtime increases. When the initial temperature is extremely high the number of temperature steps is very large. When the final temperature is extremely low the number of temperature steps is again very large. Also when the temperature cooling rate is very close to 1.0 the temperature cools very slowly and once again the number of temperature steps is very large. As a result of these trade-offs the initial and final temperatures must be chosen carefully so that the core temperature range is used more efficiently.

The `InitialTemp` and `FinalTemp` functions are very similar. Both functions perform a series of perturbations, while changing the temperature, until the number of perturbations accepted reaches a certain level. The point of this procedure is to adjust the temperature schedule to the difficulty of the problem. To start the temperature is set to an initial value. The configuration is then duplicated so that tests can be performed on the configuration without disturbing the original configuration. The main loop then begins executing blocks of perturbations. The temperature increases each iteration when searching for the initial temperature and decreases each iteration when searching for the final temperature. For each iteration the configuration is perturbed 100 times and the number of perturbations that are accepted is called the acceptance level. If the acceptance level satisfies the given condition the main loop exits and the temperature is returned. For the final temperature this condition is a minimum acceptance level. For the initial temperature this condition is a maximum acceptance level.

Algorithm 3.9 Placement Phase - InitialTemp

```

InitialTemp( $Cost_{min}$ ) =  $Temp$  : increases the initial temperature  $Temp$  until a minimum
number of perturbations are accepted

let  $Temp \leftarrow TEMP\_INIT\_MIN$ 
let  $NewGraph \leftarrow Graph$ 
let  $NewCost \leftarrow Cost_{min}$ 
for  $I \leftarrow 1$  to  $TEMP\_CHANGE\_COUNT$  do
  let  $Accepted \leftarrow PerturbGraph(NewGraph, NewCost, Temp, 100, Cost_{min})$ 
  if  $Accepted \geq TEMP\_INIT\_ACCEPT$  then
    break loop
  end if
  let  $Temp \leftarrow Temp \times TEMP\_CHANGE\_STEP$ 
end for
return  $Temp$ 

```

A number of constants are used when calculating the initial and final temperature points. The constants `TEMP_INIT_MIN` and `TEMP_INIT_ACCEPT` are critical in determining the initial temperature. The first constant determines the lowest possible initial temperature. The second

Algorithm 3.10 Placement Phase - FinalTemp

```

FinalTemp( $Cost_{min}$ ) =  $Temp$  : decreases the final temperature  $Temp$  until a maximum number
of perturbs are accepted

let  $Temp \leftarrow TEMP\_FINAL\_MAX$ 
let  $NewGraph \leftarrow Graph$ 
let  $NewCost \leftarrow Cost_{min}$ 
for  $I \leftarrow 1$  to  $TEMP\_CHANGE\_COUNT$  do
  let  $Accepted \leftarrow PerturbGraph(NewGraph, NewCost, Temp, 100, Cost_{min})$ 
  if  $Accepted \leq TEMP\_FINAL\_ACCEPT$  then
    break loop
  end if
  let  $Temp \leftarrow Temp / TEMP\_CHANGE\_STEP$ 
end for
return  $Temp$ 

```

constant determines the minimum number of perturbs that must be accepted in order to accept the initial temperature. The initial temperature should be high enough to initially accept almost all perturbations and avoid getting stuck in a local minimum. Though if the initial temperature is too high then iterations will be wasted simply around bouncing between local minimums. There is still some uncertainty in what the best values are for these constants, but the values 100 and 95, respectively, seem to work. The constants $TEMP_FINAL_MAX$ and $TEMP_FINAL_ACCEPT$ are critical in determining the final temperature. The first constant determines the highest possible final temperature. The second constant determines the maximum number of perturbs that must be accepted in order to accept the final temperature. The final temperature should be low enough that only improvements in cost are accepted in order to reach the global minimum. Though if the final temperature is too low then iterations will be wasted since the lowest cost solution has already been reached at higher temperatures. There is still some uncertainty in what the best values are for these constants as well, but the values 0.01 and 10, respectively, seem to work. There could be some benefit to using a maximum final temperature of 0.001 for CPUs much faster than a 2.0 GHz Xeon processor. The constants $TEMP_CHANGE_STEP$ and $TEMP_CHANGE_COUNT$ determine the absolute upper and lower limits for the temperature schedule. The first constant determines the rate that the temperature points increase or decrease in the acceptance loops. The second constant determines the maximum number of iterations in the acceptance loops. If the temperature changes too quickly or is allowed to change too many times the initial and final temperatures could reach extremes, thereby increasing the runtime significantly. If the temperature changes too slowly or is only allowed to change a few times the temperature schedule will not have the range needed to obtain the best quality possible. There is still some uncertainty in what the best values are for these

Property	Description
<i>Perturb.MinX</i>	Minimum X value for the target search range
<i>Perturb.MaxX</i>	Maximum X value for the target search range
<i>Perturb.MinY</i>	Minimum Y value for the target search range
<i>Perturb.MaxY</i>	Maximum Y value for the target search range
<i>Perturb.SrcVertex</i>	The source vertex for the <i>Perturb</i>
<i>Perturb.DstVertex</i>	The target vertex for the <i>Perturb</i>
<i>Perturb.SrcCoord</i>	The source location for the <i>Perturb</i>
<i>Perturb.DstCoord</i>	The target location for the <i>Perturb</i>
<i>Perturb.History</i>	List of previously tried target locations

Table 3.4: List of properties for the *Perturb* variable.

constants, but the values 5 and 100, respectively, seem to work well.

Perturb Configuration

Perturbing the configuration is a complex process and it's where all the work actually occurs. Like with the configuration cost, the perturb configuration process is broken down into many functions. These functions are `PerturbGraph`, `PerturbSetup`, `RandVertex`, `PerturbApply`, and `PerturbUndo`, listed in Algorithm 3.11, Algorithm 3.12, Algorithm 3.13, Algorithm 3.14, and Algorithm 3.15 respectively. The configuration is perturbed using a series of moves with the intention of lowering the configuration cost. An important requirement when developing moves is that any configuration can be transformed into any other configuration using a series of these moves. These moves can be as simple as swapping two elements or moving an element to an unused location. Choosing which elements to operate on is a complex operation in itself and is what makes the perturb configuration process so complex.

The global variable *Perturb*, which is used when perturbing the configuration, tracks task movements so they can be reversed if the modified configuration is not accepted. The properties for this global variable are listed in Table 3.4. The types of values stored in each field and their uses are described in further detail in the remainder of this subsection. The lifetime of the *Perturb* variable is very limited and only exists when the configuration is in an unaccepted state. Reversing perturbations is a simple way to save runtime. Alternatively, the entire graph could be duplicated before each move, which is a costly operation, then restored if the perturbation is rejected. The runtime required to prepare a perturbation is also costly, which is saved by reversing the perturbation then applying a variant of the perturbation.

The `PerturbGraph` function is responsible for executing a block of perturbs while counting the number of perturbs that are accepted. To begin a new perturb object is created. A series of

attempts are then made to try and improve the configuration. This involves applying variations of the perturb, based on the perturb object's data, and evaluating how the configuration cost changes. If the perturb variation fails to be applied, which can happen if the target for a swap operation is unmovable, the current attempt is aborted and a new attempt is made. If the new configuration has a lower cost than the global minimum then the global minimum is replaced by the new configuration. Doing this ensures that the global minimum is always the lowest cost configuration. The new configuration is also compared to the old configuration to determine whether or not the new configuration should be accepted. The probability of being accepted depends upon the cost difference between the two configurations and the temperature. The smaller the difference in cost or the higher the temperature the more likely the configuration is going to be accepted. The new configuration will be accepted if the new configuration cost is less than or equal to the old configuration cost regardless of the temperature value. There are a number of ways to calculate the acceptance probability whenever the new configuration cost is more than the old configuration cost, but the most common method is to use Equation 3.1 [14,31]. The result from this equation is then compared to a random number between 0.0 and 1.0. The new configuration is accepted if the result is greater than the random number. If the configuration is accepted then the acceptance counter is incremented and a new perturb object is created. If the configuration is not accepted then the perturb is undone and another perturb variation is applied. In the end the function returns the number of perturbs that were accepted.

$$Acceptance = \exp\left(-\frac{Cost_{new} - Cost_{old}}{T}\right) \quad (3.1)$$

As the placement phase progresses and the configuration is perturbed, the global minimum slowly improves. Figure 3.5 shows a plot of the minimum and current configuration costs over three iterations of the temperature schedule for the 802.11a wireless transmitter, further described in Chapter 5. The current configuration cost is the cost of the configuration after returning each time from the `PerturbGraph` function. The minimum configuration cost is the cost for the global minimum configuration, which is updated each time a lower cost configuration is observed during the `PerturbGraph` function.

The `PerturbSetup` function begins by initializing a new perturb object so that perturb variations can efficiently be constructed. The first step is to randomly choose a vertex to serve as the source for new perturb variations by calling `RandVertex`. The source vertex and its current location, referred to as the source location, are stored inside the new perturb object. Next the search range is calculated by adding and subtracting a constant from the source location's X and

Algorithm 3.11 Placement Phase - PerturbGraph

PerturbGraph(*NewGraph*, *NewCost*, *Temp*, *Count*, $Cost_{min}$) = *Accepted* : perturbs the configuration *NewGraph* *Count* times and returns the number of perturbs accepted, also updates *Graph* and $Cost_{min}$ if a new absolute minimum is found

```

let Accepted  $\leftarrow$  0
for Iter  $\leftarrow$  1 to Count do
  PerturbSetup(NewGraph)
  for Trial  $\leftarrow$  1 to PERTURB_TRIAL_MAX do
    if PerturbApply(NewGraph) = false then
      next iteration
    end if
    let PerturbCost  $\leftarrow$  ConfigCost(NewGraph)
    if PerturbCost <  $Cost_{min}$  then
      let Graph  $\leftarrow$  NewGraph
      let  $Cost_{min}$   $\leftarrow$  PerturbCost
    end if
    let Acceptance  $\leftarrow$   $\exp((NewCost - PerturbCost) / Temp)$ 
    let Rand  $\leftarrow$  random number from 0.0 up to 1.0
    if PerturbCost  $\leq$  NewCost or Acceptance > Rand then
      let NewCost  $\leftarrow$  PerturbCost
      increment Accepted
      break loop
    end if
  PerturbUndo(NewGraph)
end for
end for
return Accepted

```

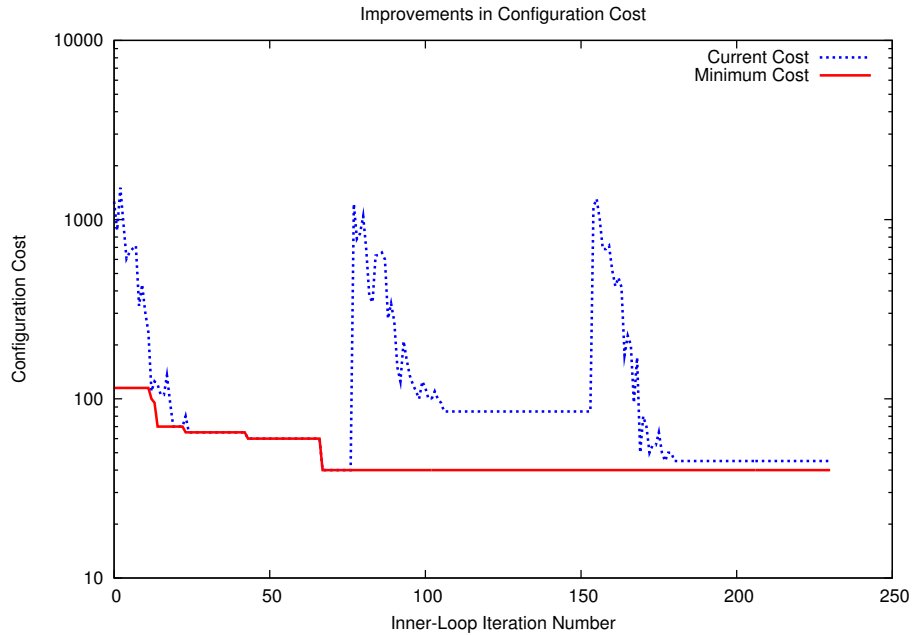


Figure 3.5: Improvements in minimum configuration cost along with the current configuration cost for three iterations of the temperature schedule while mapping the 802.11a wireless transmitter

Y values to obtain lower and upper search boundaries. If the lower boundary for the X dimension extends beyond the left edge of the array then the boundary is truncated at the left edge. If the lower boundary for the Y dimension extends beyond the upper edge of the array then the boundary is truncated at the upper edge. If the upper boundary for the X dimension extends more than one column beyond the right edge of the array then the boundary is truncated at one column beyond the right edge. If the upper boundary for the Y dimension extends more than one row beyond the bottom edge of the array then the boundary is truncated at one row beyond the bottom edge. Once the search range has been corrected it is stored inside the new perturb object.

Algorithm 3.12 Placement Phase - PerturbSetup

PerturbSetup(*NewGraph*) : creates and initializes a new *Perturb* object from *NewGraph*

```

create new global object Perturb
let Perturb.History  $\leftarrow$  queue that contains coordinates
let Perturb.SrcVertex  $\leftarrow$  RandVertex(NewGraph)
let Perturb.SrcCoord  $\leftarrow$  Perturb.SrcVertex.Coordinate
let Perturb.MinX  $\leftarrow$  Perturb.SrcCoord.X - PERTURB_RANGE
let Perturb.MaxX  $\leftarrow$  Perturb.SrcCoord.X + PERTURB_RANGE
let Perturb.MinY  $\leftarrow$  Perturb.SrcCoord.Y - PERTURB_RANGE
let Perturb.MaxY  $\leftarrow$  Perturb.SrcCoord.Y + PERTURB_RANGE
if Perturb.MinX < 0 then
  let Perturb.MinX  $\leftarrow$  0
end if
if Perturb.MinY < 0 then
  let Perturb.MinY  $\leftarrow$  0
end if
if Perturb.MaxX > NewGraph.Size.X then
  let Perturb.MaxX  $\leftarrow$  NewGraph.Size.X
end if
if Perturb.MaxY > NewGraph.Size.Y then
  let Perturb.MaxY  $\leftarrow$  NewGraph.Size.Y
end if

```

The **RandVertex** function is responsible for randomly selecting a vertex which will be a good candidate for perturbations. To begin each vertex is assigned a cost based on its connectivity. The connectivity cost for a vertex is calculated by summing the length of every output edge. Any vertices that are unmovable can not be selected and are skipped. A random number is then generated between 0 and the largest connectivity cost observed. The cost for each vertex is then compared to this random number. Vertices with a cost greater than or equal to this random number are put into a selection queue. From this queue a random element is selected, which then becomes the perturb object's source vertex. By using this method vertices with longer connections are more likely to be selected, but vertices with shorter connections still have the possibility of being selected.

Vertices with shorter connections are usually already nearest neighbor and therefore don't need to be perturbed as much.

Algorithm 3.13 Placement Phase - RandVertex

RandVertex(*NewGraph*) = *Vertex* : selects a random vertex from *NewGraph* prioritized by the total distance to connected vertices and returns this vertex in *Vertex*

```

let MaxCost  $\leftarrow$  0
for each Vertex in NewGraph do
  if Vertex.NoTouch = true then
    next iteration
  end if
  let Cost  $\leftarrow$  0
  for each output edge OutEdge of Vertex do
    let Length  $\leftarrow$  distance from Vertex to OutEdge.Target
    let Cost  $\leftarrow$  Cost + Length
  end for
  let Vertex.Cost  $\leftarrow$  Cost
  if Cost > MaxCost then
    let MaxCost  $\leftarrow$  Cost
  end if
end for
let RandCost  $\leftarrow$  random number from 0 to MaxCost
let Select  $\leftarrow$  queue that contains vertices
for each Vertex in NewGraph do
  if Vertex.NoTouch = true then
    next iteration
  end if
  if Vertex.Cost  $\geq$  RandCost then
    insert Vertex into queue Select
  end if
end for
let Vertex  $\leftarrow$  random element from queue Select
return Vertex

```

The `PerturbApply` function is responsible for actually changing the vertex locations. To begin a random target location is generated using the search range stored in the perturb object. This generated target location is then stored in the perturb object. Some simple checks are then performed to see if the source and target locations match, or if the target location has already been tried to avoid wasting time. There are two possible moves, one swaps the source and target vertices, the other relocates the source vertex to the target location. The type of move chosen depends upon whether or not a vertex is already assigned to the target location. If a vertex was found at the target location then a swap is performed. For a swap, if the target vertex is movable then the source vertex is assigned to the target location and the target vertex is assigned to the source location. If a vertex was not found at the target location then a move is performed. For a move, the source vertex is

assigned to the target location. One advantage to performing swaps is that the graph boundaries do not need to be recalculated saving some time. Once the perturb is complete the target location is saved in the perturb object's history to avoid trying the same operation again if the perturb is rejected.

Algorithm 3.14 Placement Phase - PerturbApply

```

PerturbApply(NewGraph) = Success : perturbs the configuration NewGraph by either swapping two vertices or displacing a vertex to a random location

let RandX ← random number from Perturb.MinX to Perturb.MaxX
let RandY ← random number from Perturb.MinY to Perturb.MaxY
let Perturb.DstCoord ← Coordinate (RandX, RandY)
if Perturb.SrcCoord = Perturb.DstCoord then
  return false
end if
if Perturb.DstCoord is in Perturb.History then
  return false
end if
let Perturb.DstVertex ← invalid vertex
for each Vertex in NewGraph do
  if Vertex.Coordinate = Perturb.DstCoord then
    let Perturb.DstVertex ← Vertex
    break loop
  end if
end for
if Perturb.DstVertex is valid then
  if Perturb.DstVertex.NoTouch = true then
    return false
  end if
  let Perturb.SrcVertex.Coordinate ← Perturb.DstCoord
  let Perturb.DstVertex.Coordinate ← Perturb.SrcCoord
else
  let Perturb.SrcVertex.Coordinate ← Perturb.DstCoord
  let NewGraph.Size ← updated bounding box
end if
insert Perturb.DstCoord into queue Perturb.History
return true

```

The **PerturbUndo** function undoes any changes made to the vertex locations. The advantage of having an undo function is that the graph object doesn't have to be duplicated each time a perturb is applied. There are two possible undo operations depending upon whether or not a target vertex was previously found. If a target vertex was previously found then the source vertex is assigned to the source location and the target vertex is assigned to the target location. If a target vertex was not previously found then the source vertex is assigned to the source location and nothing happens to the target vertex, which should be invalid. When the function returns, the configuration is restored to its previously accepted state and is ready for a new perturb variation to be applied.

Algorithm 3.15 Placement Phase - PerturbUndo

```

PerturbUndo(NewGraph) : undoes the last perturb applied to the configuration NewGraph
if Perturb.DstVertex is valid then
  let Perturb.SrcVertex.Coordinate ← Perturb.SrcCoord
  let Perturb.DstVertex.Coordinate ← Perturb.DstCoord
else
  let Perturb.SrcVertex.Coordinate ← Perturb.SrcCoord
  let NewGraph.Size ← updated bounding box
end if

```

Two constants are used when perturbing the configuration. The first constant *PERTURB_TRIAL_MAX* determines the number of variations to try on each new perturb object. If this value is too low the percentage of time spent creating perturb objects increases and the percentage of time spent applying perturb variations decreases. If this value is too high then time will be wasted trying variations of a perturb that may never be accepted. After a little trial and error a value of 5 was chosen, which saves time but doesn't appear to decrease the mapping quality. For a few problems a value of 10 slightly improves the mapping quality but also increases the runtime quite significantly. A value of 10 may be good to use for CPUs much faster than a 2.0 GHz Xeon processor. The second constant *PERTURB_RANGE* determines the search range used for creating new perturb variations. This value seems to work best when in the range of 2 to 4. When this value is set too high tasks spread out further in the beginning of the temperature schedule and don't seem to compress as well as lower values. A value of 3 was chosen, which seems to work very well.

3.2.3 Modifications

A few modifications were made to the classic simulated annealing algorithm. Some of these modification were borrowed from StreamIt, a portable framework for programming stream-based architectures [14]. The first borrowed modification was executing the simulated annealing algorithm multiple times, each time starting with the best configuration from the previous iteration. The quality of the mapping sometimes improved substantially with just a second pass. The second borrowed modification was sorting nodes using depth first search before initially placing them. This helped when mapping applications that had long chains of nodes, similar to software pipelines. The third borrowed modification was determining the initial and final temperature points through a series of perturbations. This saved a noticeable amount of time when mapping simpler applications. The complexity of an application was accurately reflected upon by the runtime of the mapping algorithm due to these correctly defined temperature points. The first modification, not found in other imple-

mentations, was flagging locations that could later be used to simplify the routing. The probability of successfully completing a route between two distant nodes was greatly increased by leaving these flagged locations unoccupied. The second modification, not found in other implementations, was saving the state of a perturbation so it could be reversed and other variations could be applied. This modification increased the time spent trying a perturb and reduced the time spent preparing a perturb. Each of these modifications attempt to either reduce the runtime, increase the mapping quality, or do both.

3.2.4 Summary

Simulated annealing has been a good fit for the placement phase of the mapping algorithm due to its flexibility and excellent performance as a heuristic. For the mapping problem discussed in this work, only a few modifications had to be made to the simulated annealing framework. However, components such as the configuration cost functions and the perturb graph functions had to be highly customized. A number of customizable parameters and finely tuning constants were introduced, which were determined through trial and error. There's no doubt that more work could be done to further tune parameters and implement additional optimizations but this is left for future work. Overall the current implementation has been very successful in placing tasks from a wide variety of applications onto the AsAP architecture.

3.3 Routing Phase

The routing phase is fairly straight forward. Like the placement phase, much of the basic structure used in the implementation of the routing phase came from the book *Algorithms for VLSI Design Automation* by Gerez [13]. For this phase the primary goal is to insert routing processors into the graph in order to connect non-nearest neighbor processors. An important requirement for the routing phase algorithm is that it find the shortest path available in a relatively short period of time. Maze routing was chosen as the base algorithm for this phase because the algorithm is easy to implement and guaranteed to find the shortest path if one exists. The maze routing algorithm also has the flexibility required to handle intersecting routing processors and excluded processors.

Maze routing is just one of a few different routing algorithms to choose from. Some routing algorithms target specific applications, such as routing wire segments in FPGAs [22]. Even the maze routing algorithm itself (also called the Lee path connection algorithm [21]) has many derivations for targeting specific applications. It appears that most routing algorithms are derived from maze

routing or are somehow related. Some derivations include multi-layer routing and routing to multiple end-points for a single net simultaneously. Maze routing has been used successfully for years for PC-board design. One appealing aspect of maze routing, which contributes to its ease of implementation, is that it operates on a 2D-grid. This makes the algorithm a perfect fit for the ASAP architecture. Excluded processors are also trivial to implement since the algorithm was originally designed to route around obstacles.

3.3.1 Maze Routing

The basic principle behind the maze routing algorithm is, first choose a source and target location, next search for the shortest path, next assign tasks to grid locations while tracing back along the shortest path, finally clean-up any unused grid locations. The search for the shortest path is performed using propagation waves, which bubble around obstacles, while marking grid locations with their distance from the source. Propagation continues until the wavefront encounters the target which is then marked with the length of the shortest path back to the source (plus one actually). The algorithm is guaranteed to find the shortest path if one exists because the wave propagates in every direction that's not obstructed one hop at a time. When the target is reached a path is traced back to the source decreasing the distance one step at a time until one is reached. Grid locations used by the shortest path are blocked off and the remaining grid locations are cleaned-up so they can be used for the next route.

3.3.2 Algorithm Details

The routing phase has two major components. The first component, which is optional, adds empty space to the array to help with routing. This component is primarily used to decrease the likelihood of routing conflicts for applications that are very congested. The second component is the maze routing algorithm, which iterates over every long distance connection in the array and inserts a chain of routing processors (if possible) in order to convert routes to nearest neighbor only. The remainder of this section will discuss how these two components work in detail along with modifications made to the maze routing algorithm to handle intersecting routing processors.

Framework

The `RoutingMain` function, listed in Algorithm 3.16, contains the basic maze routing framework with the additional space insertion component. This function serves as the starting point for the

routing phase. This function doesn't do any work itself and instead relies on the other subfunctions to do all the heavy lifting. The main purpose of this function is to orchestrate the process of adding space to the array, setting up the gridmap, and passing each edge in sequence to the maze router. No optimizations are really performed by this function with the exception of enabling/disabling space insertion.

Before routing can actually begin the function must first check whether or not additional space is needed and has been requested. It's important to note that the array size can change if space is inserted into the array. Therefore space has to be inserted before creating the gridmap object. Empty space is only added to the array when the space insertion component is enabled and the edge to node ratio is large enough. When additional space is requested the `InsertSpacing` function is called to selectively add empty columns and rows into the array in an intelligent fashion. Once space insertion has been completed the `InitGridmap` function can be called to create and initialize a new gridmap object. The new gridmap object will be identical in size to the target array which is why it was important to make any changes to the target array's size before initializing the gridmap. The last preparation step is to put all non-nearest neighbor edges into a list. Now to route the edges the source and target coordinates for each edge in the list are passed through three routing stages. The first stage is wave propagation. In this stage the `Propagate` function is called to search for the shortest path between the source and the target. The second stage is path traceback. This stage only occurs if there were no conflicts during the wave propagation stage. In this stage the `Traceback` function is called to trace back along the shortest path and insert routing processors. The final stage is gridmap cleanup. In this stage the `Cleanup` function is called to undo all the changes made to the gridmap object during the wave propagation stage, excluding the changes made during the path traceback stage. This basic procedure is shown visually in Figure 3.6. Once the edge list has been exhausted the function returns.

This function uses two configuration parameters. The `Config.AddSpacing` parameter determines whether or not additional space is added to the array. With additional space the number of possible routes between the source and the target increases, thereby reducing routing conflicts. Space insertion is typically used for applications where the edge to vertex ratio is very high. Applications with a high ratio are difficult to map because multiple connections will try to route through the same processor creating conflicts. The second parameter, `Config.SpaceThreshold`, determines how high this ratio must be before space is inserted. If this threshold is too low then routing processors will be inserted when they are not needed. If this value is too high then many of the long-distance

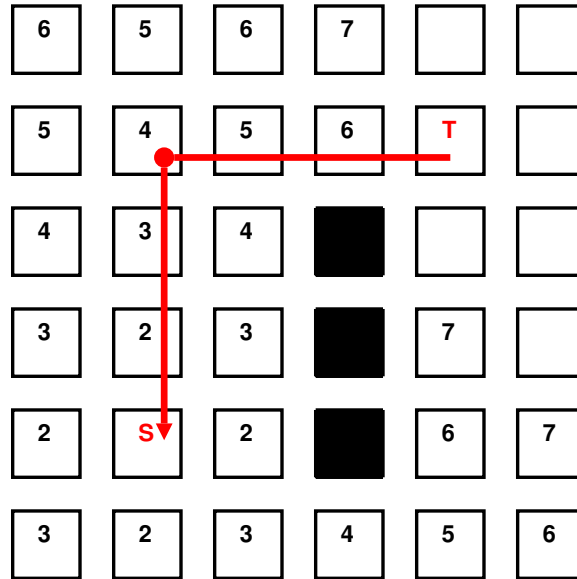


Figure 3.6: The basics of the maze routing algorithm shown visually. The labels S and T represent the source and target vertices, respectively. The numbers represent the distance to the source vertex, generated by the propagation stage. The red arrow is the shortest path from the target back to the source. The black boxes are unroutable locations (which can be due to an excluded location, a compute processor, or a fully utilized router) and the unlabeled boxes were unused during the propagation stage.

Algorithm 3.16 Routing Phase - RoutingMain

```

RoutingMain() : inserts routing processors to connect non-nearest neighbor vertices

if Config.AddSpacing then
  let NumEdges  $\leftarrow$  number of edges in Graph
  let NumNodes  $\leftarrow$  number of nodes in Graph
  let Threshold  $\leftarrow$  (NumEdges  $\times$  100) / NumNodes
  if Threshold > Config.SpaceThreshold then
    InsertSpacing()
  end if
end if
InitGridmap()
let Routes  $\leftarrow$  queue that contains edges
for each Edge in Graph do
  if Edge.Source and Edge.Target are not nearest neighbors then
    insert Edge into queue Routes
  end if
end for
for each Edge in Routes do
  let Source  $\leftarrow$  Edge.Source.Coordinate
  let Target  $\leftarrow$  Edge.Target.Coordinate
  if Propagate(Source, Target) = true then
    Traceback(Source, Target, Edge)
  end if
  Cleanup()
end for

```

connections will remain unrouted for complex applications. This parameter is expressed as an integer percentage instead of as a ratio. A value of 110 seems to work after some experimentation but there is still some uncertainty in what the best value is for this parameter. For some of the applications that were tested space insertion must be disabled in order to obtain a good mapping because the calculated threshold does not accurately reflect the applications routing complexity. There are likely better ways to decide whether or not space should be inserted.

Space Insertion

Space insertion is a rather complex process so it has been broken down into numerous functions. These functions are `InsertSpacing`, `EdgeDepends`, `ColSplits`, `RowSplits`, and `ShiftArray`, listed in Algorithm 3.17, Algorithm 3.18, Algorithm 3.19, Algorithm 3.20, and Algorithm 3.21 respectively. The goal of space insertion is to reduce routing conflicts. This is done by selectively inserting empty columns and rows into the array to increase the number of unassigned processors. When the number of unassigned processors is increased the number of possible routes between the source and the target is also increased. This is because these new rows and columns of entirely unassigned processors can be used to circle around any obstacles that were previously blocking a routable path. Intelligently choosing where to insert empty rows and columns makes space insertion complex since poor choices bloat the array. Space insertion is mostly a hack that allows complex applications (like the large Clos network on page 88) to be mapped successfully using nearest neighbor communication only. For these complex applications space insertion substantially degrades the mapping quality. In some cases the mapping quality is never even partially restored.

The `InsertSpacing` function is responsible for shifting parts of the array in order to create additional routing space. Before the array can be shifted the subfunctions `EdgeDepends`, `ColSplits`, and `RowSplits` must be called to fill the column position and row position queues. These two queues contain the column positions and the row positions to use for shifting the array. When the array is shifted horizontally everything to the right of the column being shifted is moved right one column. When the array is shifted vertically everything below the row being shifted is moved down one row. The one exception to these rules is when the shift amount is negative where everything to the right or below the shift position is moved toward the left or the top of the array. Every time the array is shifted the column and row numbers for every task below or to the right of the shift position increase by one. This plays an important role in what order the shifts are applied. To avoid invalidating pending positions in the two position queues they are first sorted in reverse numerical order. The

array is shifted once horizontally for each position in the column position queue. If a failure occurs, which can happen if a task is unmovable or the destination for a move is blocked, then all pending horizontal shifts in the queue are aborted since they will also fail. Likewise the array is shifted once vertically for each position in the row position queue. If a failure occurs then all pending vertical shifts in the queue must again be aborted since they will also fail. If every shift is successful then the X dimension will have increased by the number of items in the column position queue and the Y dimension will have increased by the number of items row position queue.

Algorithm 3.17 Routing Phase - InsertSpacing

```

InsertSpacing() : inserts additional rows and columns of empty space to aid with routing
let DependCol  $\leftarrow$  array of size Graph.Size.X containing queues that contain edges
let DependRow  $\leftarrow$  array of size Graph.Size.Y containing queues that contain edges
let SplitCol  $\leftarrow$  queue that contains column numbers
let SplitRow  $\leftarrow$  queue that contains row numbers
EdgeDepends(DependCol, DependRow)
ColSplits(DependCol, SplitCol)
RowSplits(DependRow, SplitRow)
sort queue SplitCol in descending order
for each Col in SplitCol do
  if ShiftArray(HORIZONTAL, Col, 1) = false then
    break loop
  end if
end for
sort queue SplitRow in descending order
for each Row in SplitRow do
  if ShiftArray(VERTICAL, Row, 1) = false then
    break loop
  end if
end for

```

The `EdgeDepends` function is responsible for calculating edge dependencies. For each edge there are four possible dependencies, two column dependencies, and two row dependencies. The two possible column dependencies are the column adjacent to the leftmost vertex, between the source and the target, and the column containing the rightmost vertex. The number of columns that an edge depends upon is based on the difference in the X dimension between the source coordinate and the target coordinate. If the difference is equal to zero then the edge has no column dependencies. If the difference is equal to one then both column dependencies refer to the same column, so the edge really only has one column dependency. If the difference is more than one then the edge has two column dependencies. The edge is inserted into a queue for each column that is a valid column dependency. The two possible row dependencies are the row adjacent to the topmost vertex, between the source and the target, and the row containing the bottommost vertex. The number of rows that

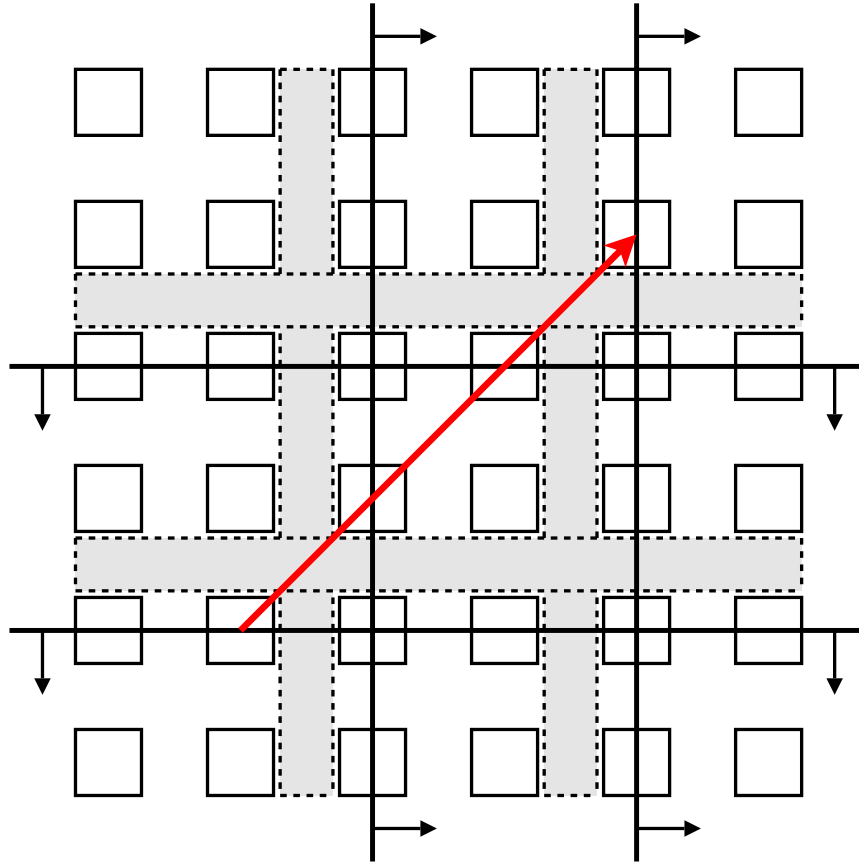


Figure 3.7: Depiction of the two possible column dependencies and the two possible row dependencies. In this example the black bars represent array shift locations and the gray regions represent newly created routing space after shifting the array.

an edge depends upon is based on the difference in the Y dimension between the source coordinate and target coordinate. If the difference is equal to zero then the edge has no row dependencies. If the difference is equal to one then both row dependencies refer to the same row, so the edge has only one row dependency. If the difference is more than one then the edge has two row dependencies. The edge is inserted into a queue for each row that is a valid row dependency. The two possible column dependencies and the two possible row dependencies are shown in Figure 3.7. The function returns once the column and row dependencies have been calculated for every edge in the graph.

Calculating split (or shift) positions from edge dependencies is divided into two functions. The `ColSplits` function is responsible for calculating which columns to split based on which columns have the most edge dependencies. The `RowSplits` function is responsible for calculating which rows to split based on which rows have the most edge dependencies. Both of these functions are identical except that the first function operates on column positions and the second function operates on row positions. The first step is to find which positions have the most edge dependencies and put these

Algorithm 3.18 Routing Phase - EdgeDepends

EdgeDepends(*DependCol*, *DependRow*) : calculates which columns, stored in *DependCol*, and which rows, stored in *DependRow*, each edge depends upon for splitting

```

for each Edge in Graph do
  if Edge.Source and Edge.Target are nearest neighbors then
    next iteration
  end if
  let MinX  $\leftarrow$  min(Edge.Source.X, Edge.Target.X)
  let MaxX  $\leftarrow$  max(Edge.Source.X, Edge.Target.X)
  let MinY  $\leftarrow$  min(Edge.Source.Y, Edge.Target.Y)
  let MaxY  $\leftarrow$  max(Edge.Source.Y, Edge.Target.Y)
  if difference between Edge.Source.X and Edge.Target.X  $\geq$  1 then
    insert Edge into queue DependCol[MinX + 1]
  end if
  if difference between Edge.Source.X and Edge.Target.X  $\geq$  2 then
    insert Edge into queue DependCol[MaxX]
  end if
  if difference between Edge.Source.Y and Edge.Target.Y  $\geq$  1 then
    insert Edge into queue DependRow[MinY + 1]
  end if
  if difference between Edge.Source.Y and Edge.Target.Y  $\geq$  2 then
    insert Edge into queue DependRow[MaxY]
  end if
end for

```

positions into a selection list. This is done by counting the number of edge dependencies at each position and comparing this number to a running maximum count. The maximum count is the highest number of edge dependencies seen so far, which is initially set to one so positions with no edge dependencies are ignored. If the number of edge dependencies is greater than the maximum count then the maximum count is updated and a new selection list is created containing only the corresponding position. If the number of edge dependencies is equal to the maximum count then the corresponding position is appended to the selection list. If the selection list is empty after every position has been counted then the function returns since there are no more edge dependencies to satisfy. If the selection list is not empty then a random position is chosen from the selection list and put into the split list. The split list is a queue that contains the positions that will actually be shifted. The final step is to remove any dependencies related to the position that was selected. A list is first created that contains every edge that depends upon the position that was selected. If another position has an edge from the list in its queue then the edge is removed from the queue. The number of edge dependencies in each position queue will decrease as more positions are put into the split list. When there are no more edge dependencies left the function returns.

The `ShiftArray` function is responsible for mass reassigning tasks to new processors in the

Algorithm 3.19 Routing Phase - ColSplits

ColSplits(*DependCol*, *SplitCol*) : chooses the columns to split, stored in *SplitCol*, based on which columns have the most dependencies in *DependCol*

```

loop
  let  $Size_{max} \leftarrow 1$ 
  let Select  $\leftarrow$  queue that contains column numbers
  for  $X \leftarrow 1$  to Graph.Size.X - 1 do
    let Size  $\leftarrow$  length of queue DependCol[X]
    if  $Size > Size_{max}$  then
      let  $Size_{max} \leftarrow Size$ 
      clear queue Select
    end if
    if  $Size = Size_{max}$  then
      insert X into queue Select
    end if
  end for
  if length of queue Select = 0 then
    break loop
  end if
  let Col  $\leftarrow$  random element from queue Select
  insert Col into queue SplitCol
  for each Edge in queue DependCol[Col] do
    for each Queue in array DependCol do
      if Edge is in Queue then
        remove Edge from Queue
      end if
    end for
  end for
end loop

```

array. Shifting the array is also referred to as splitting the array. Every shift operation depends upon two coordinates. The first coordinate is the lower bounds for selecting tasks. Every task below and to the right of the lower bounds coordinate will be included in the shift. The second coordinate is the displacement. Each task will be shifted by the amount set in the displacement coordinate. There are two types of shifts, horizontal shifts and vertical shifts. The only difference between the two types is how the two previous coordinates are initialized. For horizontal shifts, the X dimension for the lower bounds coordinate is set to the starting position and the Y dimension for the lower bounds coordinate is set to zero. Also the X dimension for the displacement coordinate is set to the shift amount and the Y dimension for the displacement coordinate is set to zero. For vertical shifts the X and Y dimensions are reversed. Before shifting the array every task must be checked to ensure that no constraints are violated. The first check that is done on each task is to ensure that the task is within the lower bounds. The second check that is done on each task is to ensure that the task is movable. If this check fails then the shift operation is impossible so the function returns the value

Algorithm 3.20 Routing Phase - RowSplits

RowSplits(*DependRow*, *SplitRow*) : chooses the rows to split, stored in *SplitRow*, based on which rows have the most dependencies in *DependRow*

```

loop
  let  $Size_{max} \leftarrow 1$ 
  let Select  $\leftarrow$  queue that contains row numbers
  for  $Y \leftarrow 1$  to Graph.Size. $Y - 1$  do
    let Size  $\leftarrow$  length of queue DependRow[ $Y$ ]
    if  $Size > Size_{max}$  then
      let  $Size_{max} \leftarrow Size$ 
      clear queue Select
    end if
    if  $Size = Size_{max}$  then
      insert  $Y$  into queue Select
    end if
  end for
  if length of queue Select = 0 then
    break loop
  end if
  let Row  $\leftarrow$  random element from queue Select
  insert Row into queue SplitRow
  for each Edge in queue DependRow[Row] do
    for each Queue in array DependRow do
      if Edge is in Queue then
        remove Edge from Queue
      end if
    end for
  end for
end loop

```

false. The final check that is done on each task is to ensure that the task will not be assigned to an excluded processor. Like the previous check, if this check fails then the shift operation is impossible so the function returns the value false. If a task passes all three checks then the task is inserted into the shift queue. Once every task has been checked the array can be shifted. Shifting the array is as simple as adding the displacement coordinate to each task in the shift queue. This effectively creates an empty column or row in the array. The final step is to update the size of the array to account for the new column or row. The function then returns the value true to indicate that the shift was successful.

Only one configuration parameter is used by these functions. The *Config.ExcludeList* parameter is a list of locations within the target array that should not have tasks assigned to them. This list is used when shifting the array to ensure that none of the tasks are moved onto an excluded processor. The shift operation will fail if even a single task is predicated to land on an excluded processor.

Algorithm 3.21 Routing Phase - ShiftArray

```

ShiftArray(Type, Start, Amount) = Shifted : shifts all nodes between Start and the array
edge by Amount either vertically or horizontally based on Type, returns true if every move was
valid

if Type = HORIZONTAL then
  let Bounds ← Coordinate (Start, 0)
  let Shift ← Coordinate (Amount, 0)
else
  let Bounds ← Coordinate (0, Start)
  let Shift ← Coordinate (0, Amount)
end if
let Select ← queue that contains vertices
for each Vertex in Graph do
  if Vertex.X < Bounds.X or Vertex.Y < Bounds.Y then
    next iteration
  end if
  if Vertex.NoTouch = true then
    return false
  end if
  let Coord ← Vertex.Coordinate + Shift
  if Coord is in Config.ExcludeList then
    return false
  end if
  insert Vertex into queue Select
end for
for each Vertex in Select do
  let Vertex.Coordinate ← Vertex.Coordinate + Shift
end for
let Graph.Size ← Graph.Size + Shift
return true

```

The constants *VERTICAL* and *HORIZONTAL* are simply enumerations used to distinguish the type of shift operation being performed.

Initialize Gridmap

The gridmap is initialized by the `InitGridmap` function listed in Algorithm 3.22. To initialize the gridmap a new gridmap object is created and each grid point in the gridmap is initialized to match its corresponding location in the target array. Every gridmap object contains three 2D-arrays each the size of the target array. These three 2D-arrays contain the information necessary to distinguish one grid point from another when deciding which grid points are available for routing. Initializing the gridmap fundamentally consists of assigning values to these three 2D-arrays so that the gridmap and target array are synchronized.

The routing gridmap is stored in the global variable *Gridmap*, which contains all the infor-

Property	Description
<i>Gridmap.Size</i>	The dimensions for the <i>Gridmap</i>
<i>Gridmap.Value</i>	2D-array containing route marker values
<i>Gridmap.Routes</i>	2D-array containing remaining intersections
<i>Gridmap.Vertex</i>	2D-array containing associated vertices

Table 3.5: List of properties for the *Gridmap* variable.

mation necessary to ensure that routers are only assigned to valid locations. The properties for this global variable are listed in Table 3.5. The types of values stored in each field and how each field is used is described in further detail in the remainder of this subsection. The *Gridmap* variable has a limited lifetime and after being initialized only exists until the end of the routing phase.

The types of values stored in the three 2D-arrays play an important role in determining which grid points are available for routing. There are three different types of values stored in the *Value* field for a grid point. The three types are negative values, positive values, and the value zero. A negative value (which most of the time is -1) is used to indicate that the grid point is occupied by a task. A positive value is used to indicate that the grid point has been considered as a possible candidate for the route currently being solved. A value of zero is used to indicate that the grid point is available and unoccupied. There are two different types of values stored in the *Routes* field for a grid point. The two types are positive values and the value zero. A positive value is used to indicate that the grid point still has ports available for routing data. This also indicates that the grid point is available. A value of zero is used to indicate that the grid point has no more ports available for routing data. The number of routes is decremented each time a new route intersects the grid point. The *Vertex* field for a grid point can either be a valid vertex (containing its associated task) or an invalid vertex. Two cases where the vertex will be valid are, when the grid point is associated with a task from the original graph or when a routing task has been inserted at that location.

The `InitGridmap` function is responsible for initializing the grid points within the gridmap. The function starts by creating a new gridmap object and setting the size of the gridmap object equal to the size of the target array. In the beginning every grid point in the gridmap is initialized as available. To appear available the value of the grid point is set to 0, the number of routes for the grid point is set to the maximum (defined by a configuration parameter), and the vertex for the grid point is set to an invalid vertex. Next the grid points in the gridmap are synchronized to the vertices in the graph. For each grid point with a corresponding vertex in the graph, the value of the grid point is set to -1 , the number of routes for the grid point is set to 0, and the vertex for the grid point is set to the corresponding vertex in the graph. These grid points will appear as unavailable.

Algorithm 3.22 Routing Phase - InitGridmap

```

InitGridmap() : creates and initializes a new Gridmap object for tracking routing progress
create new global object Gridmap
let Gridmap.Size  $\leftarrow$  Graph.Size
let Gridmap.Value  $\leftarrow$  2D-array of size Graph.Size.X  $\times$  Graph.Size.Y containing numbers
let Gridmap.Routes  $\leftarrow$  2D-array of size Graph.Size.X  $\times$  Graph.Size.Y containing numbers
let Gridmap.Vertex  $\leftarrow$  2D-array of size Graph.Size.X  $\times$  Graph.Size.Y containing vertices
for each index Coord of 2D-arrays do
  let Gridmap.Value[Coord]  $\leftarrow$  0
  let Gridmap.Routes[Coord]  $\leftarrow$  Config.MaxRoutes
  let Gridmap.Vertex[Coord]  $\leftarrow$  invalid vertex
end for
for each Vertex in Graph do
  let Coord  $\leftarrow$  Vertex.Coordinate
  let Gridmap.Value[Coord]  $\leftarrow$  -1
  let Gridmap.Routes[Coord]  $\leftarrow$  0
  let Gridmap.Vertex[Coord]  $\leftarrow$  Vertex
end for
for each Coord in Config.ExcludeList do
  let Gridmap.Value[Coord]  $\leftarrow$  -1
  let Gridmap.Routes[Coord]  $\leftarrow$  0
  let Gridmap.Vertex[Coord]  $\leftarrow$  invalid vertex
end for

```

To finish initializing the gridmap each location that is excluded in the target array must appear as unavailable. For each grid point that matches an excluded location, the value of the grid point is set to -1 , the number of routes for the grid point is set to 0, and the vertex for the grid point is set to invalid. When the function returns the gridmap and graph will be synchronized.

Two configuration parameters are used when initializing the gridmap. The *Config.MaxRoutes* parameter determines the maximum number of routes that can pass-through a single grid point. This parameter can be any number between 1 and 4, with 4 providing the most routing flexibility. The problem is that each processor in the ASAP architecture has only 2 input ports so the effective range is actually a number between 1 and 2. The *Config.ExcludeList* parameter is a list of locations within the target array that should never be used for routing. This list is used to setup permanent obstacles in the gridmap and ensure that the state of any corresponding grid points remains consistent with the target array.

Wave Propagate

The wave propagate stage is broken down into two functions. These functions are *Propagate* and *QueueNeighbors*, listed in Algorithm 3.23 and Algorithm 3.24 respectively. The goal of wave propagation is to find the shortest path from the source to the target. The shortest

path is found using an expanding wavefront, which marks each available grid point with the distance the grid point is from the source. This method is called single wave propagation. Another method is to use two wavefronts, one starting from the source, another starting from the target [29]. Both wavefronts propagate simultaneously until the two wavefronts collide. Double wave propagation is quicker than single wave propagation but single wave propagation was used because it is easier to implement and the runtime was already extremely short.

The `Propagate` function is responsible for marking grid points between the source and the target using an increasing numerical sequence. In order for the target location to be selectable the value of the grid point for the target location must be set to 0. The initial wavefront is created by inserting the source location into the wavefront queue. Since the source location is the start of the route, and wavefronts are numbered sequentially starting from 1, the value of the grid point for the source location is set to 1. The reason the sequence starts at 1 instead of 0 is to distinguish the source grid point from unoccupied grid points, otherwise the route would double back. Once the wavefront begins propagating it continues to propagate until either the target location is selected by the wavefront or the wavefront is unable to expand any further due to obstacles. To expand the wavefront a new wavefront queue is created, which is filled with the locations for next wavefront. To determine the locations for the next wavefront each location in the current wavefront queue is passed in turn to the `QueueNeighbors` subfunction. The subfunction selects all the available neighbors for a given location and puts them into the new wavefront queue. The current wavefront queue is then replaced by the new wavefront queue and the wave expands again. If a path from the source to the target was not found then the value of the grid point for the target location is reset back to -1 (the clean-up stage will not reset grid points with a value of 0) and the function returns the value false. By returning the value false back the parent function, `RoutingMain`, the parent function will know not to attempt a traceback since it will fail. If a path from the source to the target was found then the value of the grid point for the target location will contain the length of the shortest path plus one since numbering starts from 1.

The `QueueNeighbors` function is responsible for checking which neighbors are available for a given grid point and inserting them into the wavefront queue. The four neighbor locations checked are the north, south, east, and west locations relative to the given location. The first check that is done on each location is to ensure that the location is within the boundaries of the gridmap in case the edge of the array is reached. The next check that is done on each location is to verify that the location has not already been selected by a previous wavefront. Without this check the wavefront

Algorithm 3.23 Routing Phase - Propagate

```

Propagate(Source, Target) = Routable : finds the shortest path from Source to Target by
propagating in waves and bubbling around obstacles, returns true if some path was found

let Gridmap.Value[Target] ← 0
let Gridmap.Value[Source] ← 1
let Wave ← queue that contains coordinates
insert Source into queue Wave
while Target is not in Wave and Wave is not empty do
  let NewWave ← queue that contains coordinates
  for each Coord in Wave do
    QueueNeighbors(Coord, NewWave)
  end for
  let Wave ← NewWave
end while
if Wave is empty then
  let Gridmap.Value[Target] ← -1
  return false
end if
return true

```

would propagate back to the source. The next check that is done on each location is whether or not the location is occupied and if so can the location accept another route. The final check that is done on each location only applies to locations that are occupied by routing tasks. For these locations we need to perform an additional check to determine whether or not the ports between the original location and the neighbor location are already allocated to another route. If the ports have already been allocated then we will need to enter this location from another direction. If the value of the grid point for the location is zero it's assumed to be available. This is why the value of the grid point for the target location was changed to zero, otherwise it would have been skipped. If the location passes all these checks then the value of the grid point is changed to the next number in the sequence. The location is also inserted into the new wavefront queue. The function then continues on to the next neighbor until every neighbor has been checked.

Path Traceback

The path traceback stage is broken down into three functions. These functions are *Traceback*, *NextNeighbor*, and *InsertRouter* listed in Algorithm 3.25, Algorithm 3.26, and Algorithm 3.27 respectively. The goal for the path traceback stage is to assign new routing tasks to grid points along the shortest path. Tracing the shortest path from the target back to the source is as simple as following the decreasing numerical sequence. This numerical sequence starts at the value of the grid point for the target location and ends at 1. The path traceback stage contains

Algorithm 3.24 Routing Phase - QueueNeighbors

QueueNeighbors(*Coord*, *NewWave*) : checks the coordinates neighboring *Coord* and inserts available coordinates into the queue *NewWave*

```

let Value ← Gridmap.Value[Coord] + 1
for each NewCoord that is a neighbor of Coord do
  if NewCoord is outside of Gridmap then
    next iteration
  end if
  if Gridmap.Value[NewCoord] > 0 then
    next iteration
  end if
  if Gridmap.Value[NewCoord] < 0 then
    if Gridmap.Routes[NewCoord] < 1 then
      next iteration
    end if
  end if
  if edge exists in Graph from Coord to NewCoord then
    next iteration
  end if
  let Gridmap.Value[NewCoord] ← Value
  insert NewCoord into queue NewWave
end for

```

one modification to the maze routing algorithm that greatly enhances the mapping quality for complex applications. This modification is the ability to re-use certain grid points that were previously marked as unavailable. This is needed so that routing tasks with more than one route passing through them can be inserted.

The **Traceback** function is responsible for tracing the shortest path from the target back to the source and inserting or modifying routing processors. When the function first starts the active grid point is set to the target location. Since the target location is already part of the route the active grid point immediately becomes the target's predecessor and the value of the grid point for the target location is reset back to its original value of -1 . The function then starts the traceback loop, which will continue traversing the shortest path until the source location is found. In order to find the predecessor for a grid point, the value of the grid point must be one greater than its predecessor. This is important to remember because once the grid point becomes part of the route the value of the grid point is changed to reflect its new state. For this reason the first step of each iteration is to save the next grid point in the path. Once the next grid point in the path has been saved, the value of the active grid point is changed to -1 to indicate that the grid point is now part of the route. Next the vertex for the active grid point and the edge being routed are both passed to the **InsertRouter** function. If the vertex was invalid, which indicates that the grid point was unoccupied, then a new

router vertex will be created. If the vertex was not invalid, which indicates that a router vertex already exists, then the vertex is modified to allow another route to pass-through it. To make the vertex part of the actual route the edge is split into two parts and the vertex is inserted at the midpoint. The number of routes for the active grid point is then decremented to reflect the changes just made to the vertex. Newly created router tasks must be assigned to a processor location in the target array. The coordinate assigned to the new task is the location of the active grid point. In order to keep the gridmap in sync with the graph the vertex field for the active grid point must be replaced with the new or modified router vertex. The final step is to replace the active grid point with the next grid point in the path so that the next iteration can begin. After the traceback loop finishes the value of the grid point for the source location is reset back to its original value of -1 to complete the route.

Algorithm 3.25 Routing Phase - Traceback

Traceback(*Source*, *Target*, *Edge*) : traces the shortest path from *Target* back to *Source* inserting new routing vertices along *Edge*

```

let Neighbor  $\leftarrow$  NextNeighbor(Target)
let Gridmap.Value[Target]  $\leftarrow$   $-1$ 
while Neighbor is not Source do
  let Next  $\leftarrow$  NextNeighbor(Neighbor)
  let Gridmap.Value[Neighbor]  $\leftarrow$   $-1$ 
  let Vertex  $\leftarrow$  Gridmap.Vertex[Neighbor]
  InsertRouter(Edge, Vertex)
  decrement Gridmap.Routes[Neighbor]
  let Vertex.Coordinate  $\leftarrow$  Neighbor
  let Gridmap.Vertex[Neighbor]  $\leftarrow$  Vertex
  let Neighbor  $\leftarrow$  Next
end while
let Gridmap.Value[Source]  $\leftarrow$   $-1$ 

```

The `NextNeighbor` function is responsible for figuring out which neighbor precedes a given grid point. The four neighbor locations searched are the north, south, east, and west locations relative to the given location. The first check that is done on each location is to ensure that the location is within the boundaries of the gridmap in case the edge of the array is reached. The next check that is done on each location, which is the most important check, is to see whether or not the neighboring grid point actually precedes the original grid point. In order to pass this check the value of the grid point for the neighbor location must be one less than the value of the grid point for the original location. The final check that is done on each location only applies to locations that are occupied by routing tasks. For these locations we need to perform an additional check to determine whether or not the ports between the neighbor location and the original location have already been

allocated to another route. If the ports were already allocated, which is a very rare occurrence, then the path was diverted during the wave propagation stage. If the location passes all these checks then the predecessor has been found so the function returns. If the location did not pass all these checks then the function starts checking the next neighboring location.

Algorithm 3.26 Routing Phase - NextNeighbor

```

NextNeighbor(Coord) = NextCoord : finds the neighboring coordinate NextCoord that pre-
cedes Coord in the route

let Value ← Gridmap.Value[Coord] - 1
for each NewCoord that is a neighbor of Coord do
  if NewCoord is outside of Gridmap then
    next iteration
  end if
  if Gridmap.Value[NewCoord] ≠ Value then
    next iteration
  end if
  if edge exists in Graph from NewCoord to Coord then
    next iteration
  end if
  return NewCoord
end for

```

The `InsertRouter` function is responsible for creating new routing vertices and modifying existing routing vertices to handle one more route. This function is also responsible for placing the vertex along the edge being routed. Placing the vertex along the edge involves removing the original edge and inserting two new edges with the routing vertex at the midpoint. The source and target vertices for the original edge must be saved before the edge can be removed. This is required so that the two new edges can be connected properly. Though before the two new edges can be created, the routing vertex has to be created or modified. There are five categories used for classifying a vertex. The `PROCESSOR` category is used for computing tasks, which are always tasks from the original application. The `ROUTER_1WAY`, `ROUTER_2WAY`, `ROUTER_3WAY`, and `ROUTER_4WAY` categories are used for routing tasks that can handle one, two, three, or four simultaneous routes, respectively. If the vertex was previously invalid then a new router vertex is created with the category `ROUTER_1WAY`. If the vertex was not previously invalid then the category for the vertex is upgraded. The possible upgrades are: `ROUTER_1WAY` becomes `ROUTER_2WAY`, `ROUTER_2WAY` becomes `ROUTER_3WAY`, and `ROUTER_3WAY` becomes `ROUTER_4WAY`. The vertex is now ready so the new edges can be created. The first new edge connects from source vertex to the router vertex. The second new edge connects from the router vertex to the target vertex. Since the original edge was removed one of the two new edges must take its place for future splitting. The

first edge is the most logical choice since the next router vertex will be inserted between the current router vertex and the source vertex. The source side edge is shortened each time this function is called until finally the source side edge has a length of one.

Algorithm 3.27 Routing Phase - InsertRouter

InsertRouter(*Edge*, *Vertex*) : adds a midpoint to *Edge* that passes through *Vertex* and promotes *Vertex* to the next category

```

let Source ← Edge.Source
let Target ← Edge.Target
remove Edge from Graph
if Vertex is invalid then
  let Vertex ← new vertex added to Graph
  let Vertex.Category ← ROUTER_1WAY
else if Vertex.Category = ROUTER_1WAY then
  let Vertex.Category ← ROUTER_2WAY
else if Vertex.Category = ROUTER_2WAY then
  let Vertex.Category ← ROUTER_3WAY
else if Vertex.Category = ROUTER_3WAY then
  let Vertex.Category ← ROUTER_4WAY
end if
let SrcEdge ← edge added to Graph from Source to Vertex
let DstEdge ← edge added to Graph from Vertex to Target
let Edge ← SrcEdge

```

The constants *ROUTER_1WAY*, *ROUTER_2WAY*, *ROUTER_3WAY*, and *ROUTER_4WAY* are simply enumerations used to distinguish the type of an object.

Gridmap Cleanup

The gridmap cleanup stage is performed by the **Cleanup** function listed in Algorithm 3.28. The goal of the gridmap cleanup function is to reset the values of the grid points used during the wave propagation stage back to their original values. Grid points that became part of a route during the path traceback stage are ignored since their values have already been updated to reflect their new state.

The **Cleanup** function is responsible for resetting the values for any grid points used during wave propagation. Every grid point in the gridmap is analyzed in order to determine which grid points were used during wave propagation. If the value of the grid point is greater than zero then the grid point was used during wave propagation and needs to be reset. There are two types of grid points used during wave propagation, occupied grid points and unoccupied grid points. Each type of grid point has a different reset value. If the vertex for the grid point is invalid then the grid point is unoccupied, otherwise the grid point is occupied. For unoccupied grid points the value of

the grid point is reset to 0. For occupied grid points the value of the grid point is instead reset to -1 . A value of -1 indicates to the wave propagation stage that the grid point is occupied. When the function returns, every possible routable grid point will again be available.

Algorithm 3.28 Routing Phase - Cleanup

```

Cleanup() : resets gridmap values for locations that were not chosen during the traceback
for each index Coord of 2D-arrays do
  if Gridmap.Value[Coord] > 0 then
    if Gridmap.Vertex[Coord] is valid then
      let Gridmap.Value[Coord]  $\leftarrow -1$ 
    else
      let Gridmap.Value[Coord]  $\leftarrow 0$ 
    end if
  end if
end for

```

3.3.3 Modifications

Very few modifications were made to the classic maze routing algorithm. The space insertion component is not technically a modification to the maze routing algorithm, but instead a complementary component. Despite being somewhat of a hack, space insertion was necessary to map complex applications using only nearest neighbor communication. The space threshold value determines the complexity of an application and enables this component when need. However, the space threshold value is somewhat unreliable. The most substantial modification, not found in other implementations, was the addition of a route counter for each grid point. This modification enabled multiple datastreams to pass-through a single grid point. The target array is populated more effectively when grid points can be reused. A modification specific to this work was inserting new routing nodes into the graph for each grid point along a routable path. This modification is typically not applicable in most implementations since routing resources are often separate from computing resources. Each of these modifications attempt to either reduce routing conflicts or improve the utilization for the target array.

3.3.4 Summary

Maze routing is an efficient and flexible routing algorithm that works very well for the routing phase of the mapping algorithm. The implemented routing algorithm is able to find the shortest path in a short period of time under the constraints of the ASAP architecture. Not many

modifications had to be made to the maze routing algorithm with the only major modification being the space insertion component. Space insertion greatly improves the mapping quality for complex applications but there are likely other methods for inserting space that could further improve the mapping quality. For most applications the routing algorithm is able to successfully route every long-distance edge. Though for some applications routing conflicts occur that could be eliminated by routing edges in a different order. Research has already been done on ways to change the order paths are routed, but implementing this feature is left for future work. Overall the current implementation has been very successful in routing a wide variety of applications on the AsAP architecture.

3.4 Top-Level

The mapping algorithm is composed of both the placement phase and the routing phase. These two phases must somehow be combined together or the mapping algorithm won't be very useful. The top-level logic, which is responsible for combining these two phases, is performed by the `AlgorithmMain` function listed in Algorithm 3.29. This function also prepares the global variables, including the random seed, and performs some post-processing. This section describes in detail how the top-level logic is used to glue the placement and routing phases together.

The `AlgorithmMain` function is indirectly responsible for mapping applications to the AsAP architecture. This function relies upon the subfunctions `PlacementMain` and `RoutingMain` to do all of the heavy lifting. These two subfunctions have already been discussed in great detail in the previous two sections. The function starts by initializing all the global variables. The function then checks the size field of the global configuration object to ensure that the desired rectangular array area is not invalid. The desired rectangular array area will be considered invalid if it's either not set or too small for the application being mapped. If the desired rectangular array area is found to be invalid, the optimal rectangular array area will be calculated and the size field of the global configuration object will be updated with the calculated value. The optimal rectangular array area is the smallest rectangular array area that will hold the minimum number of processors. The minimum number of processors is calculated by adding the number of tasks in the application to the number of locations in the excluded processor list. The X dimension for the optimal rectangular array area is calculated first, which is done by taking the square root of the minimum number of processors and rounding the result up to the nearest integer. Next the Y dimension for the optimal rectangular array area is calculated by dividing the minimum number of processors by the previously calculated X dimension then rounding the result up to the nearest integer. The mapping process can begin

once the size field of the global configuration object is valid. The placement phase is executed first, followed by the routing phase if it's enabled. Once the mapping process is complete post-processing is applied. Post-processing involves shifting the array left and up until both the first column and the first row are no longer empty. This is also called aligning the array. Before shifting the array we need to calculate the inner coordinate. The inner coordinate is the upper-left corner of the array based on its contents, not its physical dimensions. Initially the inner coordinate is set to the bottom-right corner of the physical array. Next the location of every vertex in the graph is compared one-by-one to the inner coordinate. The inner coordinate is updated whenever a location is encountered that is further left or further up than the current inner coordinate. After every vertex location has been processed the inner coordinate contains the amount that the array must be shifted in order to align the array contents with the upper-left corner of the physical array. The number of times the array is shifted left is equal to the X dimension of the inner coordinate. The number of times the array is shifted up is equal to the Y dimension of the inner coordinate. Shifts are applied one position at a time instead of all at once in case a failure occurs. This technique results in the largest possible shift without violating any constraints. The function then returns and the mapping is complete.

Four configuration parameters are used by this function. The *Config.RandSeed* parameter contains the random seed used to initialize the random number generator. Setting the random seed to a known value at the beginning of the mapping algorithm results in a reproducible sequence of random numbers. The *Config.ExcludeList* parameter is a list of processors within the target array that should never have tasks assigned to them. The length of this list is used to calculate the minimum number of processors which is in turn used to calculate the optimal rectangular array area. The *Config.Size* parameter contains the desired rectangular array area for the target array. If this parameter is initially invalid, which is often done on purpose using the coordinate $(-1, -1)$, then the optimal rectangular array area is calculated and replaces the value of this parameter. The *Config.UseRouting* parameter determines whether or not the routing phase is executed. One possible reason for skipping the routing phase is when mapping applications to the second version of AsAP where routing processors are not always needed.

The constants *VERTICAL* and *HORIZONTAL* are simply enumerations used to distinguish the type of operation being performed.

Algorithm 3.29 AlgorithmMain

AlgorithmMain(*Graph*, *Config*) : entry point for the mapping algorithm, maps the pre-defined dataflow graph *Graph* onto the given architecture using the parameters in *Config*

```

set global variable Graph  $\leftarrow$  Graph
set global variable Config  $\leftarrow$  Config
let initial random seed  $\leftarrow$  Config.RandSeed
let Count  $\leftarrow$  number of nodes in Graph + length of Config.ExcludeList
if Config.Size is invalid or area of Config.Size < Count then
  let Config.Size.X  $\leftarrow$   $\lceil \sqrt{\text{Count}} \rceil$ 
  let Config.Size.Y  $\leftarrow$   $\lceil \text{Count} / \text{Config.Size.X} \rceil$ 
end if
PlacementMain()
if Config.UseRouting = true then
  RoutingMain()
end if
let Inner  $\leftarrow$  Graph.Size
for each Vertex in Graph do
  let Coord  $\leftarrow$  Vertex.Coordinate
  let Inner.X  $\leftarrow$   $\min(\text{Inner.X}, \text{Coord.X})$ 
  let Inner.Y  $\leftarrow$   $\min(\text{Inner.Y}, \text{Coord.Y})$ 
end for
for PosX  $\leftarrow$  Inner.X to 1 do
  if ShiftArray(HORIZONTAL, PosX, -1) = false then
    break loop
  end if
end for
for PosY  $\leftarrow$  Inner.Y to 1 do
  if ShiftArray(VERTICAL, PosY, -1) = false then
    break loop
  end if
end for

```

3.5 Conclusion

In summary the mapping algorithm consists of two phases, a placement phase and a routing phase. The placement phase is based on simulated annealing but required a few modifications to the base algorithm. The routing phase is based on maze routing and includes an additional space insertion component. Both phases contain some unique optimizations that have greatly improved the mapping quality. Even though the mapping algorithm is somewhat intended for the first version of AsAP, the algorithm can easily be re-targeted for the second version of AsAP using just a few of the many runtime configurable parameters. I believe the most valuable characteristic of the mapping algorithm is the framework it provides for mapping arbitrarily connected task graphs onto 2D-mesh nearest neighbor dominated parallel arrays. This framework is by no means complete with work still to be done on improving the mapping quality.

Chapter 4

Implementation

Programming parallel array processors like AsAP is very different than programming general purpose processors. General purpose processors usually have only a few cores and one large shared memory for storing variables. For these architectures a programmer typically writes one large program that is intended to be executed sequentially. For parallel array processors there can be as many as 1000 cores with each core having its own memory for storing variables. Variables must be passed from core to core in order for them to be shared. For these architectures, many small programs must be written or generated, one for each core, and are executed in parallel. The fact that these programs can operate independently and in parallel is the reason the AsAP mapping tool is possible.

The programming methodology used by the AsAP mapping tool is very similar to Matlab's Simulink. Simulink provides a natural way to describe signal processing systems by visually connecting together basic building blocks. When writing an application for AsAP the first step is to write a number of small independent kernels. Each kernel is then converted into a module, which is a highly reusable and sometimes configurable, building block used for creating applications. Connecting modules together is as simple as drawing a line from one module to another. What makes the mapping tool unique is the ability to build applications (from existing modules) by simply describing their dataflow. The remainder of this chapter describes the AsAP mapping tool's implementation.

4.1 Back-end

The mapping tool back-end is an implementation of the mapping algorithm described in Chapter 3. The back-end receives the dataflow graph and configuration parameters from the front-

end and uses them to execute the mapping algorithm. The mapping algorithm assigns each task in the dataflow graph a coordinate in the array and also adds a new task to the graph for each routing processor inserted. Coordinates are assigned in a way that maximizes nearest neighbor communication and minimizes area. Configuration parameters allow the location of tasks to be fixed, processors to be excluded from the mapping, and the mapping quality to be improved for certain applications. The back-end is actually capable of mapping applications to any large scale parallel array not just specifically AsAP. The procedure for the mapping algorithm has already been discussed in great detail in Chapter 3 so only the implementation will be discussed here.

The mapping tool back-end, also called the mapping library or *libamap* for short, is a statically linked library written in C++. The primary motivation for choosing C++ was that the boost graph library is written in C++ [1]. Also C/C++ compilers produce very optimized code. This is important since the mapping library accounts for more than 99% of the runtime when mapping an application. The boost graph library was designed using C++ templates so an unlimited number of vertex and edge properties can be added to the graph. Also boost graph objects can be used with many of the Standard Template Library (STL) functions to save coding time. The boost graph library is a member of the boost libraries. The boost libraries are a collection of generic programming libraries using C++ templates. In addition to the boost graph library, the boost random library, the boost date time library, and a few other members of the boost libraries are used in the implementation of the mapping algorithm. The mapping library has a simplistic API that makes it easy to include in tools other than the AsAP mapping tool. The mapping library has been tested on Windows, Linux, and MacOS X, but it is likely compatible with other POSIX platforms that are supported by the boost libraries.

4.2 Front-end

The mapping tool front-end was designed primarily to be an interface between Extensible Markup Language (XML) module files and the mapping library. The mapping tool has two modes of operation, graphical mode, and batch mode. Graphical mode makes it easy to construct and analyze complex applications using an intuitive interface. In this mode applications are designed by simply dragging modules onto a canvas and drawing lines between them to create the dataflow. An user can also look at the array layout and processor code for a module by just double-clicking. Batch mode provides a way to map applications without user intervention. This mode can be used to experiment with different configuration parameters or iterate over many random seeds. By using shell scripts

this can all be done in an automated fashion to try and improve the mapping quality. Unlike the back-end, the front-end has been designed specifically for the first and second versions of AsAP. Although the implementation is modular so it's easy to adapt the front-end to other architectures. The remainder of this section will discuss the different phases the front-end goes through and how they were implemented.

The mapping tool front-end, or *asapmap* for short, is both a graphical user interface and a command-line program written in C with some C++. Most of the application is written in C, since the gnome libraries are written in C [2], but C++ is used to create the dataflow graph and set the configuration parameters for the back-end. The gnome libraries are a collection of libraries that target the gnome platform used by Linux. These libraries contain functions for many common operations such as displaying help files and creating about boxes. The primary motivation for using the gnome libraries was the gnome canvas widget. The gnome canvas widget is a special drawing widget where visual elements, such as rectangles and lines, can be associated with events such as mouse clicks and mouse movements. Without this widget, creating applications wouldn't have been as simple as dragging modules and drawing lines. In addition to the gnome libraries, libxml is used for reading and writing XML files, and libglade is used for designing the user interface. These two extra libraries are closely tied to the gnome platform and are often included with the gnome libraries. The complete mapping tool, including both the front-end and the back-end, has been tested on Windows, Linux, and MacOS X (using X11), but Linux is the primary development platform.

4.2.1 Procedure

The mapping tool front-end goes through three phases when mapping an application. These three phases are separate from the two phases performed by the mapping library. All three phases are required when mapping any application. The first phase populates the module list and connects processors together. This can either be done by loading a saved project file or by creating an application manually using the mouse. The second phase executes the mapping algorithm and analyzes the results. The mapping algorithm objects, which are the dataflow graph and the configuration parameters, are created during this phase. The third phase modifies the modules used by the application then combines all the processors into one large module. Modifications to modules are made using a set of translation objects, which are created during the second phase. The complete process from loading the application to saving the results is described in further detail in the remainder of this subsection.

The first phase is responsible for creating a list of modules and linking together processors both inside and across modules. The module list is created by either dragging modules onto the canvas or by loading a previously saved project file. When a module is either dragged onto the canvas or loaded from a project file its XML file is parsed into memory and a new module is added to the module list. The XML file contents determine the number of inputs, the number of outputs, the number of parameters, how processors are locally connected, the name of the module, and many other little details. Processors are linked locally by finding matching pairs of opposing ports inside the XML file. If there is a mismatch between two opposing ports then a connection is not made. This is usually the case for module inputs and outputs where there is no local processor to connect to. To give an example, if the processor at location $(0, 0)$ has an east output and the processor at location $(1, 0)$ has a west input then the opposing ports match and they are connected. Processors are linked globally by either drawing a line between two modules on the canvas or using link elements inside a saved project file. Global linking is performed using module ids and port numbers to look-up which modules and which processors should be connected. The final step is to set the primary input and primary output for an application. This is done by either right-clicking on the desired port or by the input and output elements inside a saved project file. Once this is done the first phase is complete and the dataflow graph is ready. In batch mode the only option for this phase is using a project file.

The second phase is responsible for executing the mapping algorithm and converting the results into meaningful translations. The first step of this phase is to prepare the mapping parameters. This is an optional step that is only required if changes need to be made to the default parameters. For batch mode any parameter changes are requested on the command-line before the mapping tool is started. Once the dataflow graph and the mapping parameters are ready they are converted into C++ objects which can be passed to the mapping tool back-end. When constructing the C++ graph object the output ports (since input ports are mirrored) for every processor, across the entire module list, are checked to see if they are connected and to which processor. A new vertex is added to the graph for each processor and a new edge is added to the graph for each connected output port. When constructing the C++ configuration parameters object each setting from the mapping parameters is simply copied over in an appropriate format. The algorithm is then executed, which computes the coordinates for every vertex including any vertices that were created as a result of inserting routing processors. For each new router vertex inserted into the graph, a new router module is instantiated and inserted into the router list (which is similar to the module list). The next step is to analyze the mapping algorithm results. This is done by comparing the original locations

and port directions to those returned by the mapping algorithm. A location translation is added to the processor if any changes were made to its location. A location translation is simply a field inside the processor where the new target location is stored. A port translation is added to the processor if any port directions were changed. A port translation consists of an array of port index numbers that transform an old port index into a new port index. While calculating port translations, if the distance between any two processors is non-nearest neighbor then a long-distance interconnect is added. A long-distance interconnect is a direct connection specified using the source and target processor locations and the source and target port directions. Processor ports are divided into two groups. The first 4 ports are used for nearest neighbor communication and the second 4 ports are used for long-distance interconnects. When the ports are grouped the routing overlay network, used by the second version of AsAP, is easier to configure. These translation sets are used by the third phase to produce a properly working output module. In graphical mode the final array configuration is displayed once this phase is complete.

The third phase is responsible for modifying the processors inside the original XML files then combining these processors into one large XML file. For each module or router that is loaded, the parameters, processor locations, and port directions are updated. To update the parameters for a processor the original values are replaced by the new values stored in the corresponding module object from the module list. To update the location for a processor, the original location is replaced by the new location stored when the location translation was created. To update the port directions for a processor each character in the input and output masks are run through the port translation array. To determine the new port direction the original port direction is first converted to a port index that is compatible with the port translation array. This port index is then used against the port translation array to obtain a new port index. This new port index is then converted back into the new port direction. After all the processor elements inside an XML file have been updated the processor elements are copied to the output XML file. This includes the code for each processor. Once every processor from every module in the module list and every router in the router list has been copied over, the long-distance interconnects are added to the output XML file. After adding the new module name and module description to the output XML file the third phase is complete.

4.2.2 Interface

In graphical mode almost everything is done interactively using the mouse and Drag-N-Drop [3]. Since graphical mode is mainly driven by the mouse most tasks are easier to accomplish

in graphical mode than in batch mode. Some tasks such as creating or modifying an application are only possible in graphical mode. The ASAP mapping tool has four major windows. The main window is for constructing and modifying applications. The array window and code window are for viewing module attributes. The mapping window is for configuring and executing the mapping algorithm. In addition to these four major windows there are two minor windows, a parameter window for changing module parameters, and a coordinate window for fixing processor coordinates. These two additional windows are rarely used. The purpose for each of the four major windows is discussed in more detail in the remainder of this subsection.

When starting the program in graphical mode the first thing the user sees is the main window. The main window is shown in Figure 4.1 with the module palette on the left and the application canvas on the right. When the program starts the module directory is scanned and an icon is added to the palette for each module file that is found. Whenever the mouse hovers over a module in the palette the status bar along the bottom of the window displays the module's description. To create an application modules are dragged from the palette onto the canvas. When a module is dragged onto the canvas its underlying XML file is parsed into memory and an appropriate figure is drawn. Any number of identical modules can be placed on the canvas at one time without any problems. Right-clicking on a module will allow the user to change module parameters and fix processor locations. In order to construct the application's dataflow, modules must be linked together. Modules are linked together by dragging a line from the output port of one module to the input port of another module. Connections can only be point-to-point. Once a port is linked it can not be used again (unless unlinked first). Right-clicking on any free port will allow the user to set the primary input or primary output, which are colored green in the figure. Anything that needs to be done in order to construct a new application can be done from this window.

The array window and code window are used for displaying module attributes. The array window is shown in Figure 4.2 and the code window is shown in Figure 4.3. The array window displays the processor layout for a module, which can be accessed by double-clicking the module on the canvas. Processors are drawn as rectangles with their coordinate in the upper-right and connections are drawn as lines with arrow heads to indicate their direction. Connections that are drawn in green represent module inputs and module outputs. Connections that are drawn in red represent non-nearest neighbor connections. When viewing a module that was created by mapping an application, the name of the module from where the processor originated from is shown in the center of the rectangle. The code window displays the different code files inside a module and can be

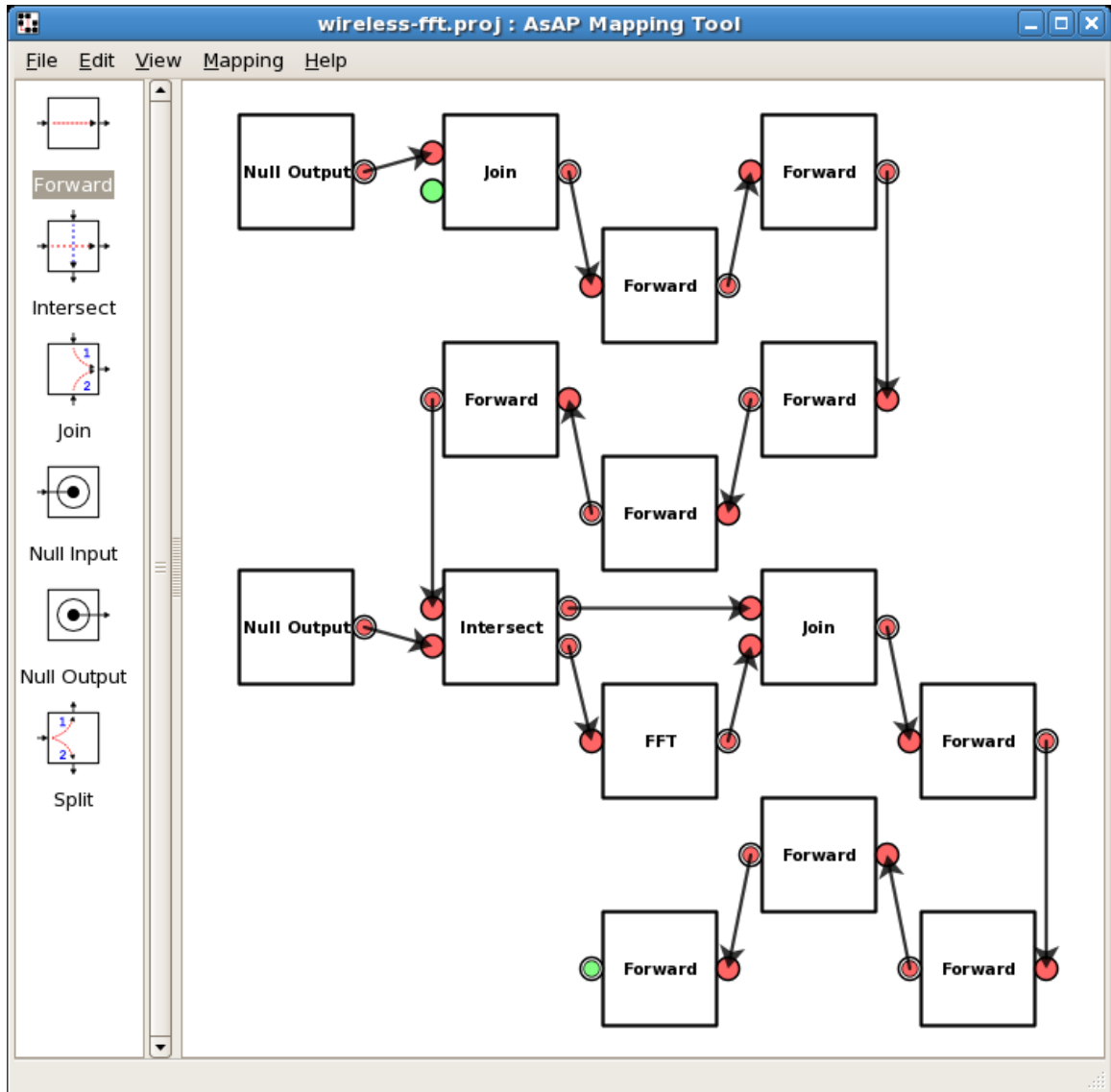


Figure 4.1: The main window of the ASAP mapping tool showing an 802.11a wireless transmitter application being created using an FFT module

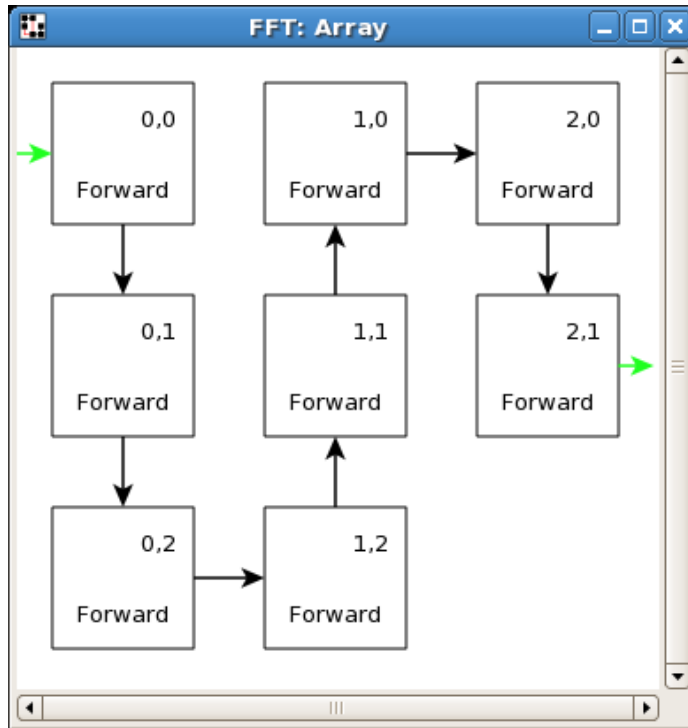
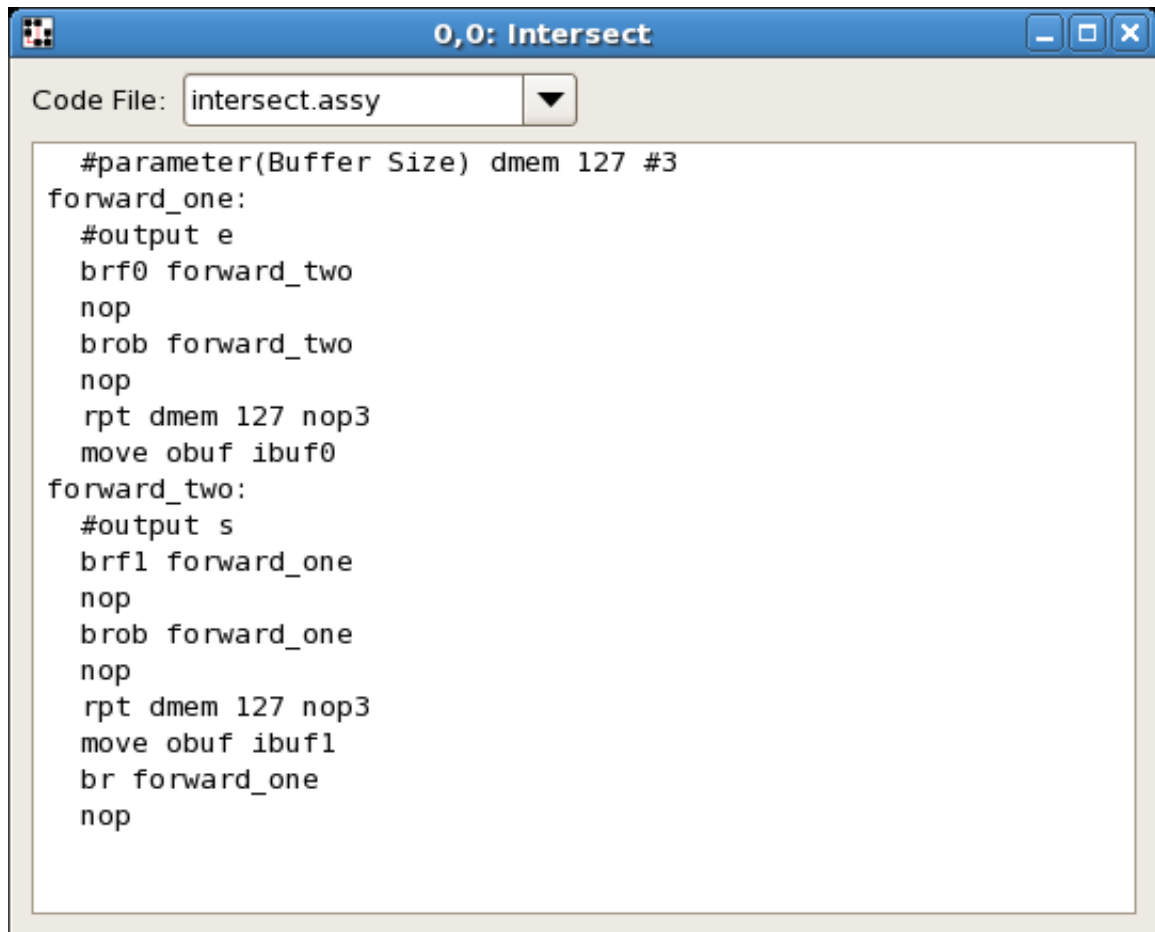


Figure 4.2: The array window of the AsAP mapping tool showing the array layout for the FFT module used by the 802.11a wireless transmitter

accessed by double-clicking a processor in the array window. A module can contain multiple code files because AsAP supports running multiple configuration programs before running the computation program. The active code file can be changed by selecting a new filename from the combo box at the top of the window. Some simple formatting is applied to the code, such as aligning labels and comments, to make it easier to read. The formatter only supports code files for the first version of AsAP, which will change once the second version of AsAP is ready. Viewing the code inside a processor can help determine its purpose or help the user select the correct input ports or output ports to use for linking. Without these two windows the user would need to look through the module's XML file in order to obtain this information.

Once an application has been created the next step is to map the application. The mapping algorithm can be configured and executed using the mapping dialog, which is accessible from the mapping menu of the main window. The three tabs of the mapping dialog are shown in Figure 4.4, Figure 4.5, and Figure 4.6, each containing a different category of configuration parameters. The first tab in the mapping dialog contains the array configuration parameters, which effect the structure of the mapping. These are parameters like the desired array size, the input location, and the output location. The second tab in the mapping dialog contains the algorithm configuration parameters,



The screenshot shows a window titled "0,0: Intersect" with a "Code File:" dropdown menu set to "intersect.assy". The code window contains the following assembly code:

```

#parameter(Buffer Size) dmem 127 #3
forward_one:
#output e
brf0 forward_two
nop
brob forward_two
nop
rpt dmem 127 nop3
move obuf ibuf0
forward_two:
#output s
brf1 forward_one
nop
brob forward_one
nop
rpt dmem 127 nop3
move obuf ibuf1
br forward_one
nop

```

Figure 4.3: The code window of the AsAP mapping tool showing the code for an intersecting routing processor used by the 802.11a wireless transmitter

which effect how the mapping is optimized. These are parameters like the random seed, the number of placement iterations, and space insertion. The third tab in the mapping dialog contains any remaining configuration parameters, such as the cost weights used by the configuration cost function. Only values from the array settings tab and the excluded processors are saved to the project file since the other parameters are most often varied in batch mode when trying to obtain the best mapping. The mapping dialog also contains a combo box at the top of the window for selecting a mapping profile. Mapping profiles change a number of configuration parameters at one time to achieve some goal. One example is the *Application Module* profile, which sets the configuration parameters for the array size, input location, and output location to match the first version of AsAP. Once all the configuration parameters have been set the execute button is clicked to start the mapping algorithm. Any changes made to the configuration parameters are saved automatically before the mapping algorithm is executed. Once the mapping algorithm finishes and the results

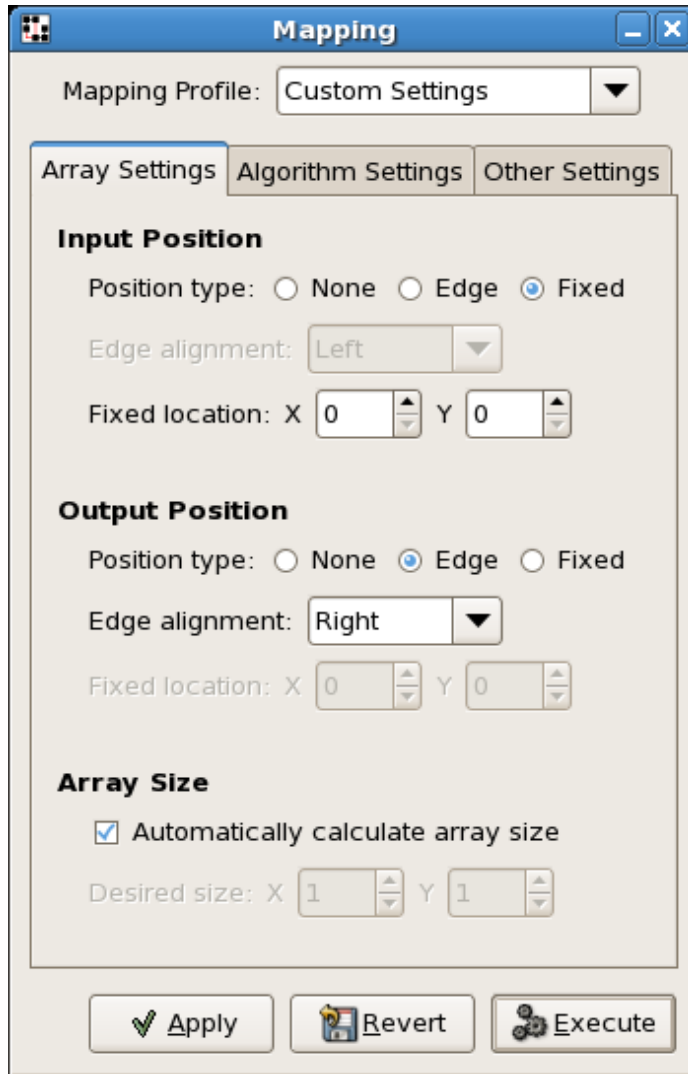


Figure 4.4: The array settings tab of the mapping dialog for the ASAP mapping tool, which is used for configuring the array input/output and the array size

have been processed, a new array window will appear which shows how the final module will look. The final module can be saved using the mapping menu from the main window. Since the mapping results are displayed automatically the mapping window provides an easy way to see how different parameters effect the mapping quality.

In batch mode everything is done entirely without user intervention. This mode requires an existing project file. This can either be created with scripts or created in graphical mode. All configuration parameters are specified on the command-line before the program is executed. The mapping results are saved to an output module file also specified on the command-line. Batch mode is extremely useful for performing a large number of mappings where the configuration parameters

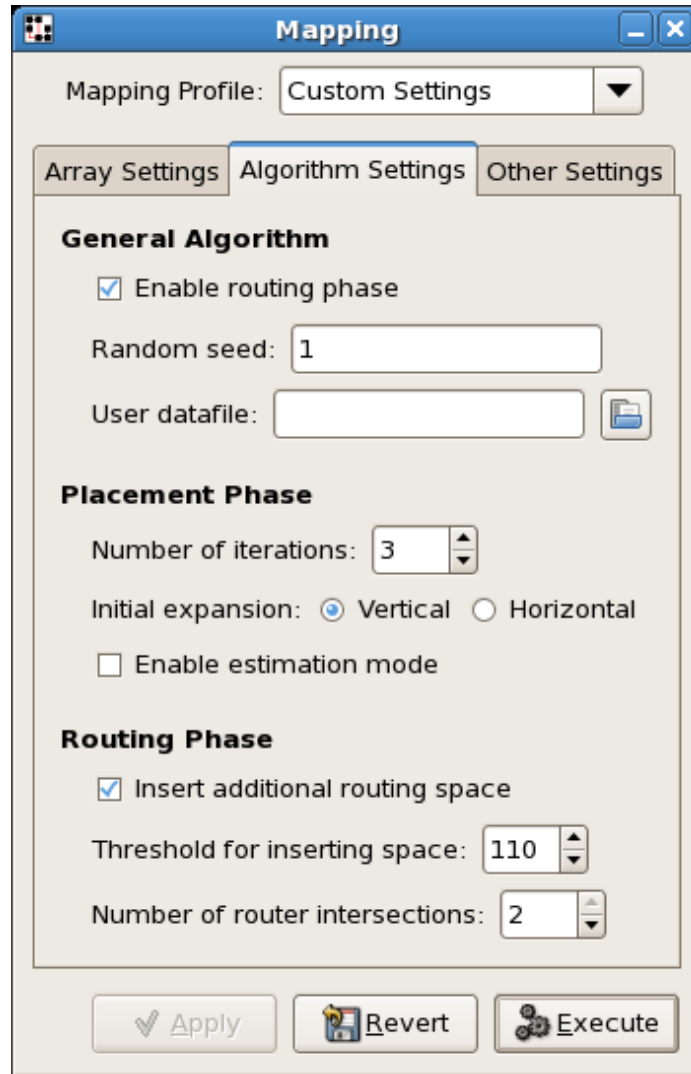


Figure 4.5: The algorithm settings tab of the mapping dialog for the AsAP mapping tool, which is used for configuring the placement and routing phases and the random seed

are varied to try and obtain the best mapping. When working with very large applications, with many levels of dependencies, batch mode can be used to update modules after changes have been made to its dependencies (using makefiles). Since everything is done from the command-line the results are not displayed once the mapping is complete. A submode of batch mode, called module view mode, provides a quick and easy way to view mapping results. Module view mode only displays the array window for the module then exits. Viewing the module in graphical mode requires the module to be placed on the canvas then double-clicked. Some tasks, such as executing the mapping algorithm, are quicker in batch mode than in graphical mode since it has a minimal amount of interaction.

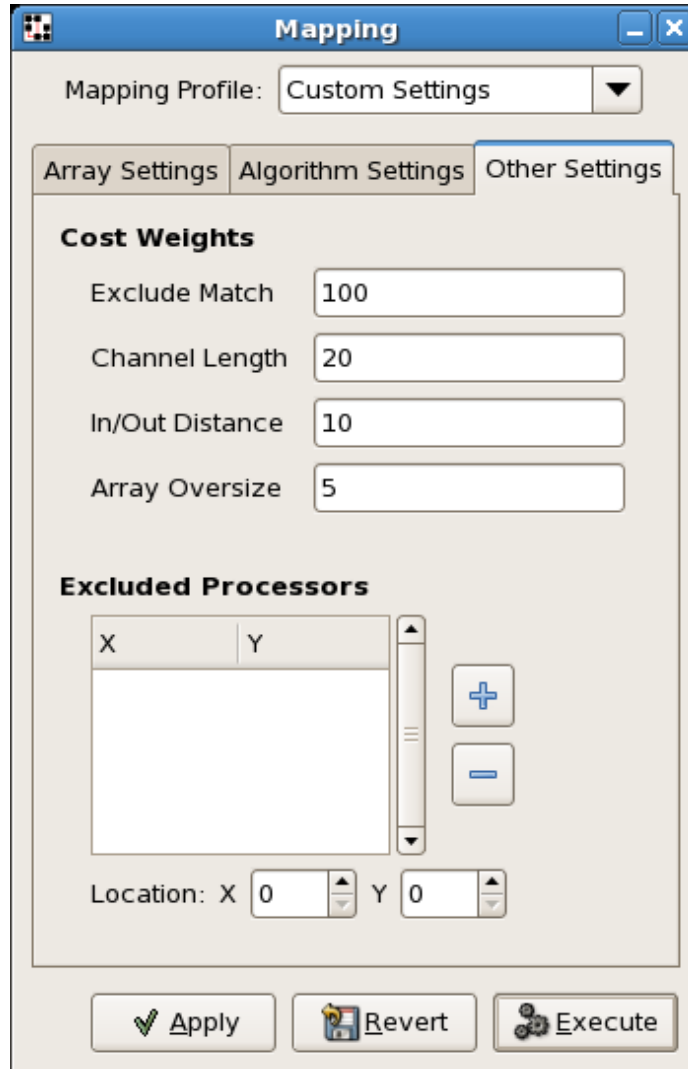


Figure 4.6: The other settings tab of the mapping dialog for the AsAP mapping tool, which is used for configuring the excluded processors and cost weights

4.3 XML File Formats

The AsAP mapping tool would be useless if it were unable to store modules and projects in an easily accessible and consistent format. For this reason modules and projects are stored in Extensible Markup Language (XML) files. XML has many advantages over other custom formats. Two of these advantages are: XML is an industry standard and files can be edited with a text editor. By choosing XML there were already a plethora of tools and libraries available for reading, writing, manipulating, searching, and viewing these types of files. The XML format is a very good fit for both module and project files due to its hierarchical nature. The remainder of this section will describe in more detail the contents and possible uses for these two file types.

4.3.1 Module Files

Modules files are the main unit of input and output for the mapping tool. Modules are loaded into the mapping tool when creating an application and modules are created by the mapping tool when saving mapping results. Allowing module files to contain vastly different levels of granularity is the very reason the mapping tool is so easy to use. One module file may contain a single processor while another module file may contain an array of 1000 processors. Even though these two module files are vastly different, they can be linked together with a simple dragging of the mouse. Regardless of how many processors are inside a module each module has a defined set of inputs and outputs, which conform to the AsAP FIFO interface. This allows any two modules to be seamlessly linked together. The XML module format is flexible enough that it could be used by other tools instead of just as an input and output format for the AsAP mapping tool. By storing processor code inside the XML file other tools can avoid the complexity involved in parsing AsAP assembly code.

An XML module file is self-contained meaning that everything needed to simulate the module is contained within the XML file. Each module file contains a number of processors, the layout for the processors, a list of long-distance interconnects, and even assembly code for each processor. Figure 4.7 shows the hierarchy of a module file when using the Document Object Model (DOM). We can see from the figure that each module file contains an array of one or more processors, which can each contain a number of configuration options and one or more code files. These code files contain the processor's input ports and output ports, which are used for linking processors together. The top-level input ports and output ports simply point to which processor ports other modules should connect to. After all, connecting processors globally is no different then connecting

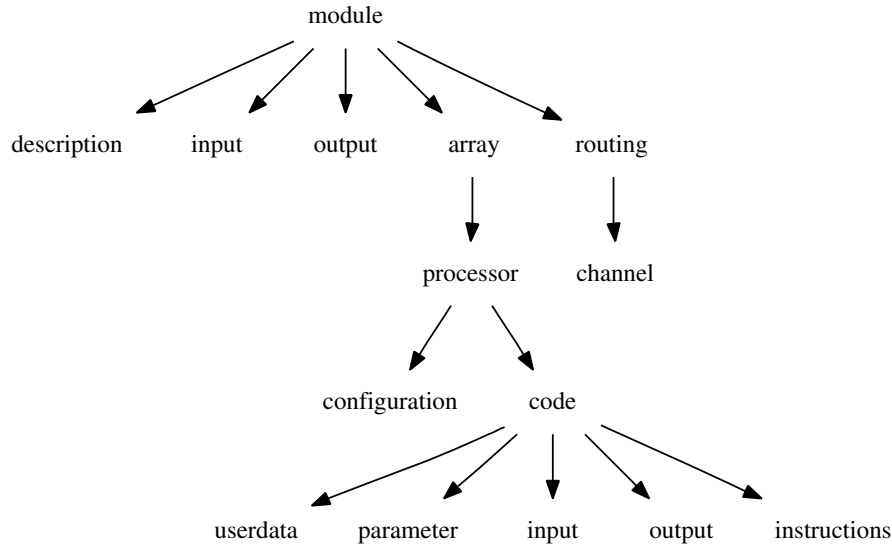


Figure 4.7: The Document Object Model hierarchy for XML module files

processors locally inside the mapping tool. Long-distance interconnects (or routing channels) are described similarly to module input ports and module output ports except that the two ports are connected together instead of left dangling. The XML module format should be easy to adapt to other parallel arrays since only the instruction elements are truly AsAP specific.

Module files are used quite extensively throughout the AsAP mapping tool. What is missing is a way to import AsAP assembly code, to perform mappings, and a way to extract AsAP assembly code, to perform simulations. To fill this void conversion scripts have been written that convert back and forth between AsAP flat files (assembly, configuration, etc.) and module files. The first script, *asap2mod.py*, will parse all flat files within the current directory and create a new module file. When executed it will prompt for a module name, a module description, and an icon filename, which are all used during graphical mode. The second script, *mod2asap.py*, will extract a module file into a number of flat files and place them in the current directory. The resulting flat files are ready to be simulated. These two scripts are included with the source code for the AsAP mapping tool. Since the AsAP architecture has only one primary input and one primary output, modules that require more than one primary input or one primary output must be edited manually after the conversion process. Parsing and storing C-code is more difficult than parsing and storing assembly code so this is left for future work.

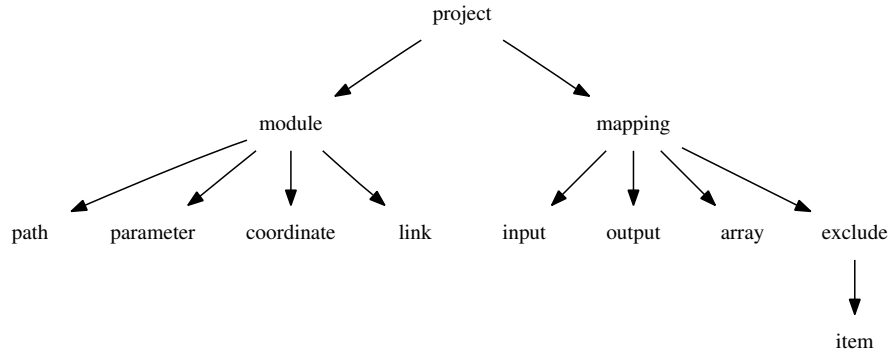


Figure 4.8: The Document Object Model hierarchy for XML project files

4.3.2 Project Files

Project files store applications that are in the pre-mapping stage along with some basic configuration parameters. These files essentially contain the dataflow graph for an application. This is why project files are the input format used for batch mode. After applications have been successfully mapped they are stored in module files. Since module files are self-contained, the project files are only needed to update the module files when any changes are made to their dependencies. The remainder of this subsection will explain in more detail how an application is rebuilt whenever a project file is loaded.

A project file at its very core is nothing more than a list of module references and a list of links for connecting them. Figure 4.8 shows the hierarchy of a project file when using the Document Object Model (DOM). We can see from the figure that each project file contains one or more modules, which each contain a number of properties. Project files maintain references to modules using filepaths and id numbers so that modules contain the latest information when they are reloaded again later. Project files maintain links between modules using id numbers and port numbers. When restoring a link the source and target modules are found using the id numbers and the source and target ports are found using the port numbers. One problem with linking modules this way is that when the number of ports on a module changes, or they are reordered, the project file will likely fail to load or function correctly. Module parameter values and fixed processor locations are also stored in the project file so they can be used in the final mapping. The graphical position for a module is stored in the module's location attribute so the application can be correctly redrawn when its reloaded in graphical mode. Rebuilding the application each time the project file is loaded is the key to creating applications with many levels of dependencies.

Figure 4.8 also shows that these project files can store some mapping parameters. The four mapping parameters stored in these project files are: the primary input and its position, the primary output and its position, the desired array size, and a list of excluded processor locations. The **input** element determines the primary input and its desired location. The primary input is determined using a module id number and a port number. There are three types of input locations, don't care, edge based, and fixed. When the input location is set to don't care the processor is placed in whatever location appears most optimal. When the input location is set to one of the four edges the processor is placed along the desired edge. When the input location is set to fixed the processor is placed in the location specified by an additional location attribute. The **output** element is the same as the **input** element except that it configures the primary output instead of the primary input. The **array** element determines the desired array size. If the size is set to $(-1, -1)$ then the optimal size is calculated, otherwise the desired size is used verbatim. The **exclude** element contains zero or more child elements each containing one excluded location. Each processor that matches a location in this list will not have a task assigned to it in the final mapping. Other configuration parameters are intended to vary between mappings so they are not stored in the project file.

4.4 Conclusion

New programming tools and programming languages are being developed in order to efficiently program parallel array processors. The AsAP mapping tool explores one possible method for programming parallel array processors, which is creating applications visually based on their dataflow. The AsAP mapping tool exploits the fact that kernels operate independently and in parallel on a homogeneous architecture. This allows applications to be constructed by simply chaining together tasks. Other research groups have been focusing on new programming languages that diverge from the sequential execution model we are accustomed to. StreamIt is just one of these programming languages [15]. Single chip parallel array processors have only recently started to appear. There is still work to be done before programming these architectures is as simple and common place as writing a C program for a general purpose processor.

Chapter 5

Evaluation Methods

The mapping algorithm is nothing more than an academic exercise unless real applications are actually mapped to the array and the results are evaluated. When evaluating an algorithm or processor we would typically use a set of benchmarks. This presents a problem since there are no established benchmarks for nearest neighbor dominated parallel arrays. Another problem that arises when evaluating the mapping algorithm is that mappings can be optimized in many different ways and even for specific chips. To overcome these problems an excellent set of metrics must be developed. These metrics must not only compare mappings to each other but also determine absolute quality. The absolute quality of a mapping helps to determine how far the mapping is from a theoretical optimal. In addition to excellent metrics a wide variety of applications are needed for performing evaluations. These applications should consist of dataflow patterns that are likely to be found in realistic applications. These applications should also contain dataflow patterns that will be difficult to map in order to stress the mapping algorithm. In this chapter a set of metrics and several applications are developed, which are used to evaluate the quality of the mapping algorithm in Chapter 6.

5.1 Quality Metrics

To evaluate the quality of the mapping algorithm a few metrics need to be established. These metrics represent the most desirable properties in the algorithm results. Each metric is expressible as a scalar quantity so different configurations can be compared. Since metrics have different priorities a scale factor is applied to emphasize the importance of one metric over another. The metrics used in this work improve as the value decreases. Therefore the best configuration has

the lowest value for each metric. The metrics have been chosen so the best possible configuration, which is also referred to as optimal, has a combined value of zero across all metrics.

5.1.1 Communication

For this metric the goal is to quantify the simplicity of mapping applications to a purely nearest neighbor parallel array. This metric primarily quantifies the length of communication channels between processors. It seems pretty straight forward to quantify this value as the combined length of all communication channels. It only requires a minor adjustment, which is to subtract the minimum channel length. The minimum channel length occurs when two processors are adjacent having a channel length of 1. For this work a value of 1 is subtracted from the length of each channel. This metric is incremented once for each channel with a length still greater than zero.

For the first version of AsAP the communication metric is the most important metric. If this value is greater than zero than the configuration can not be simulated. It may sometimes be acceptable to have a low number for this metric if the configuration can easily be modified by hand. To signify the importance of this metric, due to its criticality, a large scale factor is applied. A scale factor of 4 is applied to this metric, which is then squared. This result is then added to the optimization cost. A scale factor of 4 is used so the metric grows quicker linearly than the other metrics. The metric is squared to assure that even small values have a significant impact on the optimization cost. This scale factor is somewhat related to the *CostChannelLength* configuration parameter, which determines the penalty for using long-distance connections. When communication is weighted equally with area the array size is smaller, but the result is a significant increase in the number of long-distance interconnects. For the second version of AsAP this metric is still important to maximize nearest neighbor communication, or communication delay, but the scale factor can be reduced to decrease the area.

5.1.2 Area

For this metric the goal is to quantify the size of the array. This includes both the computation processors and the routing processors (if they were inserted). The easiest way to quantify this metric is to simply multiply the maximum x-dimension by the maximum y-dimension, which is the rectangular array area. The problem with quantifying the array size using this method is that the value will never reach zero unless the array has no processors or routers. To overcome this problem the optimal rectangular array area is first calculated then subtracted from the actual rectangular

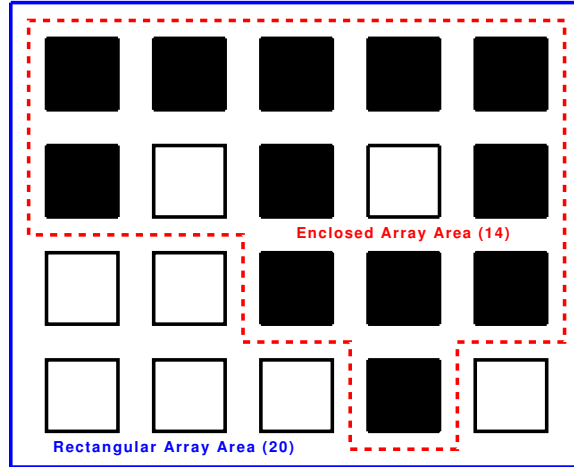


Figure 5.1: Visual depiction of the difference between rectangular array area (in solid blue) and enclosed array area (in dotted red) for a graph with 12 nodes

array area. The optimal rectangular array area is the smallest rectangular array area that will fit every node in the graph, excluding routing processors and ignoring data dependencies. The formula used to calculate the optimal rectangular array area is shown in Equations 5.1, 5.2, and 5.3.

$$DimX = \lceil \sqrt{NumNodes} \rceil \quad (5.1)$$

$$DimY = \lceil NumNodes / DimX \rceil \quad (5.2)$$

$$Optimal Area = DimX \times DimY \quad (5.3)$$

Since larger arrays can execute multiple applications at one time it's sometimes desirable to calculate the enclosed array area. The difference between the rectangular array area and the enclosed array area is roughly the number of processors available for other applications. To calculate the enclosed array area, processors not enclosed by the application are subtracted from the rectangular array area. Locations that are unoccupied, have fewer than three occupied neighbors, and are not surrounded by occupied processors are not considered enclosed and can be subtracted. Figure 5.1 gives an example of the difference between the rectangular array area and the enclosed array area. The rectangular array area is used for quantifying this metric instead of the enclosed array area because the rectangular array area is much simpler to calculate.

Just like in VLSI, the smaller the design the better (assuming consistent performance). For AsAP smaller designs typically consume less power since more processors can be turned off. Also communication channels are typically shorter since fewer hops are required, thereby decreasing communication delay. By having a smaller design more applications can be fit onto a single chip. The area metric is important for both the first and second version of AsAP. For the second

version of AsAP this metric may have equal importance and sometimes more importance than the communication metric. For this work the metric is multiplied by a scale factor of 2 before adding it to the optimization cost. A scale factor of 2 was chosen so the metric will grow quicker than the utilization metric but not as quick as the communication metric. This scale factor is somewhat related to the *CostArraySize* configuration parameter, which determines the penalty for increasing the array dimensions. When area is weighted equally with utilization the array tends to be a little larger since the focuses shifts towards removing routing processors.

5.1.3 Utilization

For this metric the goal is to quantify the percentage of the array used for computation as opposed to routing. This percentage is derived from the number of computation processors with respect to the total number of processors (including both computation and routing processors). In other words an increase in utilization means a decrease in the number of routing processors. The value for this metric is simply the number of routing processors inserted into the mapping. There is no need to calculate the actual percentage since the number of computations processors is fixed for a single application. This metric is not as significant as the other metrics but it allows very similar configurations to be compared.

In AsAP, when a routing processor is inserted an additional hop is needed to pass data between any connected nodes. Minimizing the number of routers decreases the number of hops between nodes, which in-turn decreases communication delay. Also when fewer routing processors are used less area is required to map the application so power can be saved by turning off additional processors. This metric has no scale factor applied and is simply added to the optimization cost. No scale factor was applied to this metric because it's less important than the other metrics that have a much larger impact on the mapping qualities we desire.

5.1.4 Runtime

For this metric the goal is to quantify the time required to obtain a given mapping. This metric does not actually reflect the quality of the mapping but instead the quality of the implementation of the mapping algorithm. The runtime is calculated by summing the time expended for all previous mapping attempts up to the selected mapping. The runtime is useful for comparing hand mappings to automatic mappings. All mappings were performed on a lightly loaded cluster with two systems containing 2.0 GHz Intel Xeon processors, four systems containing 2.4 GHz Intel

Operation	Runtime
Configuration Cost	51.49 %
Perturb Configuration	22.76 %
Graph Object Manipulation	15.68 %
Random Number Generation	10.07 %
Other Operations	00.00 %

Table 5.1: A breakdown of the runtime for the mapping algorithm while mapping the 802.11a wireless transmitter using the default settings.

Xeon processors, and one system containing 3.2 GHz Intel Xeon processors, all dual-socket with hyper-threading enabled (donated by the Intel Corporation). Testing was initially done on my home computer, which contains two 1.6 GHz AMD Opterons processors.

One problem with the mapping problem discussed in this work is that the solution space is essentially infinite since the array size can grow without bounds. For this reason the runtime is important in order to determine how quickly near-optimal mappings can be obtained. When near-optimal mappings are obtained quickly time can be spent improving the mapping by hand to obtain a very good mapping. For this metric a perfect value of zero seconds will never be attainable unless the application was already mapped. This metric is therefore not included in the optimization cost, but it is interesting to see how other metrics decrease over time.

The mapping algorithm consists of two phases with each phase having many parts. To get a better idea for where time is spent the mapping algorithm was profiled while mapping the 802.11a wireless transmitter application, described later in this chapter. Table 5.1 shows how the runtime is broken down and what operations require the most time. Operations are listed by category instead of function name for simplicity. Most of the time is spent calculating the configuration cost, which makes sense since it's the most complex part of the algorithm. Perturbing the configuration is the second most complex part of the algorithm so it makes sense that this operation is second in the table. Random number generation is used primarily when perturbing the configuration but this operation has been put into its own category because the percentage of time it consumes is significant. All categories, with the exception of a small fraction of the *Graph Object Manipulation* category and the *Other Operations* category, belong to the placement phase. The remainder of the runtime belongs to the routing phase. Based on the data in this table it's quite obvious that most of the time is spent in the placement phase of the mapping algorithm. The routing phase is entirely predictable so it doesn't require the time needed by the placement phase to search the solution space. Configuration parameters that effect the placement phase, like the number of iterations, result in the biggest runtime savings. Although runtime savings usually result in lower mapping quality.

5.1.5 Summary

In order to compare one configuration against another we must combine the three quality metrics into one comparable quantity. This quantity is called the optimization cost. The lower the optimization cost the better the mapping. If the optimization cost is zero then the mapping is considered optimal. Equation 5.4 shows how the optimization cost is calculated for a given mapping using the three metrics. This simple cost equation is used to evaluate mappings throughout Chapter 6 in order to find the best mapping for an application. The optimization cost is different than the configuration cost, though there are a number of similarities. The configuration cost is used by the placement phase of the mapping algorithm to determine whether or not to accept a perturbation. The optimization cost equation is a simpler version of the configuration cost function and is used for comparing final mappings, not intermediate mappings.

$$\textit{Optimization Cost} = (4 \times \textit{Communication})^2 + (2 \times \textit{Area}) + \textit{Utilization} \quad (5.4)$$

5.2 Applications

To evaluate the quality of the mapping algorithm a set of applications must be mapped then analyzed. The following applications vary in size and structure. Each application was considered because of its dataflow patterns or its popularity in the signal processing field. These applications are used throughout Chapter 6 to demonstrate various properties of the mapping algorithm.

5.2.1 Building Blocks

Applications are created by combining a number of kernels that have each been coded for a specific task. For this work the code inside each kernel is not as important as the application's dataflow (how the kernels are connected). Instead of writing actual code for each module, pre-built modules are used to simplify the design. These pre-built modules, or basic building blocks, contain the necessary dataflow components needed to build any application. There are six basic building blocks, which are shown in Figure 5.2. The **Null Input** and **Null Output** blocks act as a data sink and a data source, respectively. The **Null Input** block is used for either testing purposes or special measurements and therefore isn't used by any of the applications in this work. The **Forward** and **Intersect** blocks pass data between one or two pairs of processors, respectively. The **Split** and **Join** blocks split one datastream into two datastreams and join two datastreams back into one

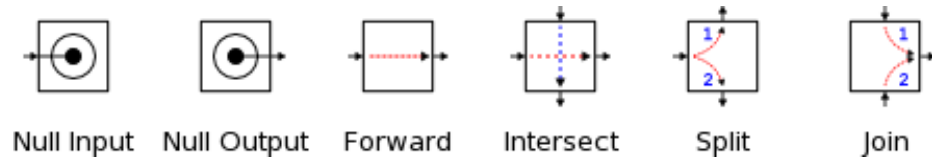


Figure 5.2: The basic building blocks used for creating applications

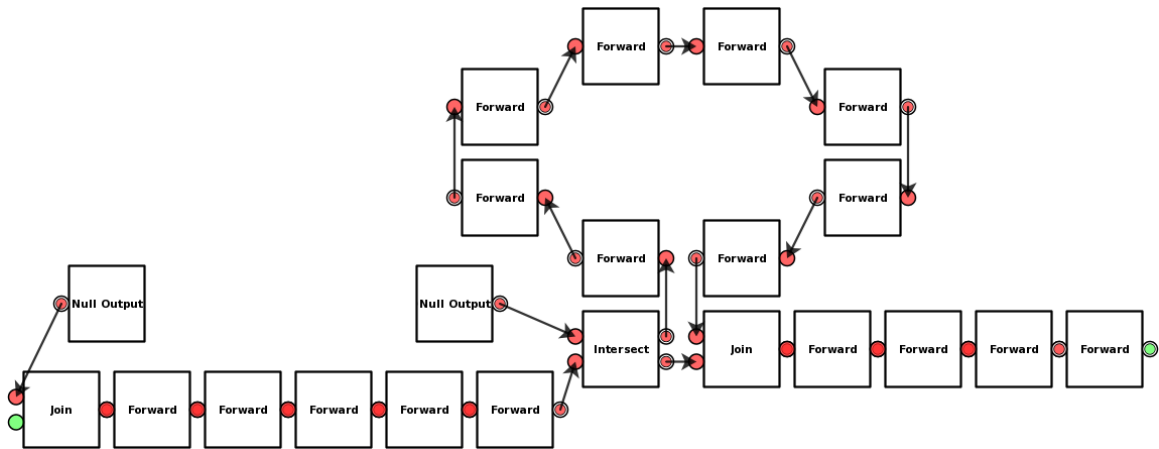


Figure 5.3: The dataflow graph entered into the mapping tool for the 802.11a wireless transmitter

datastream, respectively. Datastreams are split and joined using the round-robin method. Every application presented in this work is built using different combinations of these basic blocks.

5.2.2 802.11a Wireless Transmitter

Implementations of the IEEE 802.11a and 802.11g wireless LAN standard include a number of subsystems such as digital baseband processing, analog circuits, and high-frequency RF circuits. An important workload examined throughout this work is the processing required for the digital baseband transmitter. The 802.11a wireless baseband transmitter contains a number of common DSP components, such as filters, an FFT, and some codecs. The dataflow graph for this application is almost linear but it has been mapped previously to AsAP by another student, Michael Meeuwsen, which is useful for comparing hand mappings to automatic mappings [26]. The simplicity of this application also allows more restrictive mapping parameters to be tested. The dataflow graph for the 802.11a wireless transmitter is shown in Figure 5.3.

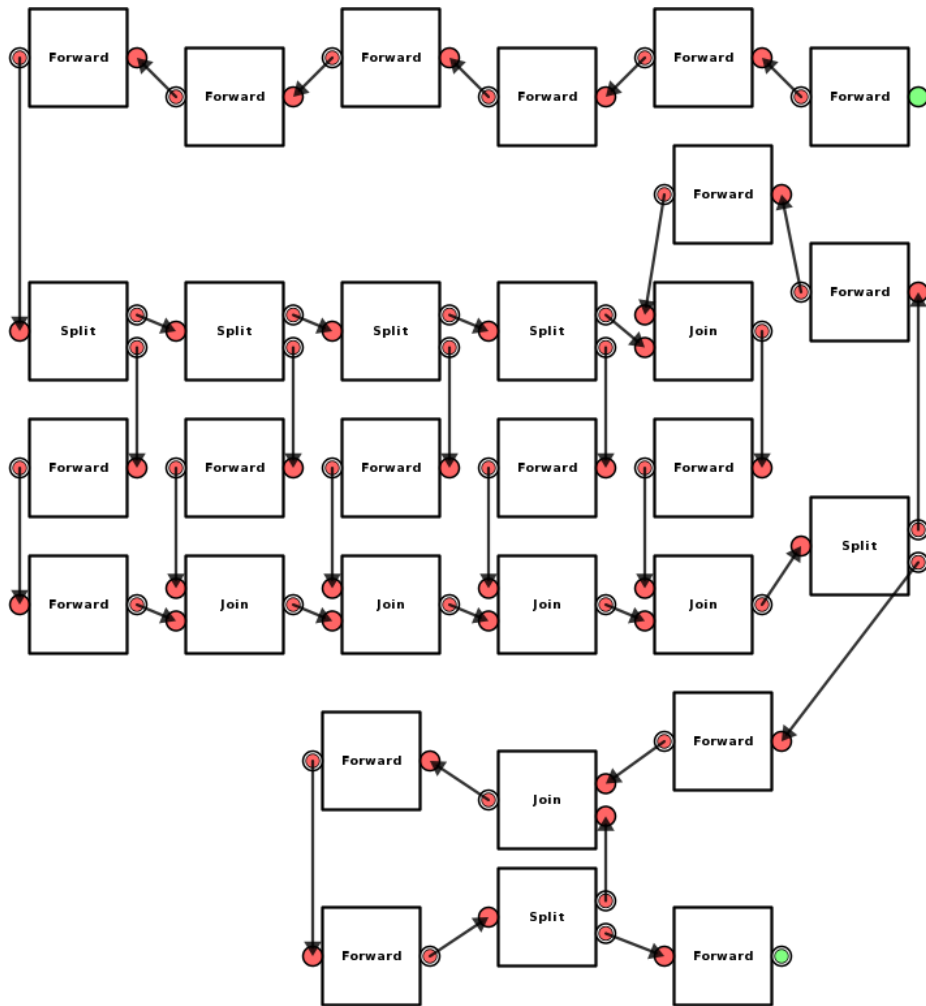


Figure 5.4: The dataflow graph entered into the mapping tool for the Viterbi decoder

5.2.3 Viterbi Decoder

Viterbi is an error correction code algorithm that performs fairly well and requires relatively little computation [34]. This algorithm is a very common component in a number of wireless standards, such as IEEE 802.11. Viterbi first creates a trellis through a series of adds, compares, and selects which is then followed by a traceback phase to determine the original code. This algorithm has also been previously mapped to AsAP by another student in our research group, Daniel Gurman, and was given to me in private communication. This algorithm also has an interesting dataflow graph with a number of small feedback loops. The dataflow graph for the Viterbi decoder is shown in Figure 5.4.

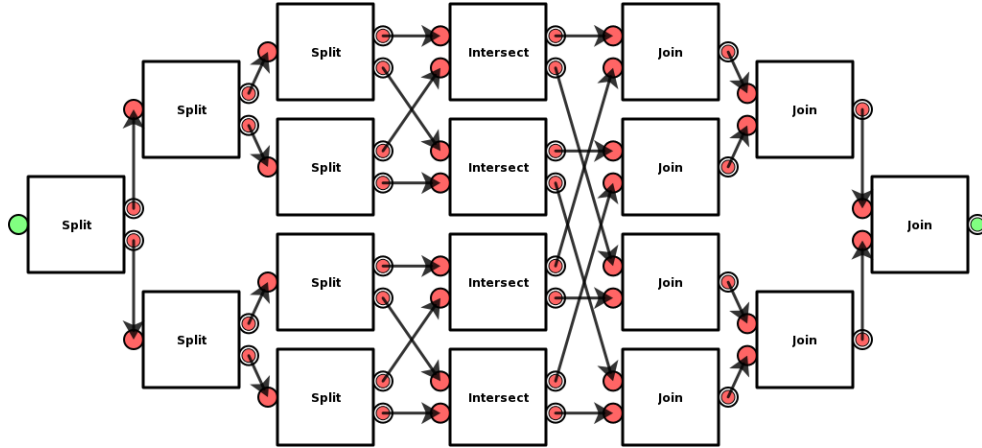


Figure 5.5: The dataflow graph entered into the mapping tool for the Fast Fourier Transform

5.2.4 Fast Fourier Transform

The Fast Fourier Transform is a very common building block in many DSP systems. The Fast Fourier Transform, or FFT, is used to transform an N -point discrete time domain signal into a frequency domain signal required by many algorithms. There are many forms of FFTs but the form used in this work is the Cooley-Tukey Radix-2 Decimation in Time [28]. The size of the FFT is undefined as only the dataflow for a block based FFT is important for this work. The FFT was chosen because of its high fan-out and fan-in as well as its regular communication structure. The dataflow graph for the Fast Fourier Transform is shown in Figure 5.5.

5.2.5 Clos Networks

A Clos network is a multistage switching network designed to route data between a large number of nodes when switches with only a small number of ports are available. There are several methods for combining these switches so they resemble larger versions of the smaller switches. The Clos networks used in this work are $N \times N$ non-blocking networks, which allow N nodes to talk to another N nodes simultaneously. These Clos networks are based on designs from the book *Principles and Practices of Interconnection Networks* by Dally and Towles [12]. Two different size networks are used, a smaller 4×4 network, and a larger 8×8 network. The larger 8×8 network was previously mapped by an employee at Google, Wei-Hwa Huang, and was given to me in private communication. His mapping offered some guidance when mapping the other complex applications by hand. Clos networks were chosen because of their complex routing paths which have many crossing connections. The dataflow graphs for the small Clos network and the large Clos network are shown in Figure 5.6

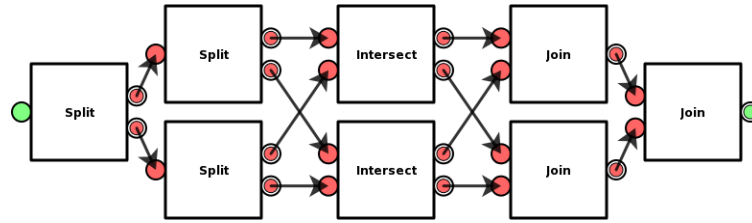


Figure 5.6: The dataflow graph entered into the mapping tool for the small Clos network

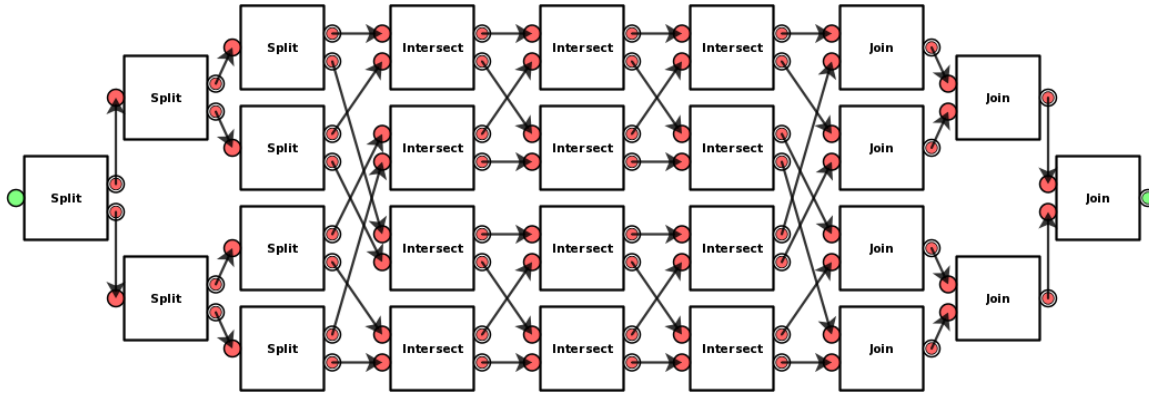


Figure 5.7: The dataflow graph entered into the mapping tool for the large Clos network

and Figure 5.7 respectively.

5.2.6 Random Graphs

Randomly generated graphs, which are converted into pseudo-applications, are used by a few of the tests in Chapter 6. Each random application has the same relative mapping difficulty. The algorithm used for generating these applications starts by selecting between two types of constructs, a forward chain or a split/join, which are shown in Figure 5.8. Each choice is weighted by some probability which makes choosing one construct more probable than the other. The forward chain ranges from 1 to 5 nodes with each possibility weighted equally. The splits/joins can have either 2, 4 or 8 branches, with a probability of $3x$, $2x$, and $1x$ respectively. Between the split nodes and the join nodes another random selection is made from the two constructs but the forward chain is given a higher probability to keep the application from growing too large. Constructs continue to be added until finally only forward chains are selected. This entire process is then repeated until the total number of nodes exceeds a given minimum. All randomly generated segments are then connected in series with the first and last segments connected to the module's input and output respectively. Examples of these randomly generated graphs can be seen in Figure 5.9. Since the number of nodes

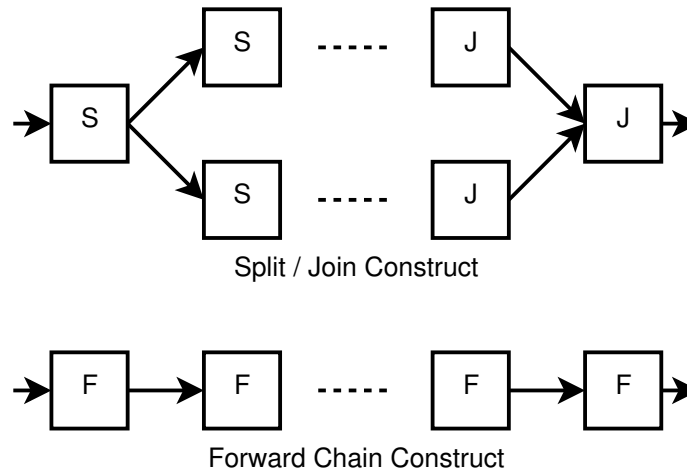


Figure 5.8: Basic constructs used to build the random node applications

can exceed the given minimum the script must be run multiple times to get an exact number of nodes.

5.2.7 Multi-App Application

The multi-app application is constructed by chaining together a number of smaller applications as shown in Figure 5.10. The applications combined are, the 802.11a wireless transmitter, the Fast Fourier Transform, and the Viterbi decoder, in that order. The resulting dataflow graph is shown in Figure 5.11. The principle behind creating an application of this type is that each of the smaller applications create clusters which could present some interesting challenges for the mapping algorithm. This is a typical scenario for ASAP where multiple applications are combined to make up a complete system. It also wouldn't be unexpected to put a few independent applications onto a single chip when there are a very large number of processing elements.

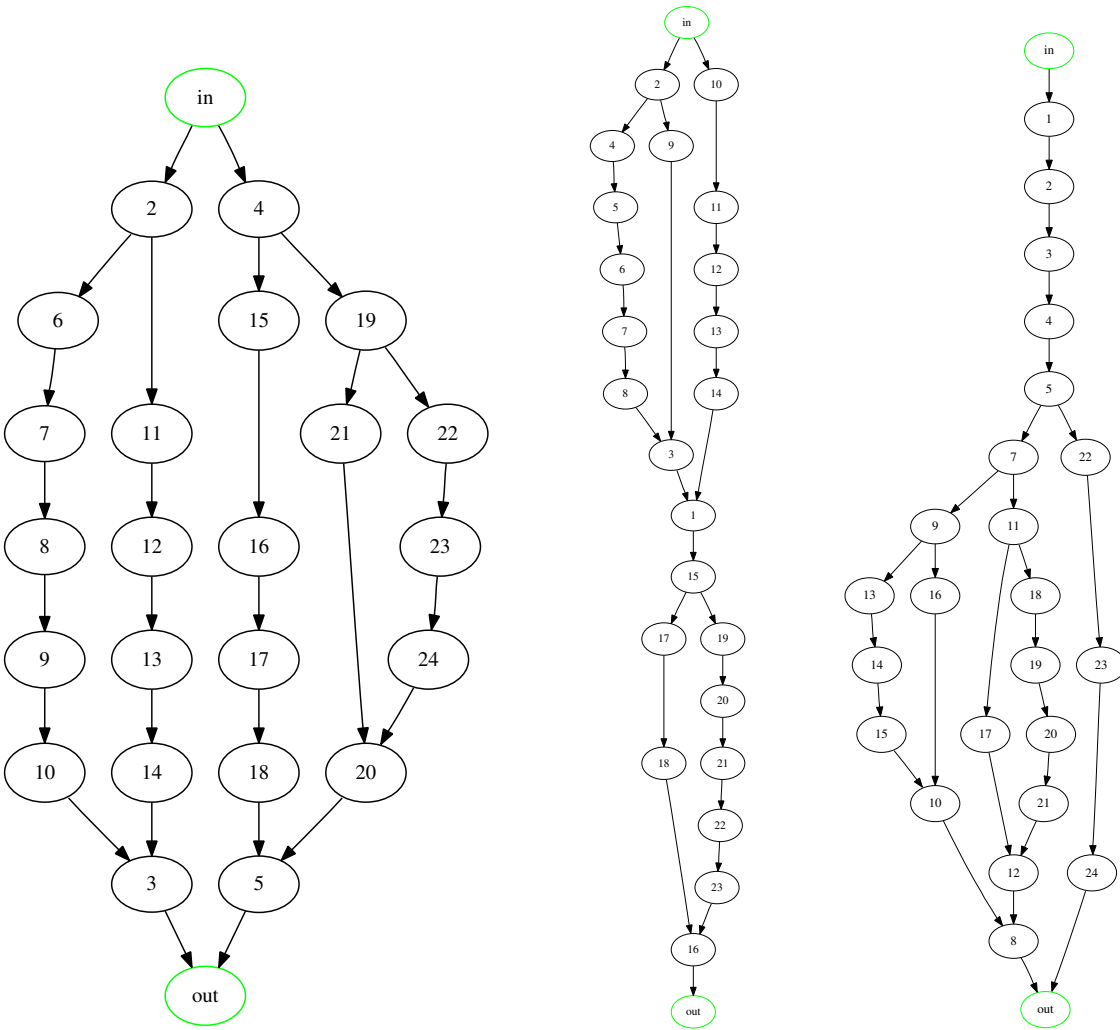


Figure 5.9: Examples of randomly generated graphs that are used for the random node applications. Each graph shown here contains 25 nodes with the input and output nodes colored green.

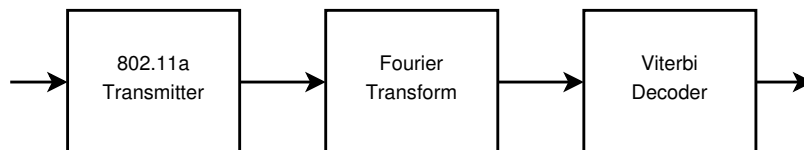


Figure 5.10: Applications used for constructing the multi-app application

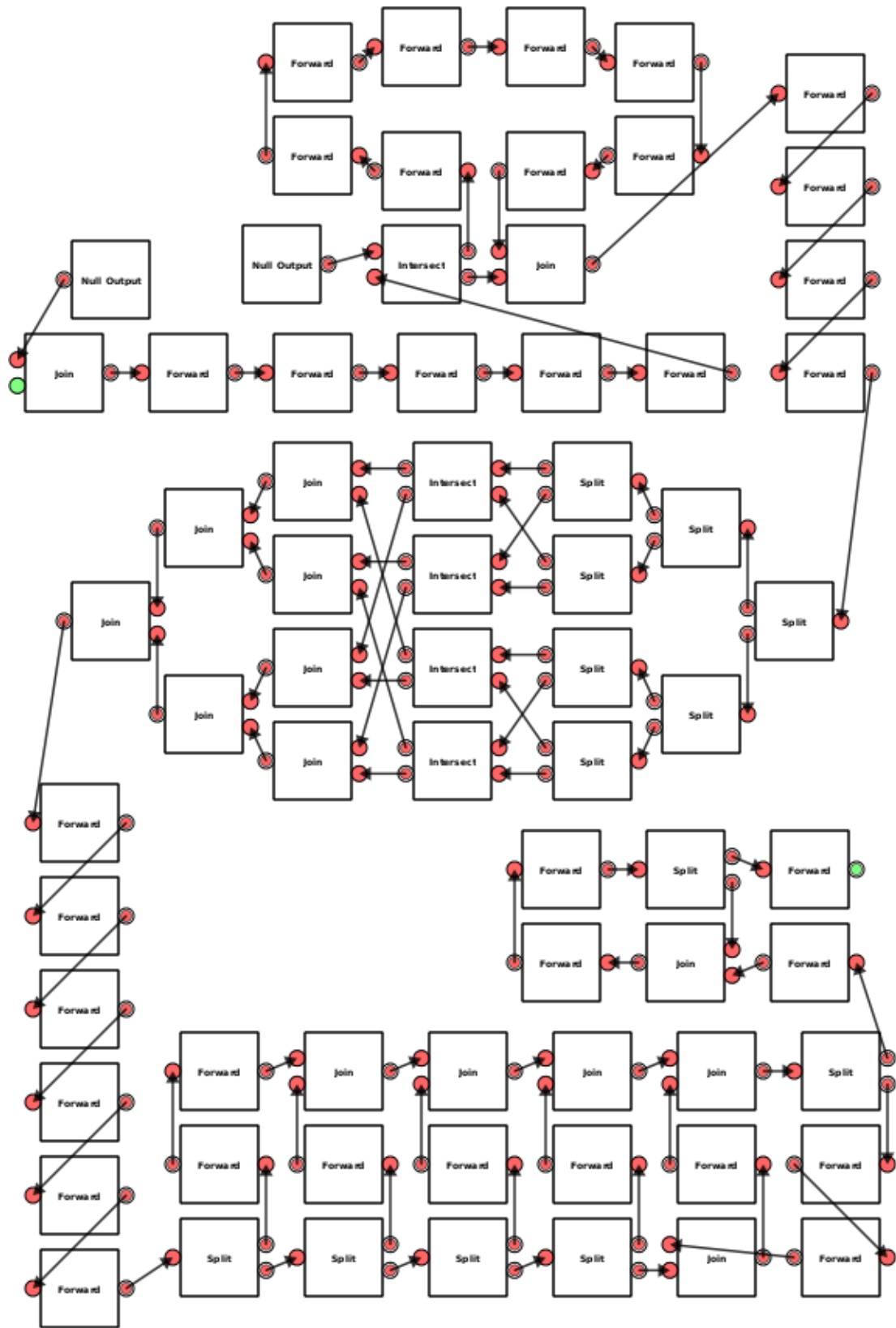


Figure 5.11: The dataflow graph entered into the mapping tool for the multi-app application

5.3 Conclusion

The mapping algorithm contains many complex optimization factors that make its quality difficult to evaluate. Since there were no benchmarks for nearest neighbor dominated parallel arrays a set of metrics had to be developed in order to evaluate mappings. Each metric, with the exception of the runtime metric, quantifies a desirable attribute for the mapping. The closer each metric is to zero the more optimal and more desirable the mapping. A collection of applications were created to produce the diverse mappings needed to properly evaluate the mapping algorithm. These applications contain common dataflows patterns and vary in difficulty and problem size to help stress the mapping algorithm. In Chapter 6 these metrics and applications are used to evaluate the quality and runtime for the mapping algorithm.

Chapter 6

Results

This chapter analyzes the quality and run-time of the mapping algorithm using the various metrics and applications discussed in Chapter 5. Each of these results demonstrate a different aspect of the mapping algorithm. These aspects include the efficiency of automatic mappings to hand mappings, scalability over growing problem sizes, overcoming faults in the physical device, and optimizing for frequency and leakage differences due to fabrication. The analyses in this chapter focus on the mapping quality for popular DSP applications and tasks. By focusing on DSP applications in the general sense, the mapping algorithm avoids becoming application specific.

6.1 Procedure

The following procedure was used to obtain the results in the remainder of this chapter. Derivations from this procedure are noted where they apply. First the application is prepared by constructing its dataflow graph inside the graphical user interface using the basic building blocks. Next the input and output configuration parameters are set to be compatible with the first version of AsAP, where the input processor is fixed at location $(0, 0)$ and the output processor is aligned with the right edge. Next the static parameters, those different from the default values listed in Table 6.1, are appended to the batch mode command-line. Next the dynamic parameters are generated (using scripts) and appended to the batch mode command-line. Dynamic parameters are derived from the trial number, which is typically used directly as the random seed. The parameters set on the batch mode command-line are identical to those used by the mapping algorithm. The purpose of each parameter is explained in Chapter 3 and the Glossary. Finally jobs are spawned across multiple machines each working on a subset of the solution space. These steps have been summarized in the

Parameter	Default Value
QuickPlace	False
UseRouting	True
AddSpacing	True
ExpandType	Vertical
NumIters	3
MaxRoutes	2
SpaceThreshold	110

Table 6.1: Relevant default batch mode configuration parameters, *see Glossary*.

following list.

1. Prepare the application dataflow graph
2. Set the configuration parameters for the input and output
3. Determine and set the static configuration parameters
4. Generate dynamic configuration parameters from the trial number
5. Execute the mapping algorithm using a batch mode script
6. Analyze the results from the mapping algorithm

6.2 Efficiency

For some problems there is a finite solution space that is simply searched more efficiently using heuristics, such as an iterative improvement algorithm. For the mapping problem addressed in this work the solution space would be finite if the array dimensions had an upper limit. This would mean that through permutations of every node at every location all possible configurations could be explored in some finite amount of time. Therefore the optimal configuration could eventually be determined. However, even with a finite array size the runtime would be extremely prohibitive. The array size though has no upper limit, so it can be difficult to determine the optimal configuration for some large applications.

Since the solution space is infinite, the optimal configuration has to either be determined theoretically or approximated by hand mappings. The reason we compare with hand mappings is that the programmer has an intuition about how the application was developed. These intuitions are, knowledge about any dataflow patterns and any underlying mathematical optimizations. Hand

mappings are almost always desirable but sometimes they can be very time consuming. It's important to also analyze how the mapping tool can aid the programmer when mapping both large and small applications.

This section will compare hand mappings to automatic mappings for a number of the applications listed in Chapter 5. For each comparison the two most important factors are the mapping quality and the time required to obtain the mapping. When comparing mapping qualities the three values of interest are: 1) the quality of the hand mapping, 2) the quality of the best automatic mapping, and 3) the best theoretical mapping (optimization cost equal to zero). When comparing mapping times the two values of interest are: 1) the estimated time for preparing the hand mapping, and 2) the sequential time necessary to obtain the automatic mapping used for quality comparisons. For these mappings 1000 trials were executed (random seeds 1 through 1000).

The last three applications tested in this section (Fast Fourier Transform, Small Clos Network, and Large Clos Network) can't be mapped using only nearest neighbor communication without the use of routing processors. For the second version of AsAP some of these routing processors can be converted into long-distance interconnects. This reduces the rectangular array area by allocating fewer processors for pure routing. Instead this data is passed through the switched routing overlay network. Though sometimes routing processors can be desirable since they can be programmed to perform tasks such as buffering (if this does not effect the area by too much). In this section we will compare automatic mappings targeting the second version of AsAP to automatic mappings targeting the first version of AsAP for these last three applications. The values of interest are the reduction in rectangular array area and the number of routing processors removed with respect to any increase in the number of long-distance interconnects.

6.2.1 802.11a Wireless Transmitter

Settings: Defaults + "UseRouting = False"

The 802.11a wireless transmitter is relatively simple to map to AsAP and requires no routing processors so the router insertion flag has been disabled, which saves time. The two mappings in Figure 6.1 show that the rectangular array area for both mappings are equal ($6 \times 4 = 6 \times 4$), the enclosed array area is slightly larger for the automatic mapping ($23 < 24$), both have no long-distance interconnects, and both have no routing processors. In fact both mappings look very similar. The optimal rectangular array area for this application is 25 (5×5). This means that the automatic mapping has an optimization cost of zero, which is considered optimal by the metrics used in this

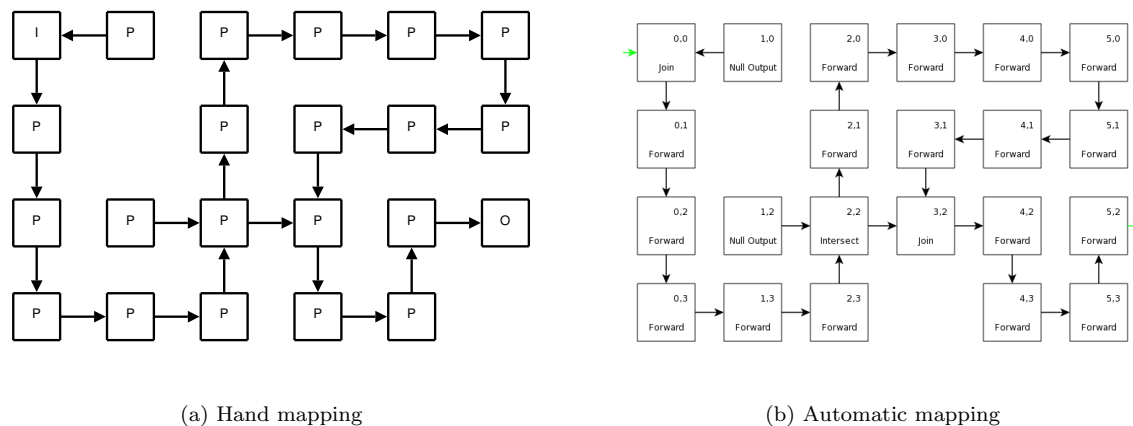


Figure 6.1: Side-by-side comparison of the hand mapping and the automatic mapping for the 802.11a wireless transmitter shown in Figure 5.3 on page 85

work. Figure 6.2 shows that the automatic mapping took 1 minute and 9 seconds to obtain. The hand mapping only took around 5 minutes since the application had been previously mapped by hand. The mapping tool is very fast for this small application. For this application the automatic mapping is just as good as the hand mapping. This application requires no routing processors so there would be no benefit to mapping this application using parameters for the second version of AsAP since the mapping would be identical.

6.2.2 Viterbi Decoder

Settings: Defaults + “UseRouting = False”

The Viterbi decoder is also relatively simple to map to AsAP, and again requires no routing processors, but the small loops in the traceback phase make this application a little more difficult to map. Since we know that the application can be mapped without routers we again disable router insertion to save time. Routing processors are of course not used in both mappings, but the two mappings in Figure 6.3 show that the hand mapping has a little bit smaller rectangular array area ($6 \times 6 < 8 \times 5$), the enclosed array areas are both equal ($30 = 30$), and that both mappings have no long-distance interconnects. The optimal rectangular array area for this application is 30 (5×6). Neither mapping is considered optimal by the metrics used in this work, but the hand mapping is slightly closer. Figure 6.4 shows that the automatic mapping took 18 minutes and 13 seconds to obtain. The hand mapping took around 10 minutes since the application had already been mapped. For this application the automatic mapping is very close to the hand mapping and even equal to the

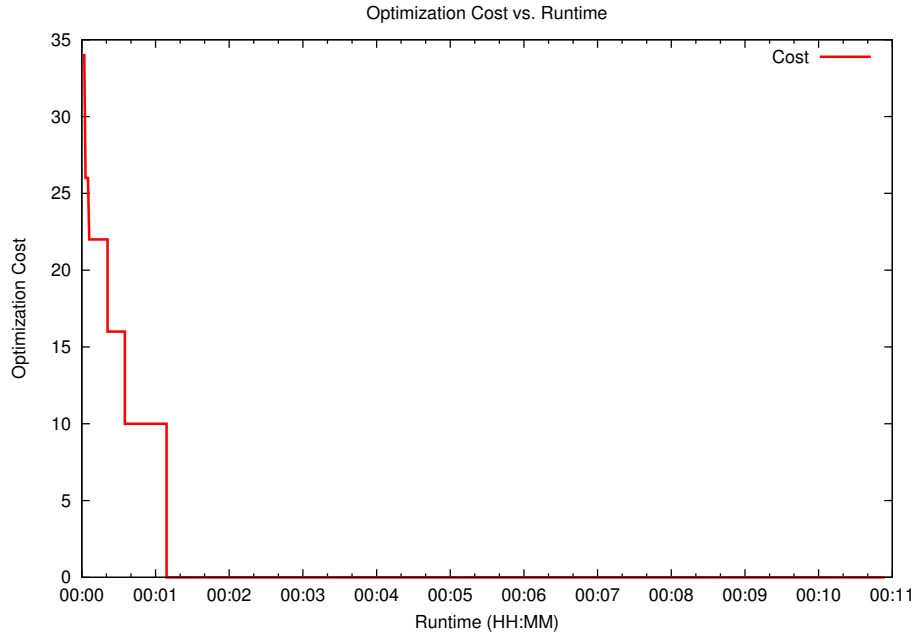


Figure 6.2: Reduction in optimization cost over time for the 802.11a wireless transmitter using 1000 trials

hand mapping in terms of enclosed array area. Again this application requires no routing processors so there would be no benefit to mapping this application using parameters for the second version of AsAP since the mapping would be identical.

6.2.3 Fast Fourier Transform

The Fast Fourier Transform application is more complex than the previous two applications making the mapping impossible without routers. The two mappings in Figure 6.5 show that the rectangular array area for the hand mapping is smaller ($7 \times 7 < 8 \times 9$), the enclosed array area for the hand mapping is smaller ($43 < 64$), and that both mappings have no long-distance interconnects. Counting the number of routing processors, the hand mapping requires only 23 routing processors, which is less than the 43 routing processors required by the automatic mapping. The optimal rectangular array area for this application is 20 (4×5). This is far below the area of the two mappings but this is expected since the mapping requires routers. As seen in Figure 6.6 the time required to obtain the automatic mapping was 6 minutes and 44 seconds. This is not the mapping with the lowest optimization cost because we prefer a mapping with no long-distance interconnects. The hand mapping took around 1 hour. For this application the automatic mapping was somewhat larger than the hand mapping but the automatic mapping was much quicker. The user could modify the

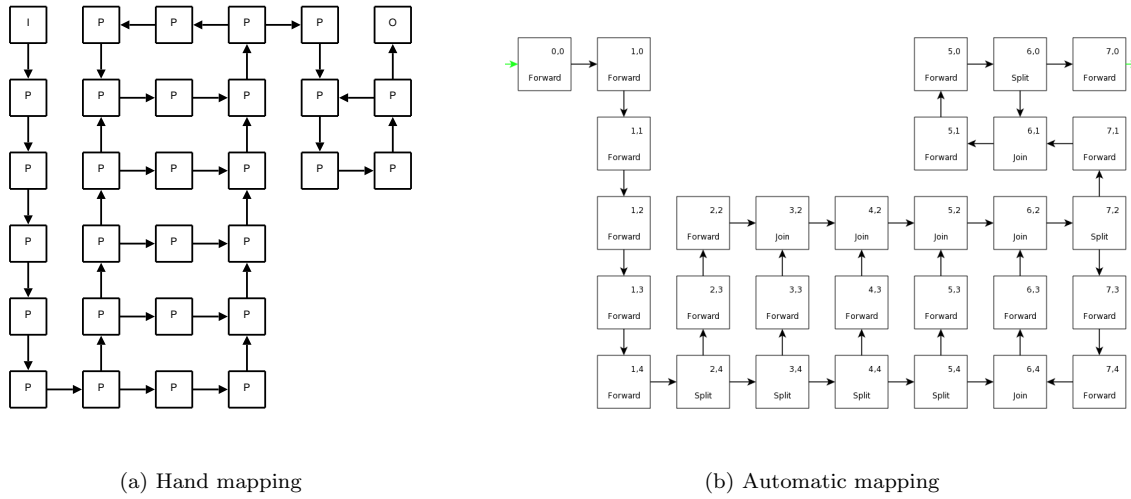


Figure 6.3: Side-by-side comparison of the hand mapping and the automatic mapping for the Viterbi decoder shown in Figure 5.4 on page 86

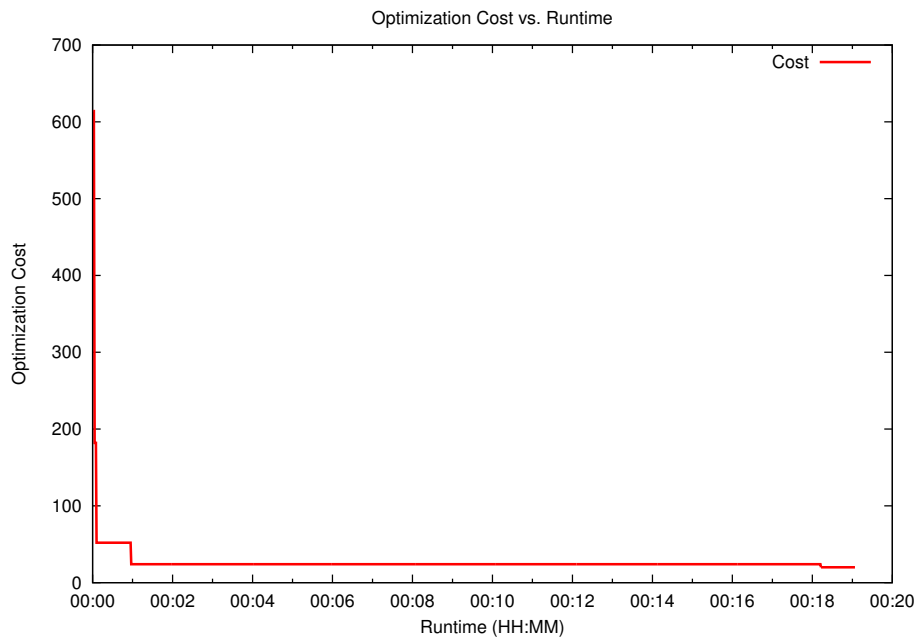


Figure 6.4: Reduction in optimization cost over time for the Viterbi decoder using 1000 trials

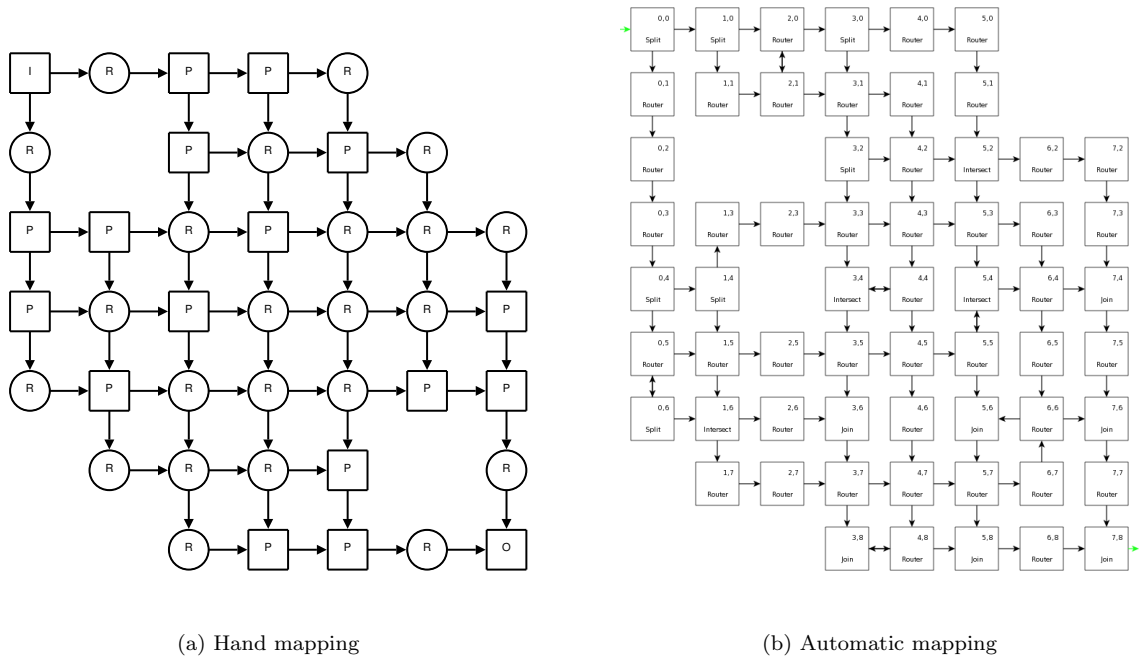


Figure 6.5: Side-by-side comparison of the hand mapping and the automatic mapping for the Fast Fourier Transform shown in Figure 5.5 on page 87

automatic mapping and save time over a fully manual mapping.

AsAP Version 2.0

Settings: Defaults + “AddSpacing = False” + “CostArraySize = 2X”

In order to target the second version of AsAP the spacing insertion flag is disabled and the cost for increasing the area during the placement phase is doubled. This still allows routing processors to be inserted but will produce a much tighter mapping, decreasing the rectangular array area. The automatic mapping in Figure 6.7 shows that the rectangular array area decreased by quite a bit ($4 \times 5 < 8 \times 9$) and also that the enclosed array area decreased by quite a bit ($20 < 64$) from the previous automatic mapping. Comparing routing processors, this new mapping uses only 2 routing processors, which is far less than the previous 43 routing processors required. This decrease in rectangular array area and routing processors only added 7 long-distance interconnects, all of which are easily routable using the routing overlay network. This new mapping targeting the second version of AsAP is much better than the previous mapping targeting the first version of AsAP because the array area is significantly less.

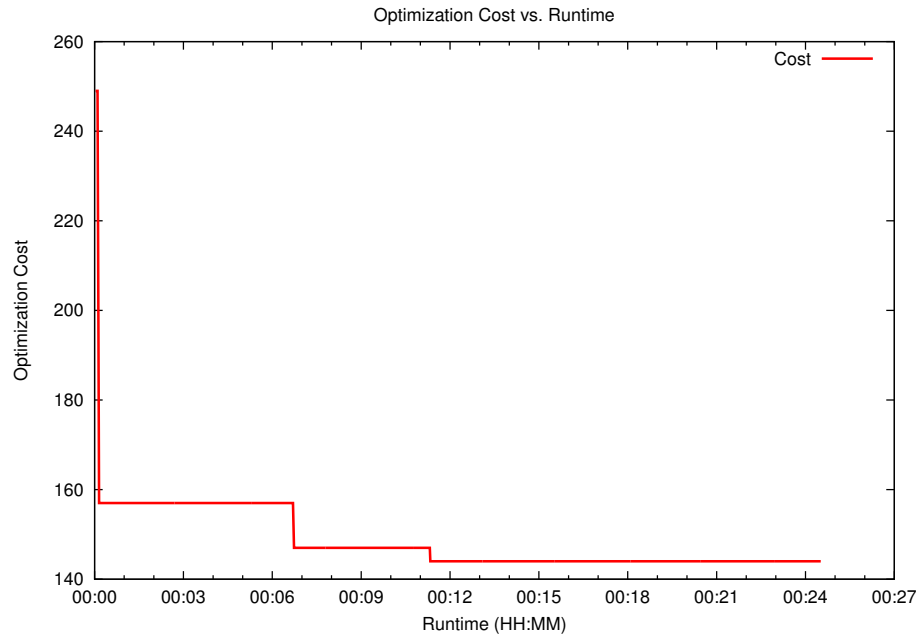


Figure 6.6: Reduction in optimization cost over time for the Fast Fourier Transform using 1000 trials

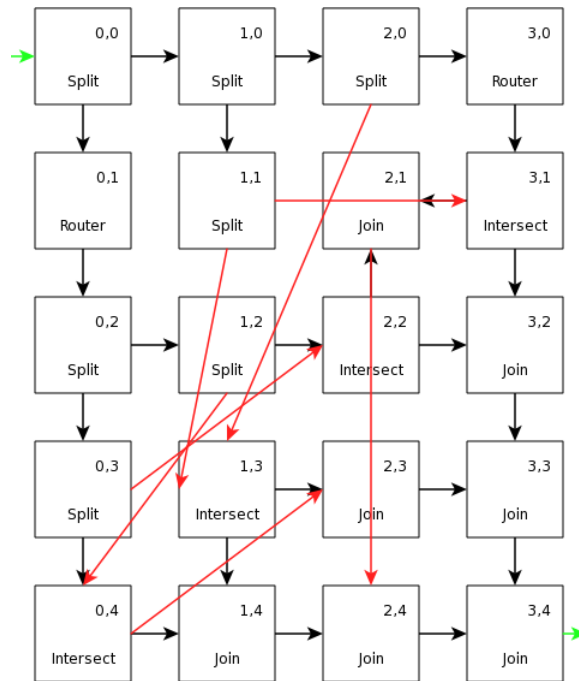


Figure 6.7: Automatic mapping for the Fast Fourier Transform shown in Figure 5.5 on page 87 when targeting the second version of AsAP

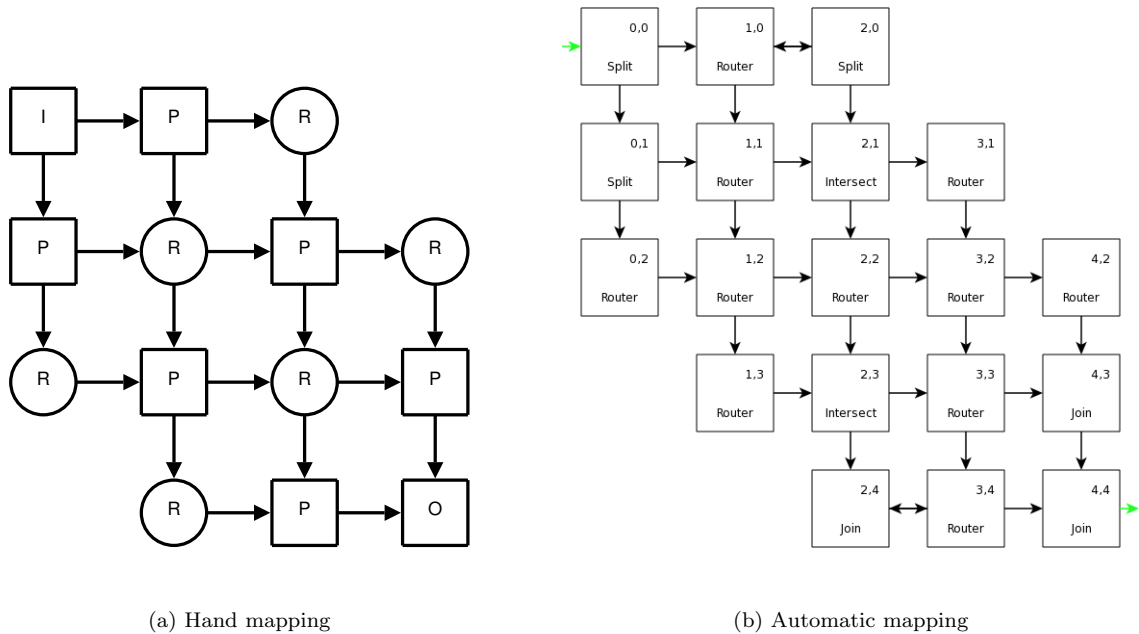


Figure 6.8: Side-by-side comparison of the hand mapping and the automatic mapping for the small Clos network shown in Figure 5.6 on page 88

6.2.4 Small Clos Network

The small Clos network is a relatively small application but each node has a high input and output degree. This becomes a problem because not every node can be configured to use nearest neighbor communication only. This creates a large number of intersecting routes that must be negotiated using routers. Figure 6.8 shows that both mappings require quite a few routing processors. The hand mapping requires 6 routing processors and the automatic mapping requires 11 routing processors. Both mappings have no long-distance interconnects. Comparing areas, the hand mapping has a rectangular array area of 16, which is smaller than the automatic mapping that has a rectangular array area of 25 ($4 \times 4 < 5 \times 5$). The hand mapping also has a smaller enclosed array area than the automatic mapping ($14 < 19$). The optimal rectangular array area is 9 (3×3) for this application. The high optimization cost for this application indicates that this application is a poor fit for the AsAP architecture but nevertheless still an interesting application. As seen in Figure 6.9 the time required to obtain the automatic mapping was 5 minutes and 26 seconds. The time required to perform the hand mapping was about 30 minutes. The hand mapping is a little smaller than the automatic mapping as expected, but this application is very small so a decent solution can be reached in a short period of time and improved manually from there.

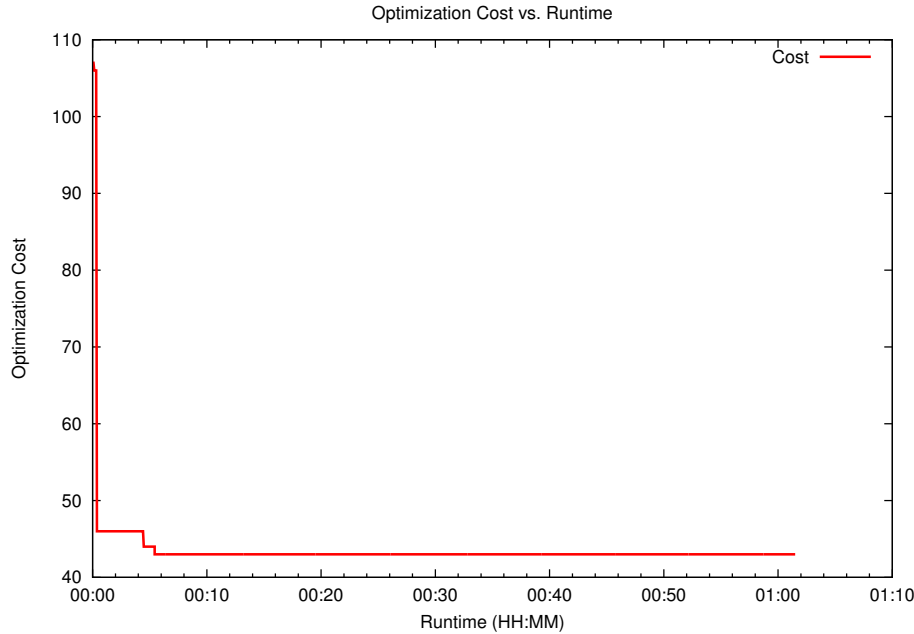


Figure 6.9: Reduction in optimization cost over time for the small Clos network using 1000 trials

AsAP Version 2.0

Settings: Defaults + “AddSpacing = False” + “CostArraySize = 2X”

Like before space insertion is disabled and the area cost during the placement phase is doubled to target the second version of AsAP. The automatic mapping in Figure 6.10 shows that the new mapping has a smaller rectangular array area ($4 \times 4 < 5 \times 5$) and a smaller enclosed array area ($14 < 19$) than the previous automatic mapping. Not only did the area decrease but the new automatic mapping is exactly the same as the hand mapping. This happened because space was inserted in the previous automatic mapping, due to the high edge to node ratio, when it was not needed. This is an example of how tuning the configuration parameters can improve the mapping quality for certain applications. This new mapping is not only better for the second version of AsAP it’s also better for the first version of AsAP since it has a lower rectangular array area, uses fewer routing processors, and still has no long-distance interconnects.

6.2.5 Large Clos Network

The large Clos network is larger than the previously tested applications. This application has a large number of intersecting routes that must all be dealt with by routers. For this reason the areas of both the hand mapping and the automatic mapping will be far from the optimal rectangular

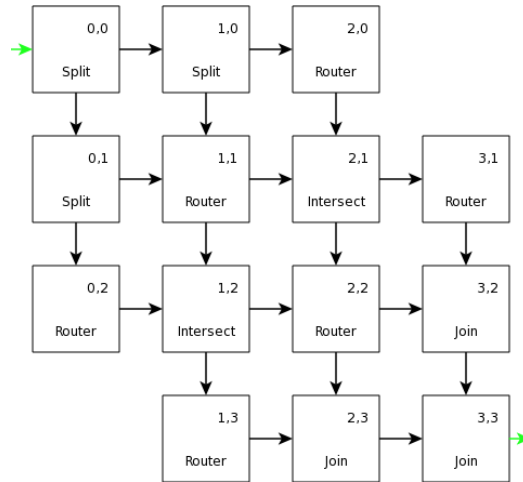


Figure 6.10: Automatic mapping for the small Clos network shown in Figure 5.6 on page 88 when targeting the second version of AsAP

array area, which is 20 (4×5). The two mappings in Figure 6.11 show that both mappings have no long-distance interconnects and that the hand mapping has a rectangular array area of 100, which is less than the automatic mapping that has a rectangular array area of 132 ($10 \times 10 < 12 \times 11$). The enclosed array area is also smaller for the hand mapping ($70 < 109$). The areas for these two mappings may not be extremely close but they are within the same ballpark, which suggests that the mapping algorithm handles this application decently even though this application is not well suited for AsAP. Comparing the number of routing processors, the hand mapping has 42 routing processors, which is quite a bit less than the automatic mapping that has 73 routing processors. The hand mapping took only 15 minutes because it had been previously mapped by another individual. The original author estimates that the hand mapping took him somewhat less than an hour and in addition to his mapping I made a number of less successful attempts. As seen in Figure 6.12, the automatic mapping took 24 minutes and 7 seconds to obtain. The automatic mapping was faster for this application but the mapping still contains some trivial improvements.

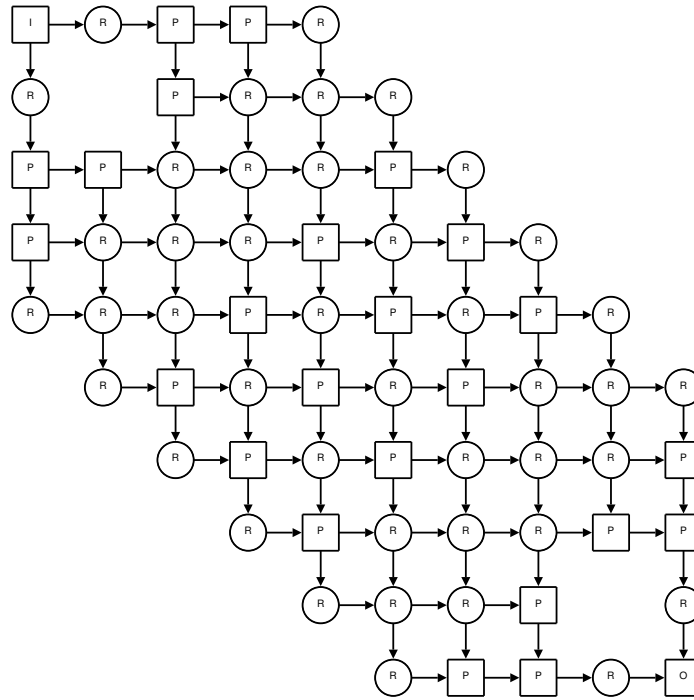
The first two implementations of the mapping algorithm were unable to produce a valid mapping for the first version of AsAP when mapping the large Clos network. The mapping was compacted more than necessary so the remaining intersecting edges were left unrouted due to routing conflicts. My first attempt at solving this problem was to detect and remove intersecting edges, but this resulted in a large number of parallel edges that caused even more routing conflicts. My second attempt at solving this problem was to insert additional rows and columns of empty space so intersections were handled by additional routing processors. This solved the problem but it severely degraded the mapping quality for other applications. The solution was to add a threshold value

to determine when to enable this feature. However, this threshold value was unreliable for some applications, like the small Clos network, so space insertion had to be disabled manually for these applications.

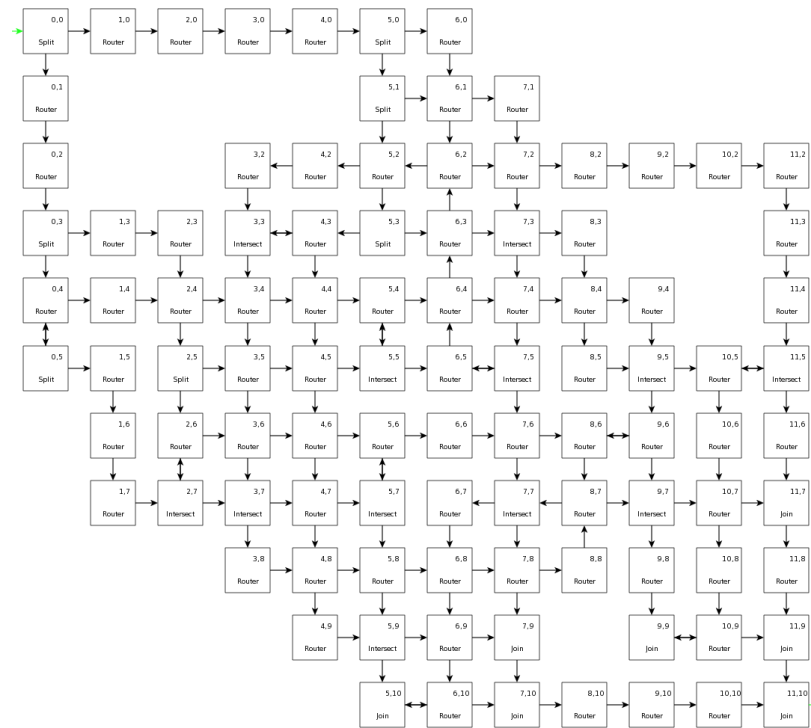
AsAP Version 2.0

Settings: Defaults + “AddSpacing = False” + “CostArraySize = 2X”

To target the second version of AsAP the spacing insertion flag has again been disabled and the cost for increasing the area has been doubled. The automatic mapping in Figure 6.13 shows that the rectangular array area has decreased quite a bit ($6 \times 5 < 12 \times 11$) and that the enclosed array area has also decreased quite a bit ($29 < 109$) from the previous automatic mapping. Comparing the number of routing processors, the new mapping has 3 routing processors while the previous mapping had 73 routing processors. This substantial decrease in rectangular array area and routing processors comes with a penalty of 16 long-distance interconnects. These long-distance interconnects can be negotiated using the routing overlay network, but configuring these long-distance connections is not trivial. For the second version of AsAP this new mapping is much better than the previous mapping due to the significant decrease in array area.



(a) Hand mapping



(b) Automatic mapping

Figure 6.11: Side-by-side comparison of the hand mapping and the automatic mapping for the large Clos network shown in Figure 5.7 on page 88

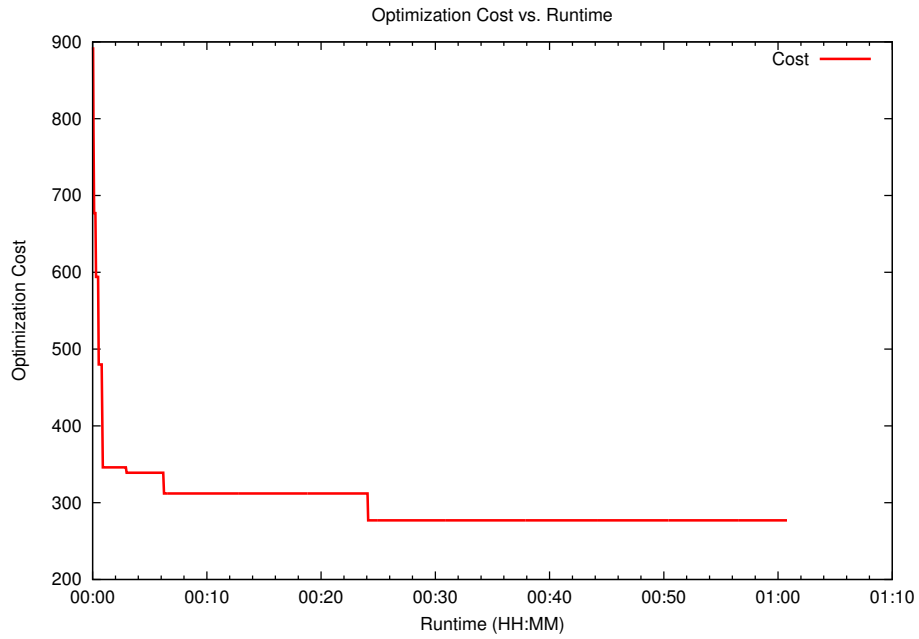


Figure 6.12: Reduction in optimization cost over time for the large Clos network using 1000 trials

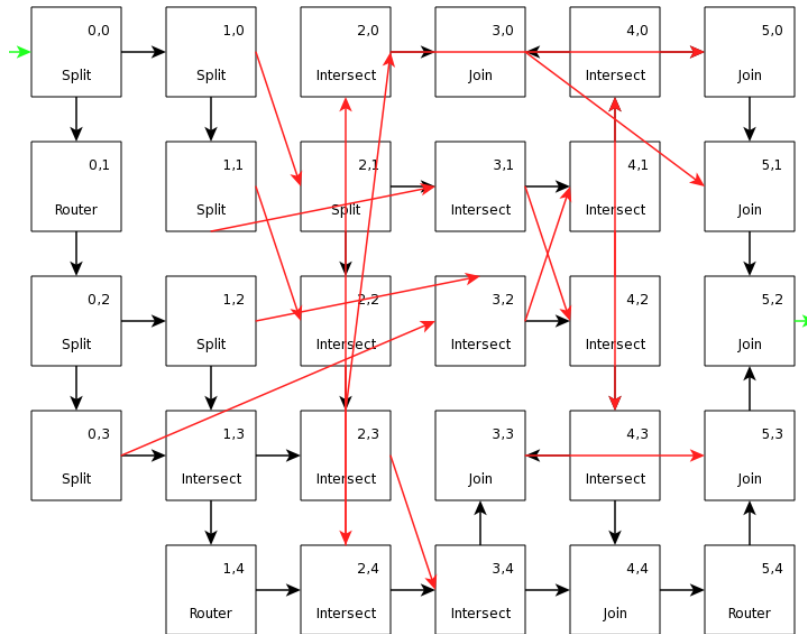


Figure 6.13: Automatic mapping for the large Clos network shown in Figure 5.7 on page 88 when targeting the second version of ASAP

Application	Hand Mapping (Rect. Area)	Auto Mapping (Rect. Area)	Auto Mapping Area Increase
Wireless	24 (6x4)	24 (6x4)	0%
Viterbi	36 (6x6)	40 (8x5)	11.1%
Fourier	49 (7x7)	72 (8x9)	46.9%
Small Clos	16 (4x4)	25 (5x5)	56.3%
Large Clos	100 (10x10)	132 (12x11)	32.0%

Table 6.2: The rectangular array area for the hand mappings and the automatic mappings, along with the percentage overhead, when targeting the first version of AsAP

6.2.6 Summary

What the mappings in this section show is that automatic mappings are often viable alternatives to hand mappings and sometimes even equal to hand mappings. Table 6.2 compares the rectangular array area, when targeting the first version of AsAP, and shows that the best mapping was the 802.11a wireless transmitter with no overhead and that the worst mapping was the small Clos network with an overhead of approximately 56%. Table 6.3 compares the enclosed array area, when targeting the first version of AsAP, and shows that the best mapping was the Viterbi decoder with no overhead, and that the worst mapping was the large Clos network with an overhead of approximately 56%. The percentage overhead is quite low for the first two applications, which are well suited to the first version AsAP. However, the overhead is more than 30% for the last three applications. This is because the last three applications have a higher routing complexity than the first two applications and require routing processors which typically bloat the array. Table 6.4 compares the rectangular array area, when targeting the second version of AsAP, and shows that the most improved mapping was the large Clos network with a decrease of 70% and that the least improved mapping was the small Clos network with no decrease at all. Table 6.5 compares the enclosed array area, when targeting the second version of AsAP, and shows that the most improved mapping was again the large Clos network with a decrease of approximately 59% and that the least improved mapping was again the small Clos network with no decrease at all. The percentage decrease is very high for two of these applications. This is because these two applications make extensive use of routing processors. The bloat typically associated with inserting routing processors is removed when routing processors are converted into long-distance interconnects.

For most of these applications the time required to obtain a decent mapping is much lower when mapped automatically than when mapped entirely by hand. These automatic mappings can sometimes be further improved by the user. Therefore quickly achieving a decent mapping will save the user a great deal of time. This is especially true for very large applications. Automatic

Application	Hand Mapping (Encl. Area)	Auto Mapping (Encl. Area)	Auto Mapping Area Increase
Wireless	23	24	4.3%
Viterbi	30	30	0%
Fourier	43	64	48.8%
Small Clos	14	19	35.7%
Large Clos	70	109	55.7%

Table 6.3: The enclosed array area for the hand mappings and the automatic mappings, along with the percentage overhead, when targeting the first version of AsAP

Application	Hand Mapping (Rect. Area)	Auto Mapping (Rect. Area)	Auto Mapping Area Decrease
Fourier	49 (7x7)	20 (4x5)	59.2%
Small Clos	16 (4x4)	16 (4x4)	00.0%
Large Clos	100 (10x10)	30 (6x5)	70.0%

Table 6.4: The rectangular array area for the hand mappings and the automatic mappings, along with the percentage savings, when targeting the second version of AsAP

mappings could also provide some guidance when mapping applications by hand. When mapping applications to the second version of AsAP the mapping algorithm does an excellent job of reducing the rectangular array area and the number of routing processors without using too many long-distance interconnects. All these tests show that the mapping algorithm can be an efficient tool when mapping various types of applications.

6.3 Scalability

As VLSI technology continues to improve the number of processing elements on a single die continues to increase. This means that as time progresses there will be more processing elements on a single die that need to be programmed. This in-turn results in larger applications or many smaller applications sharing a single chip. An important goal for this work is to develop a mapping algorithm that is scalable and able to map larger applications even when these applications can not be realized on current physical designs.

If the mapping algorithm was ideal the runtime would grow linearly with respect to the

Application	Hand Mapping (Encl. Area)	Auto Mapping (Encl. Area)	Auto Mapping Area Decrease
Fourier	43	20	53.5%
Small Clos	14	14	00.0%
Large Clos	70	29	58.6%

Table 6.5: The enclosed array area for the hand mappings and the automatic mappings, along with the percentage savings, when targeting the second version of AsAP

problem size. If this growth was instead exponential the runtime would be prohibitive for large applications. Also, if the mapping algorithm was ideal the mapping quality would have equal variance from the optimal mapping for all problem sizes. In summary, the algorithm should perform enough work on each node that the runtime remains linear and the quality remains relatively constant for any problem size. Estimating the runtime for the mapping algorithm using Big-O notation, or $\mathcal{O}(N)$ (which approximates an algorithm's runtime based on its input data size), is a rather complex process since many decisions in the algorithm are made based upon some random value. Therefore the algorithm is simulated using randomly generated applications with problem sizes of 100, 250, 500, and 1000, in order to get an estimate of the algorithms scalability. Each application is simulated using the same settings and 100 trials are executed (random seeds from 1 to 100) on each application.

6.3.1 100 Random Nodes

The 100 random nodes application is the smallest of the four applications tested for scalability so it will serve as a baseline for the other applications. To get an idea of the runtime of the algorithm we choose the mapping with the lowest optimization cost and noted the time required to obtain this mapping. As seen in Figure 6.14 the time required to obtain the mapping with the lowest optimization cost was about 2 hours and 5 minutes. At this point the rectangular array area is 399 (21x19), 143 routing processors are used, and there are zero long-distance interconnects. For 100 processors the optimal rectangular array area is of course 100 (10x10). In relative terms we have about a 4x increase in rectangular array area, and about 1.5x the number of routing processors as the minimum number of processors. These relative numbers will be used for comparisons with the remaining tests.

6.3.2 250 Random Nodes

For the 250 random nodes application we would ideally expect the runtime to be around 2.5x longer than the 100 random nodes application and have the same relative quality. Trends can not yet be formed without a third data point. This time we chose a mapping that does not have the lowest optimization cost since the mapping with the lowest optimization cost uses long-distance interconnects. As seen in Figure 6.15 the mapping chosen required about 22 hours and 1 minute to obtain. This mapping has a rectangular array area of 1224 (36x34), uses 550 routing processors, and has no long-distance interconnects as stated before. Comparing runtimes for the 100 random nodes application and the 250 random nodes application the increase is about 11x, which is more than

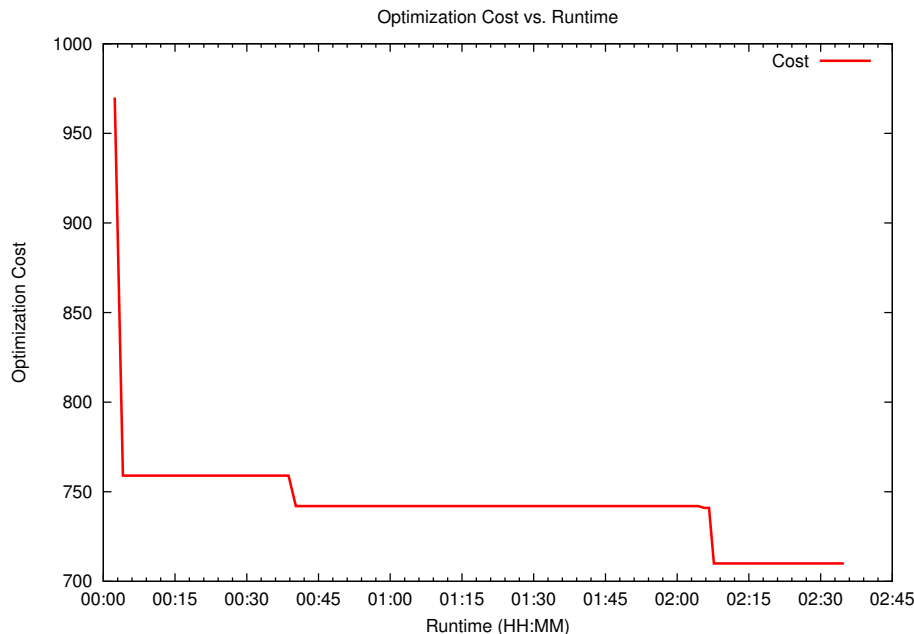


Figure 6.14: Reduction in optimization cost over time for the 100 random node application using 100 trials

the ideal 2.5x increase. The optimal rectangular array area is 256 (16x16), which gives a relative increase in rectangular array area of about 4.75x and about 2x the number of routing processors as the minimum number of processors. The runtime is higher than expected but the quality metrics are close to those of the 100 random nodes application.

6.3.3 500 Random Nodes

For the 500 random nodes application we would ideally like the runtime to be around 5x longer than the 100 random nodes application with the same relative quality. Based on runtime results from the previous tests, we can expect the runtime to increase by about twice as much for this test, or around 22x. Once again the point chosen was not the mapping with the lowest optimization cost since the mapping with the lowest optimization cost has long-distance interconnects. As seen in Figure 6.16 this mapping was obtained in around 109 hours and 8 minutes. This mapping has a rectangular array area of 2548 (49x52), uses 1302 routing processors, and has no long-distance interconnects. Comparing the runtime of this application to the runtime of the 100 random nodes application the increase is about 55x, about twice what we expected. Comparing quality, the optimal rectangular array area is 506 (22x23), which gives a relative increase in the rectangular array area of about 5x. The number of routing processors is about 2.5x the minimum number of processors.

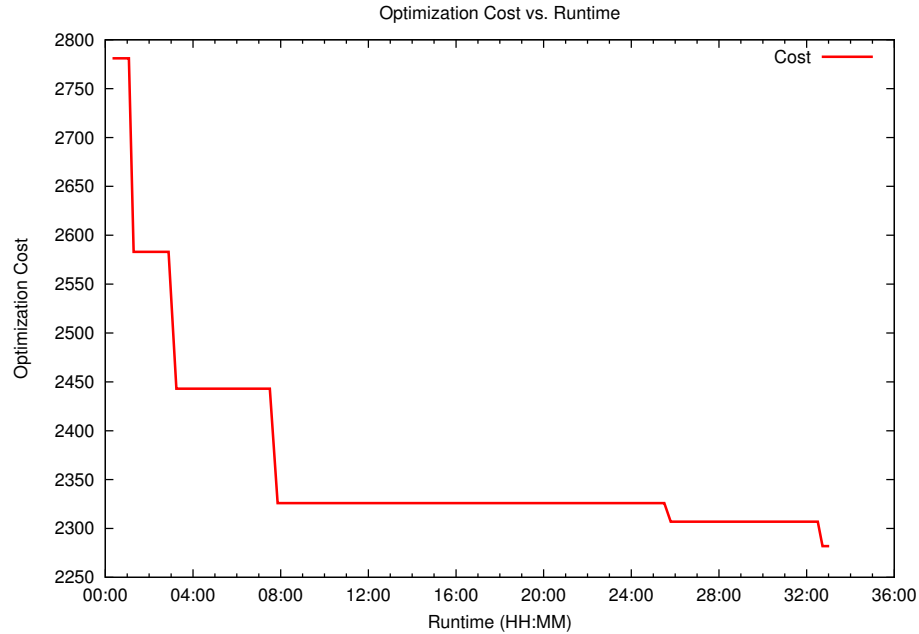


Figure 6.15: Reduction in optimization cost over time for the 250 random node application using 100 trials

For this application the runtime is higher than expected but the quality metrics are quite close.

6.3.4 1000 Random Nodes

For the 1000 random nodes application we would ideally like the runtime to be around 10x longer than the 100 random nodes application with similar quality metrics. Based on the previous tests we expect the runtime to be around 110x. For comparisons the mapping selected is again the mapping with the lowest optimization cost. The selected mapping (and all the other mappings) doesn't have zero long-distance interconnects so the mapping is invalid for the first version of AsAP. Therefore, the values obtained from this test will serve as lower bounds instead of actual data points. As seen in Figure 6.17 the selected mapping was obtained in around 163 hours and 52 minutes. It's important to also notice that the final mapping for this test, obtained in around 393 hours and 28 minutes, is also invalid. This indicates that more time is needed to find a valid mapping and that the final mapping is the best choice for the lower bounds regarding runtime. For this mapping the rectangular array area was 5112 (71x72), 2814 routing processors were inserted, and there were 17 long-distance interconnects that could not be removed. For the mapping to be valid these 17 long-distance interconnects would have to be converted into routing processors. This would not only increase the number of routing processors but also increase the rectangular array area. Therefore

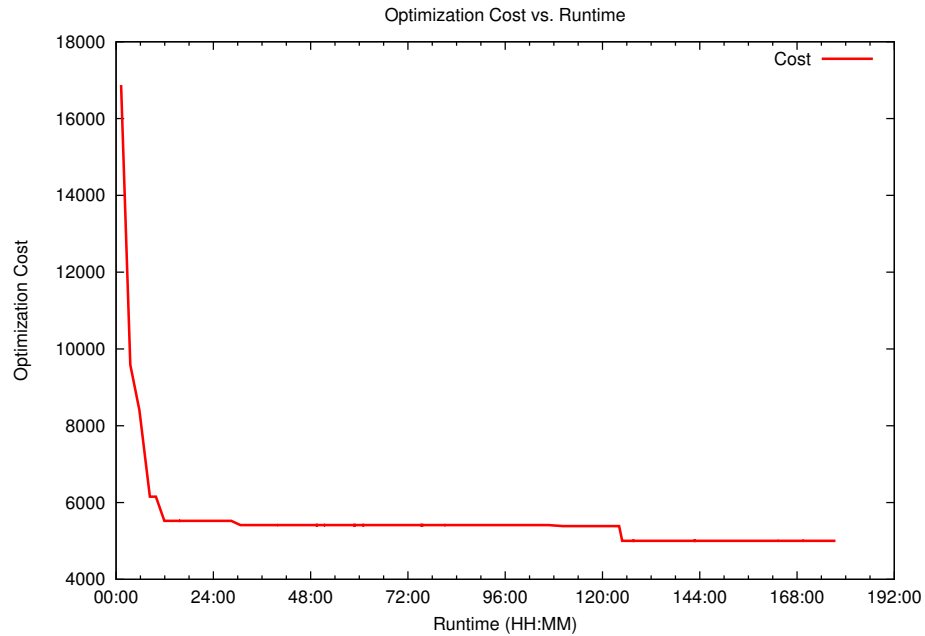


Figure 6.16: Reduction in optimization cost over time for the 500 random node application using 100 trials

the mapping with the lowest optimization cost is the best choice for the lower bounds with regards to metric quality. Comparing the runtimes between this application and the 100 random nodes application the increase is about 197x, again about twice what we expected. Comparing quality, the optimal rectangular array area is 1024 (32x32), which gives a relative increase in rectangular array area of about 5x. The number of routing processors is about 2.75x the minimum number of processors. For this application the runtime is again higher than expected but the quality metrics are close to previous values.

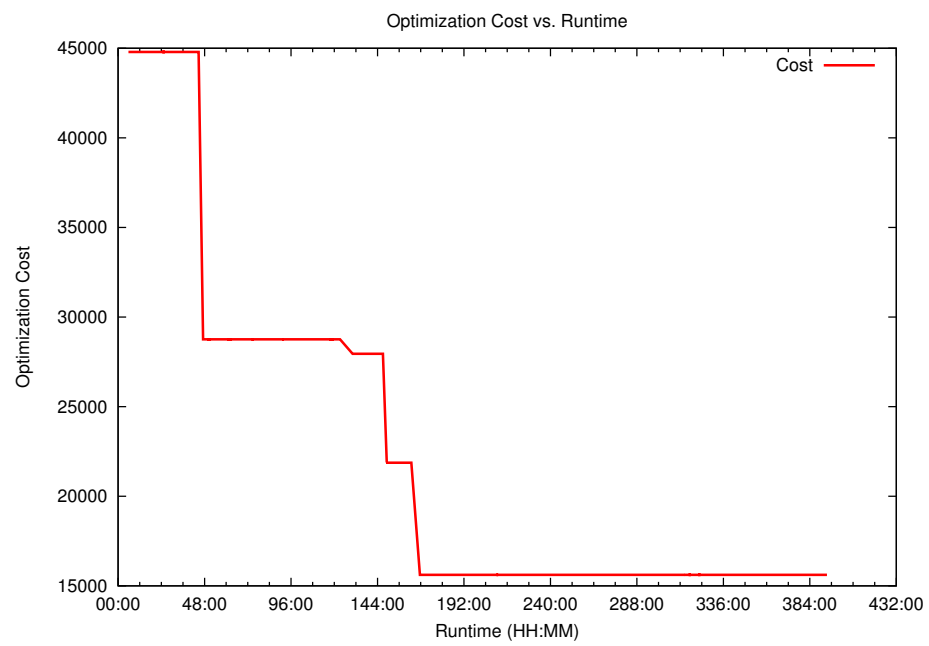


Figure 6.17: Reduction in optimization cost over time for the 1000 random node application using 100 trials

Problem Size	Runtime (Hours)	Increase in Rect Area (Relative Min Nodes)	Increase in Routers (Relative Min Nodes)
100	2.08	3.99 X	1.43 X
250	22.02	4.78 X	2.15 X
500	109.13	5.04 X	2.57 X
1000	393.47	4.99 X	2.75 X

Table 6.6: Runtime and increase in metric quality, relative to the minimum number of nodes, with respect to problem size for the random node applications.

6.3.5 Summary

The four tests performed in this section show that the mapping algorithm is scalable. As stated earlier, the runtime should not grow exponentially and the relative quality should not grow uncontrollably in order for the mapping algorithm to be considered scalable. The resulting data from these four tests is listed in Table 6.6. Figure 6.18 plots the runtime for each application with respect to problem size. The coefficient for the best-fit line shows an increase of about 27 minutes for each node added. If jobs were divided across 30 CPUs the increase would be less than 1 real world minute for each node added. Since the last data point is only a lower bound this leads to two possible outcomes. The best case scenario is that the true data point is just a few iterations away, resulting in a linear increase. The worst case scenario is that the increase is actually quadratic (somewhat indicated by the other data points) and therefore the true data point would be over 500 hours. Neither outcome results in an exponential increase, which would disprove scalability. Figure 6.19 plots the increase in rectangular array area and the increase in the number of routing processors relative to the minimum number of nodes with respect to problem size. Like before there are two possible outcomes based on these lower bounds. The best case scenario is that the true data points are on par with the lower bounds, resulting in near equal variance for the metric quality across all problem sizes. The worst case scenario is that the true data points are much higher than the lower bounds, resulting in a linear increase. Having equal variance is not required for scalability but is instead ideal. Based on the observed trends for the runtime, rectangular array area, and number of routing processors, the mapping algorithm has been shown to be scalable. To get more precise results more trials would have to be performed. This would provide a concrete data point for the 1000 random nodes application. Also testing intermediate problem sizes will increase the accuracy of the observed trends.

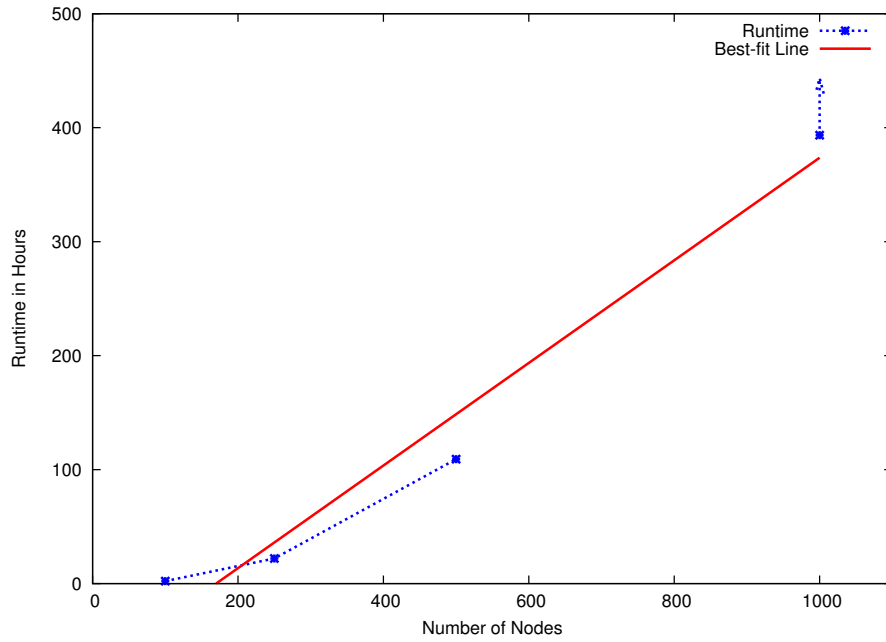


Figure 6.18: Plot of the application runtime with respect to problem size for the random nodes applications. The up arrow indicates that the point is a lower bound.

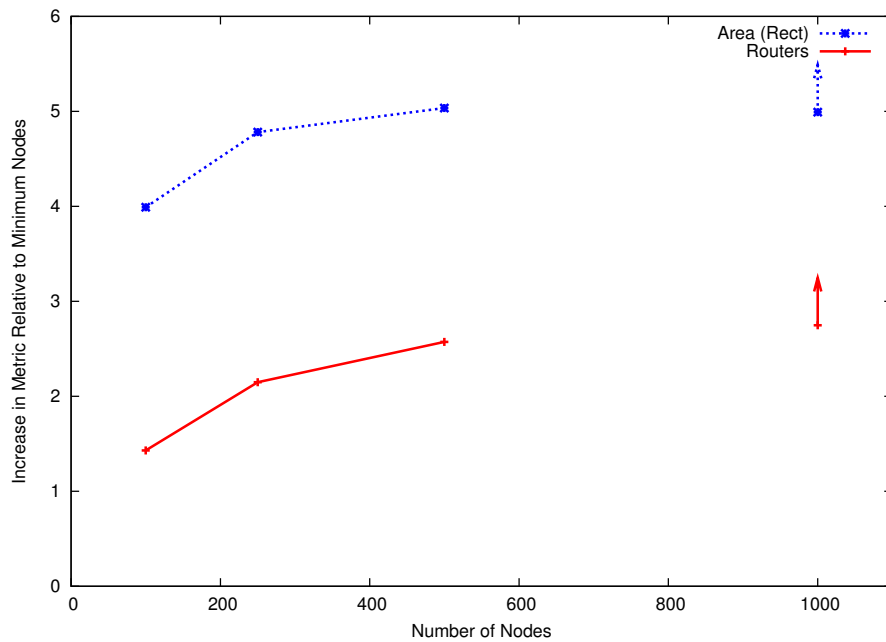


Figure 6.19: Plot of the increase in metric quality, relative to the minimum number of nodes, with respect to problem size for the random nodes applications. The up arrows indicate that these points are a lower bound.

6.4 Fault Tolerance

When fabricating a large number of chips some chips will inherently have flaws due to defects. These defects are expected so complex testing procedures have been developed to determine which of these chips contain flaws. Industry standards have been developed that estimate the yield expected for a given number of chips fabricated. This estimate is based on the size of the design, the size of the wafer, the technology used, and other factors. Chips found to have fabrication errors must be discarded so this is accounted for in the final cost of a design.

One goal of this work is to mitigate this yield problem. This is possible with AsAP because it has a large number of identical processing elements (also called a homogeneous array), that can be configured in many different ways to accomplish the same task. If one of these processing elements is determined to be bad it can be *excluded* and the mapping algorithm will choose another processor to take its place. An excluded processor is a processor within the target array that must remain unused in the final mapping. This means no task may be assigned to a location associated with an excluded processor. Every chip must still be tested, which can be difficult, but chips found to have defects can still be used if enough processing elements remain, thus mitigating the yield problem. The mapping algorithm can be considered tolerant to processor failures if the area, utilization, and communication metrics remain close to their original values (which is when no processors are excluded). However it's likely not possible for these metrics to remain close to their original values if the number of excluded processors is close to the number of nodes in the application. In this case we would like to know how many additional processors are needed for a given number of excluded processors.

This section first tests two larger applications and one smaller application to see if they can tolerate minor fabrication errors, or single processor failures, without increasing any of the metrics. The first large application is the multi-app application (page 117), which has 70 nodes. The second large application is the 100 random nodes application (page 134), from the previous section (6.3). The small application is the 802.11a wireless transmitter application (page 150), which has only 22 nodes and is very easy to map. The two larger applications are tested on both the first and second versions of AsAP using an array of size 25x25 and by executing 100 trials for each excluded processor. Comparisons are made between the two platforms for these tests. The smaller application is only tested on the first version of AsAP, but uses an array of size 8x8 and executes 1000 trials for each excluded processor to obtain higher resolution results. The minimum values for each metric from each of these tests are plotted in a color-coded 2D-array by excluded processor location, as a histogram, and as a cumulative distribution function. The best automated mapping for each

application is also shown, which includes one of the excluded processors.

This section next tests how the 802.11a wireless transmitter application (page 154) tolerates a large number of fabrication errors, or multiple processor failures. Excluded processors are chosen randomly from an array of size 10x10. In one test case there are actually more excluded processors than nodes in the application. Fabrication errors are simulated within the mapping tool using the excluded coordinate feature of the mapping algorithm. This feature prevents the tool from assigning tasks to processors that match an excluded coordinate. Tests are performed by excluding 100 different combinations of randomly selected processors, mapping the application 100 times (random seeds from 1 to 100), then analyzing the results. To analyze the results, the minimum values for each metric from each of these 100 trials are plotted along with the mean of these minimum values and the *base values*. Base values are obtained by performing 100 trials using no excluded processors then finding the minimum value for each metric from these 100 trials. Any mappings that utilize long-distance interconnects are ignored since these mappings are invalid for the first version of AsAP.

6.4.1 Multi-App Application

Settings: Defaults + “InputEdge = Left”

For this test each processor from an array of 25x25 is excluded one-by-one in sequence. For each excluded processor 100 trials are performed using unique random seeds. Base values are obtained by performing 100 trials with no excluded processors then finding the minimum value for each metric from these 100 trials. Minimum values for each metric are then obtained from every 100 trial block that includes an excluded processor. Next the mean and the median of these minimum values are computed. The array input is allowed to float along the left edge of the array so coordinate (0, 0) can be excluded even though the first version of AsAP doesn't support this. Also by making this adjustment to the array input, mappings are more likely to converge to a nearest neighbor only solution when performing these fault tolerance tests.

In Figure 6.20 the minimum rectangular array area for each excluded processor has been plotted as a 2D-array and each square has been labeled with its value. The lighter the color of the square the higher its value. A white square with a black X indicates that the application can not be mapped when the location in question is excluded. Similarly Figure 6.23 shows a 2D-array of the minimum number of routing processors for each excluded processor. In Figure 6.21 and Figure 6.22 the minimum rectangular array area has been plotted as a histogram and as a cumulative distribution function, respectively. The median, mean, and base values are provided as tick marks

across the top of the graph along with their exact values for comparisons. Similarly, Figure 6.24 and Figure 6.25 show the histogram and the cumulative distribution function for the minimum number of routing processors. Figure 6.26 shows the best automated mapping for the multi-app application when targeting the first version of AsAP, which happens to exclude processor (10, 1). Plots are not necessary for the minimum number of long-distance interconnects since this value must always be zero when targeting the first version of AsAP.

The 2D-array plots indicate that the application is more difficult to map when processors are excluded near the center of the array. The minimum rectangular array area increases when processors are excluded near the region between columns 11 and 13 and between rows 5 and 7. In some cases the application even becomes unmappable. This is not entirely unexpected since the application is very cluster oriented. The Viterbi decoder, which is part of the multi-app application, requires a large contiguous space for mapping the traceback cycle. The Fast Fourier Transform, also part of the multi-app application, is very demanding in terms of routing resources. Excluded processors not only disrupt the contiguity of the target array but they also interfere with the space insertion component. This leads to unwanted long-distance interconnects. The histograms for the minimum rectangular array area and the minimum number of routing processors show that the mean and base values for these two metrics are quite close. This indicates that the application is mostly unaffected by the excluded processors and is therefore tolerant of single processor exclusions. These histograms also show the base value being greater than both the mean and median values. This is quite unexpected! This unexpected occurrence is largely due in part to the size of the application and the number of trials that were performed. If a significantly higher number of trials were performed, or a more highly optimized algorithm was available, the entire graph would shift left and the base value would drop below the mean and median values. Later in this section (page 150) the 802.11a wireless transmitter application is mapped using 10 times the number of trials to compare how the number of trials affects the base, mean, and median values. It's important to notice that in the two cumulative distribution functions 1.0 is never reached because a few of the mappings were unsuccessful. The cumulative distribution functions indicate that by using a rectangular array area of 288 there is a 95% chance that the mapping will be successful with one excluded processor.

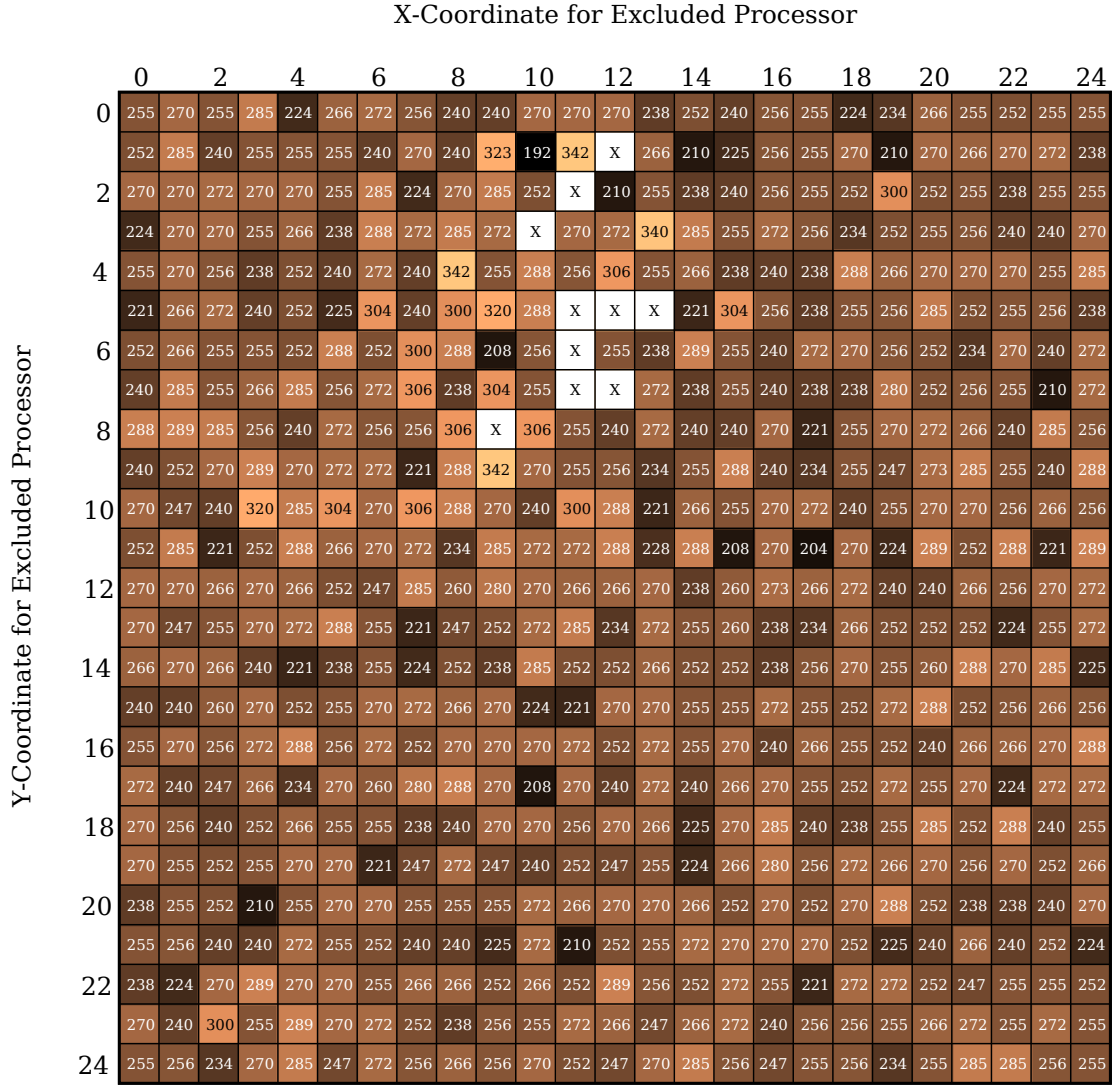


Figure 6.20: 2D-plot of the minimum rectangular array area when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the multi-app application targeting the first version of ASAP. The square with the darkest color has the lowest rectangular array area. The rectangular array area increases as the square lightens in color. A white square with a black X indicates an unmappable location. The statistics for this test regarding the minimum rectangular array area are: minimum = 192; maximum = 342; mean = 259.6; median = 256; base = 270.

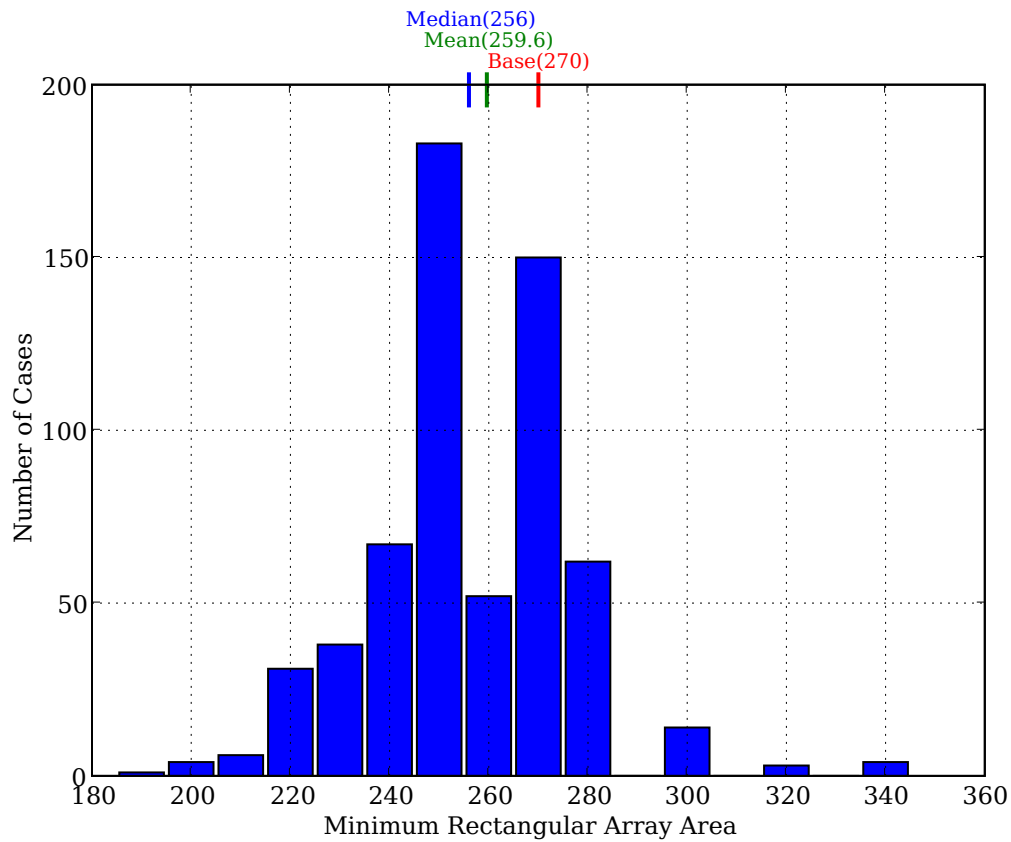


Figure 6.21: Histogram of the minimum rectangular array area, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the multi-app application targeting the first version of ASAP. The statistics for this test regarding the minimum rectangular array area are: minimum = 192; maximum = 342; mean = 259.6; median = 256; base = 270.

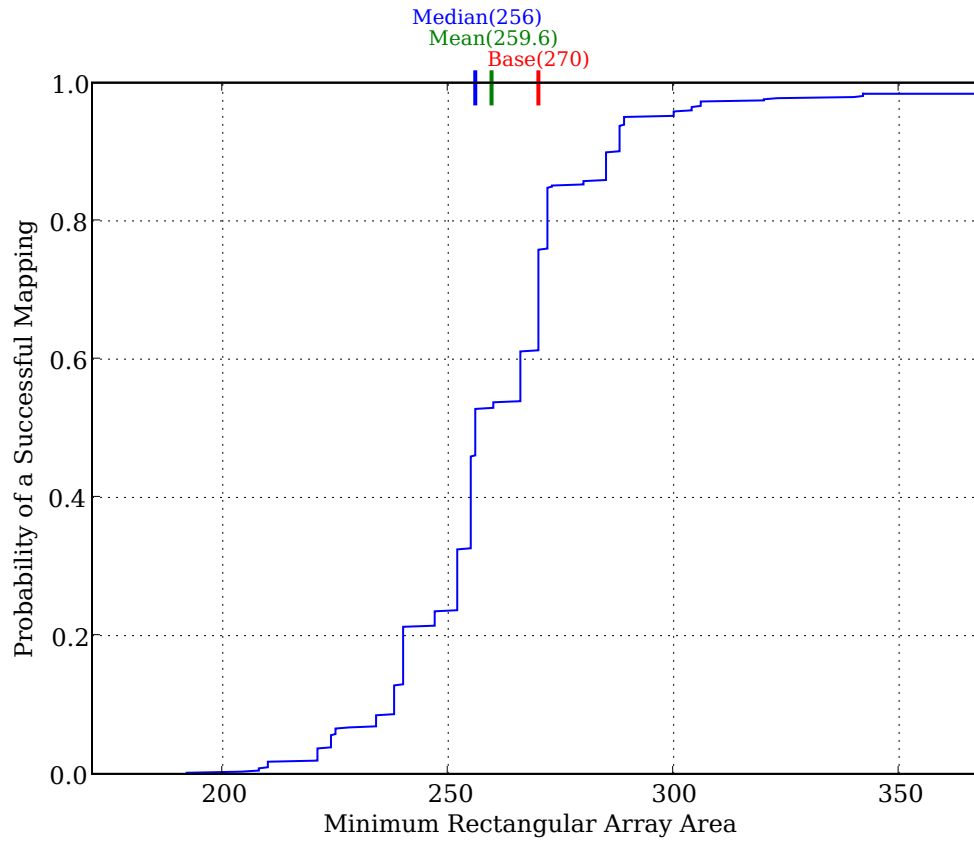


Figure 6.22: Cumulative Distribution Function for the minimum rectangular array area, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the multi-app application targeting the first version of ASAP. The statistics for this test regarding the minimum rectangular array area are: minimum = 192; maximum = 342; mean = 259.6; median = 256; base = 270.

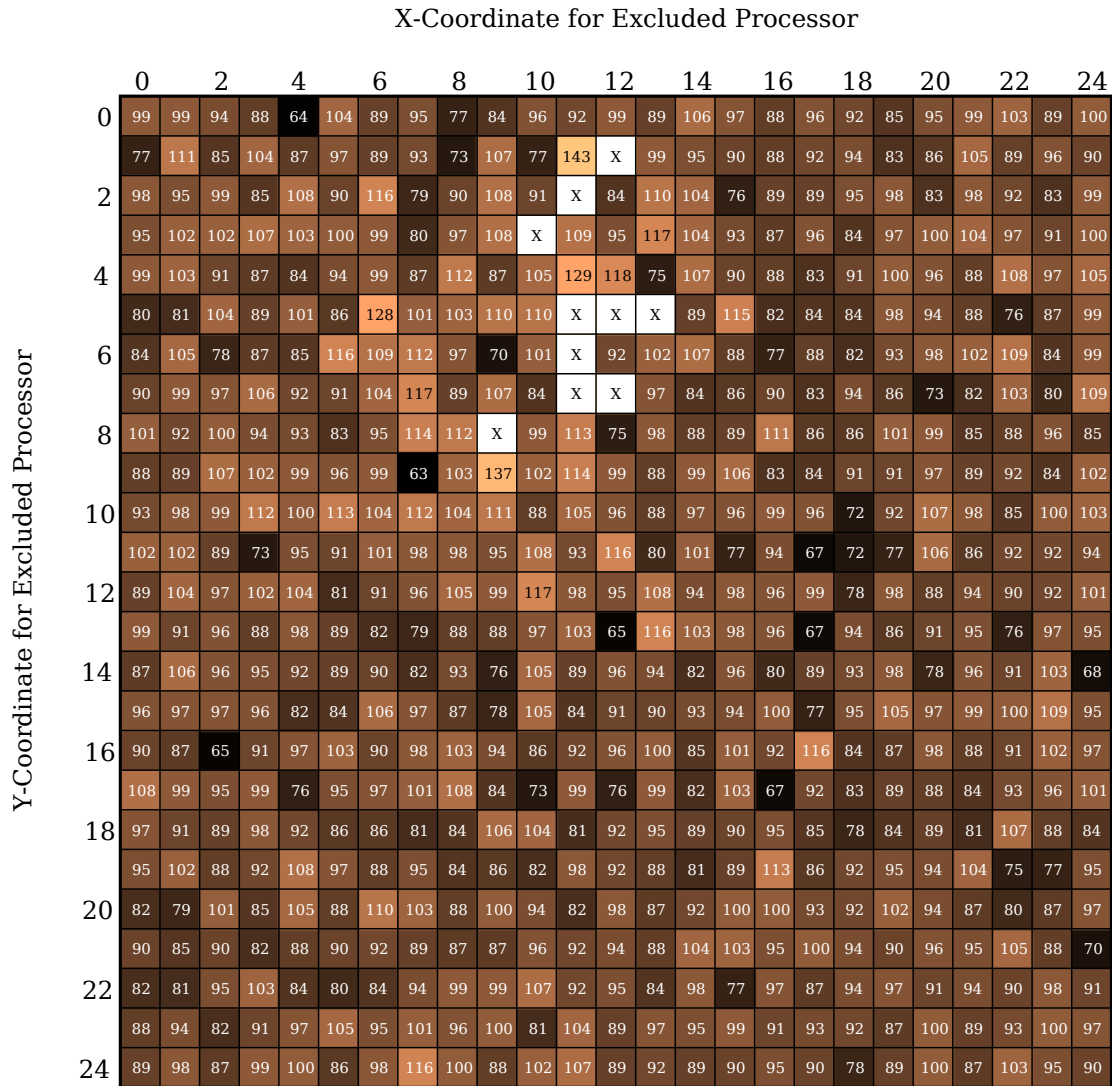


Figure 6.23: 2D-plot of the minimum number of routing processors when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the multi-app application targeting the first version of ASAP. The square with the darkest color has the lowest number of routing processors. The number of routing processors increases as the square lightens in color. A white square with a black X indicates an unmappable location. The statistics for this test regarding the minimum number of routing processors are: minimum = 63; maximum = 143; mean = 93.8; median = 94; base = 105.

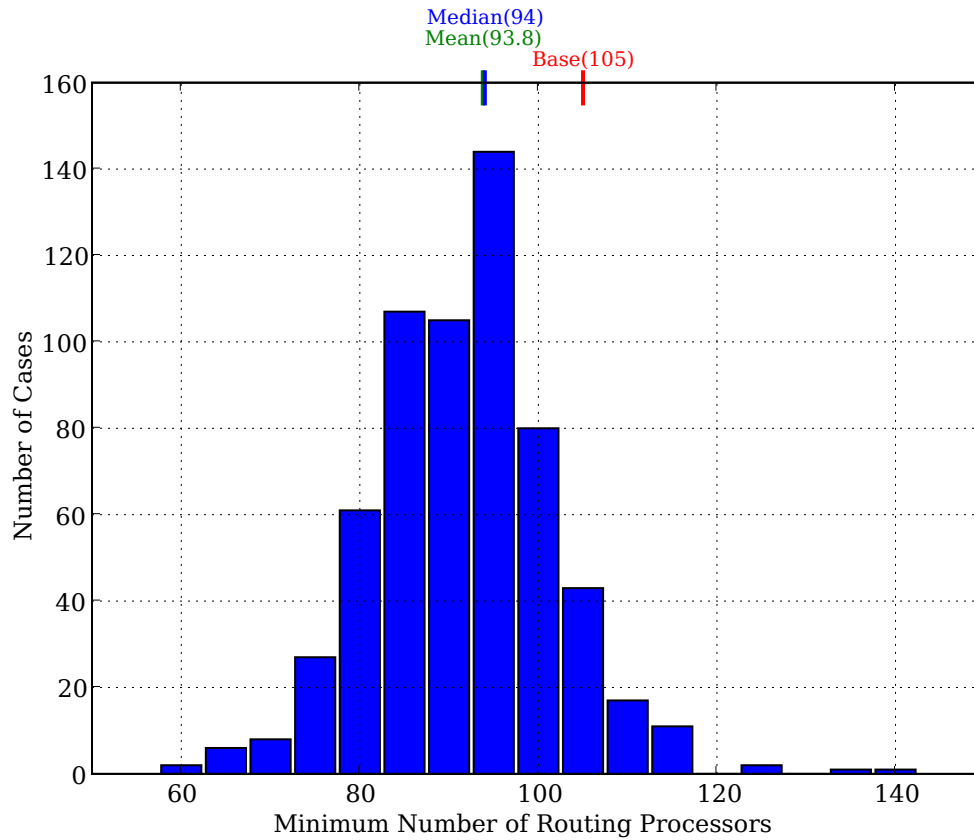


Figure 6.24: Histogram of the minimum number of routing processors, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the multi-app application targeting the first version of ASAP. The statistics for this test regarding the minimum number of routing processors are: minimum = 63; maximum = 143; mean = 93.8; median = 94; base = 105.

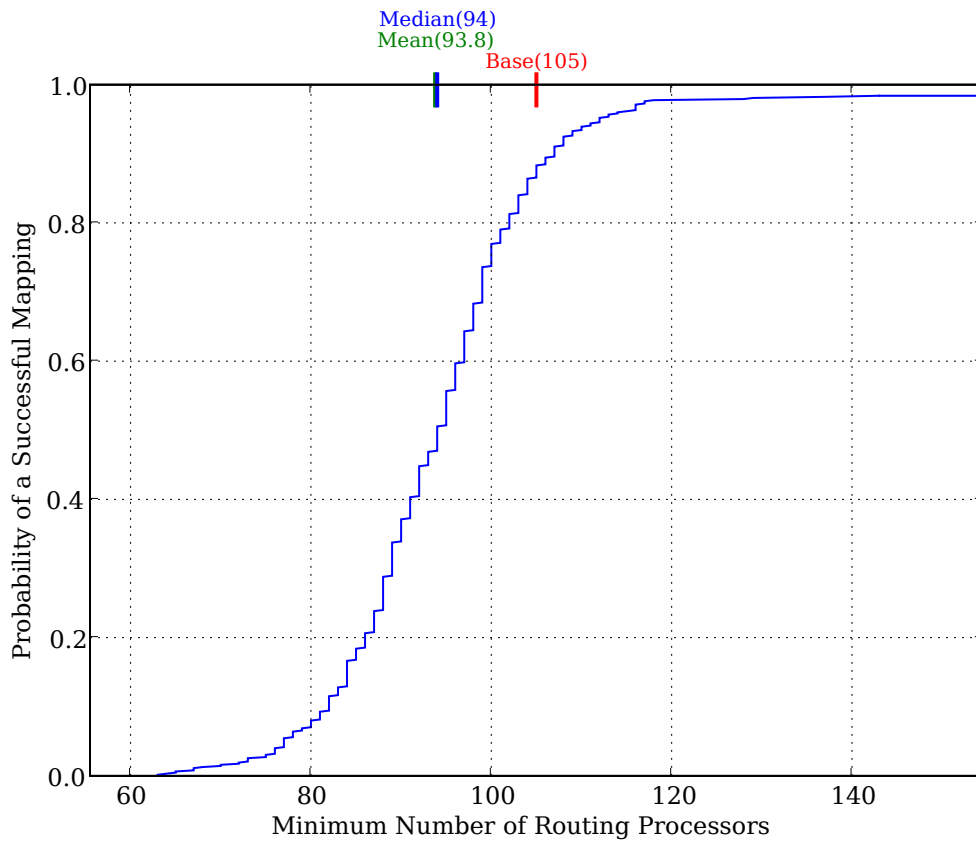


Figure 6.25: Cumulative Distribution Function for the minimum number of routing processors, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the multi-app application targeting the first version of AsAP. The statistics for this test regarding the minimum number of routing processors are: minimum = 63; maximum = 143; mean = 93.8; median = 94; base = 105.

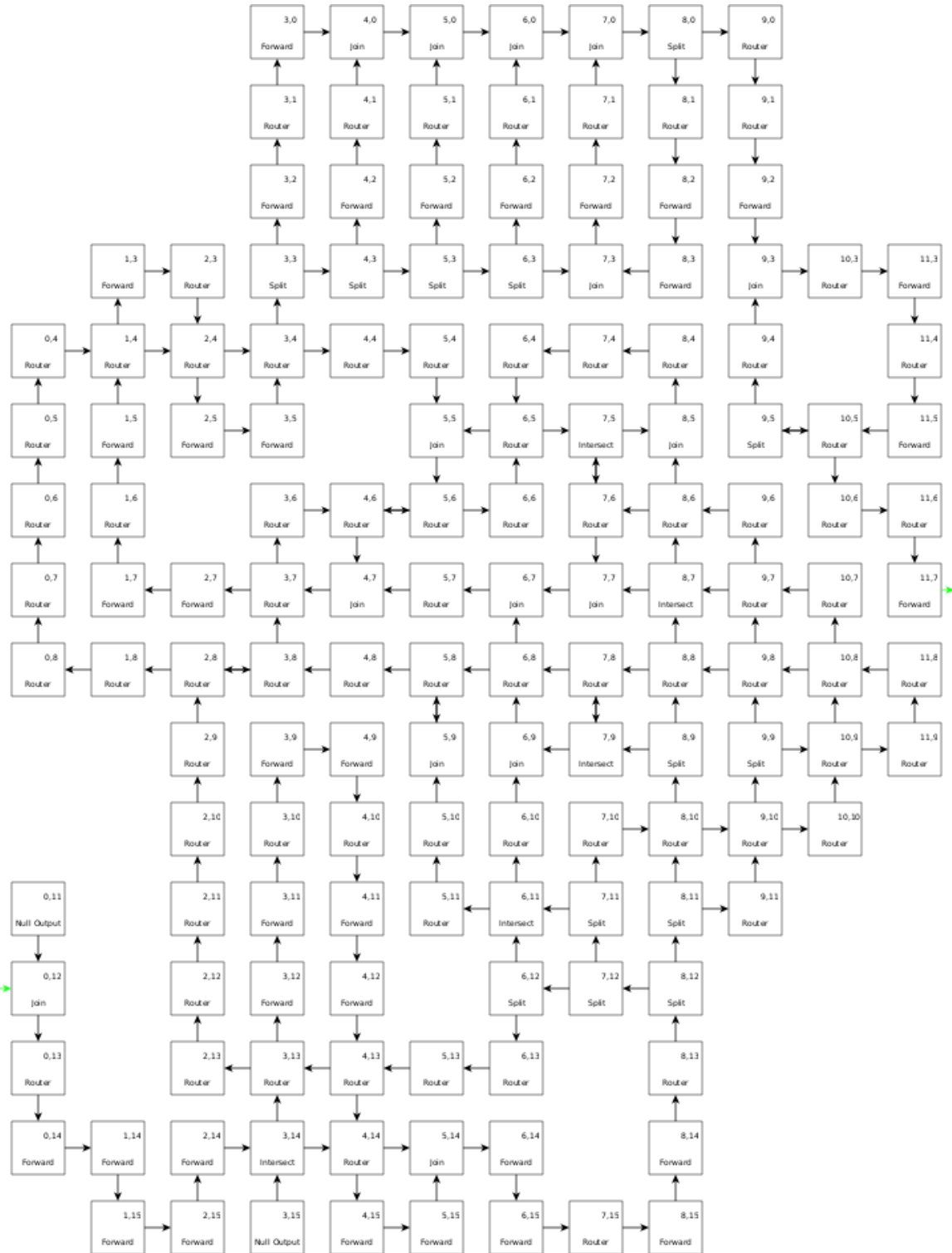


Figure 6.26: Best automatic mapping for the multi-app application after excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) while targeting the first version of ASAP. This automated mapping excludes processor (10, 1), has a rectangular array area of 192 (12x16), and uses 77 routing processors.

AsAP Version 2.0

**Settings: Defaults + “InputEdge = Left” + “UseRouting = False”
+ “CostArraySize = 2X”**

The setup for this test is almost identical to the setup for the previous test. Each processor from an array of 25x25 is excluded in sequence and 100 trials are performed for each excluded processor. One run is also made with no excluded processors and 100 trials, producing the base values. Next, the minimum values for each metric are obtained from each 100 trial block, followed by the computation of the mean and median values from these minimums. Once again the array input is allowed to float along the left edge, but for this test routing is also disabled and the cost for increasing the array size is doubled. Routing has been disabled to keep the mapping algorithm from switching back and forth between routing processors and long-distance interconnects, keeping results consistent. These additional settings are used to target the second version of AsAP and save time.

Similar to before, Figure 6.27 and Figure 6.30 are plots of the minimum rectangular array area and the minimum number of long-distance interconnects, respectively, shown as color-coded 2D-arrays. Figure 6.28 and Figure 6.31 plot the minimum rectangular array area and the minimum number of long-distance interconnects as histograms, respectively, and include value tick marks. Similarly, Figure 6.29 and Figure 6.32 plot the minimum rectangular array area and the minimum number of long-distance interconnects as cumulative distribution functions, respectively, and again include value tick marks. Figure 6.33 once again shows the best automated mapping for the multi-app application, but this time targeting the second version of AsAP and excludes processor (13, 2). Plots are not needed for the minimum number of routing processors since routing was disabled so this value is always zero.

In this test, unlike the previous test, there are no regions within the 2D-array plots that were particularly difficult to map because of excluded processors. This is because long-distance interconnects are more flexible and can be routed in different ways whereas routing processors need to be placed in certain locations (which may be blocked) in order to complete a nearest neighbor route. The histograms for the minimum rectangular array area and the minimum number of long-distance interconnects both show that the mean and base values for these two metrics are very close. This again indicates that the application is mostly unaffected by a single excluded processor. Also the base value is again greater than or equal to the mean and median values, which indicates that more trials are needed or a more highly optimized algorithm is needed. The

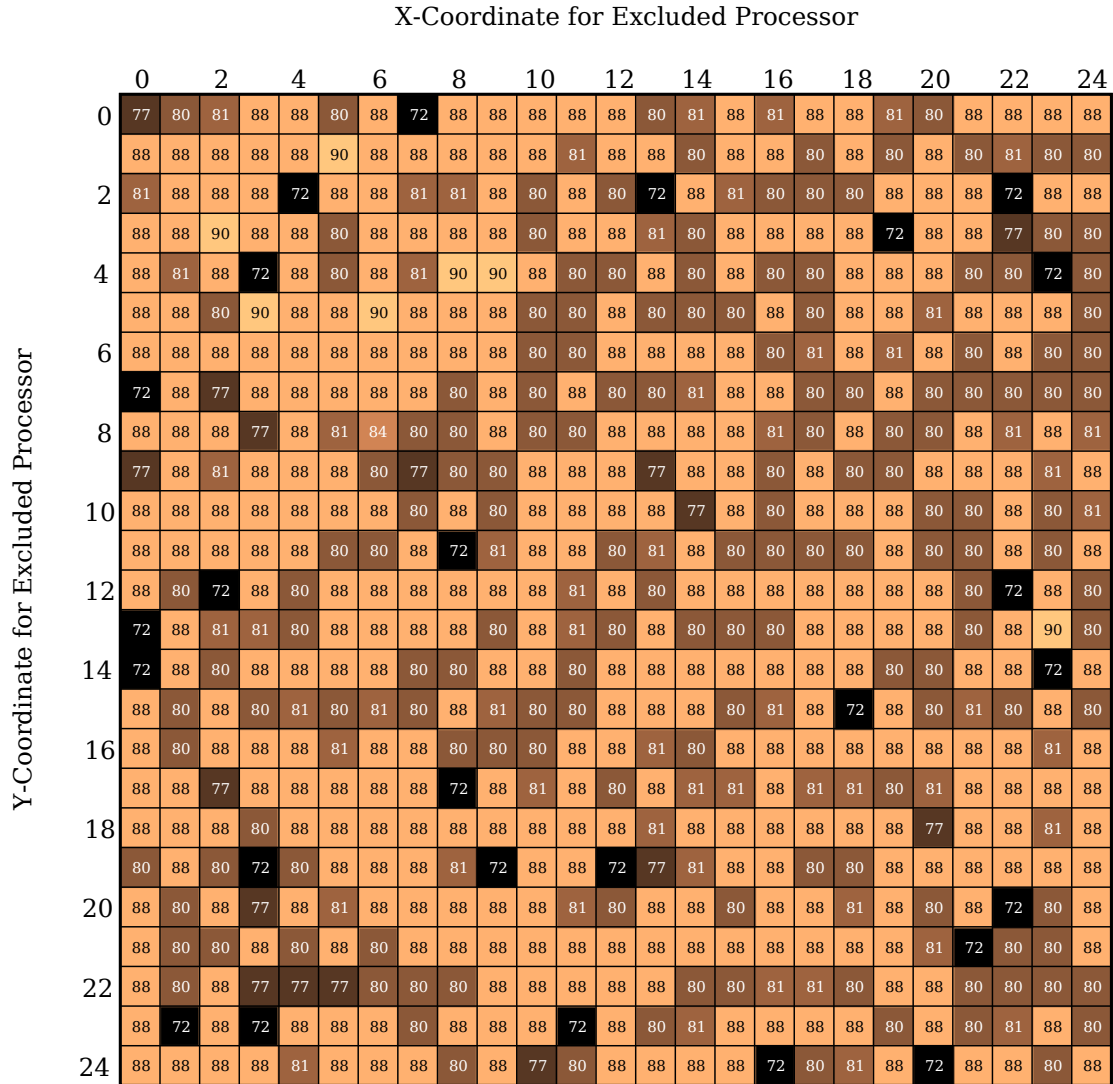


Figure 6.27: 2D-plot of the minimum rectangular array area when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the multi-app application targeting the second version of AsAP. The square with the darkest color has the lowest rectangular array area. The rectangular array area increases as the square lightens in color. The statistics for this test regarding the minimum rectangular array area are: minimum = 72; maximum = 90; mean = 84.4; median = 88; base = 88.

cumulative distribution functions show that by using a rectangular array area of 88 there is a 95% chance that the mapping will be successful with one excluded processor. By comparing the two automated mappings for the first and second versions of AsAP we notice that the rectangular array area decreased substantially ($9 \times 8 < 12 \times 16$). This shows that a significant number of routing processors were replaced by long-distance interconnects. This is a very desirable trade-off when targeting the second version of AsAP.

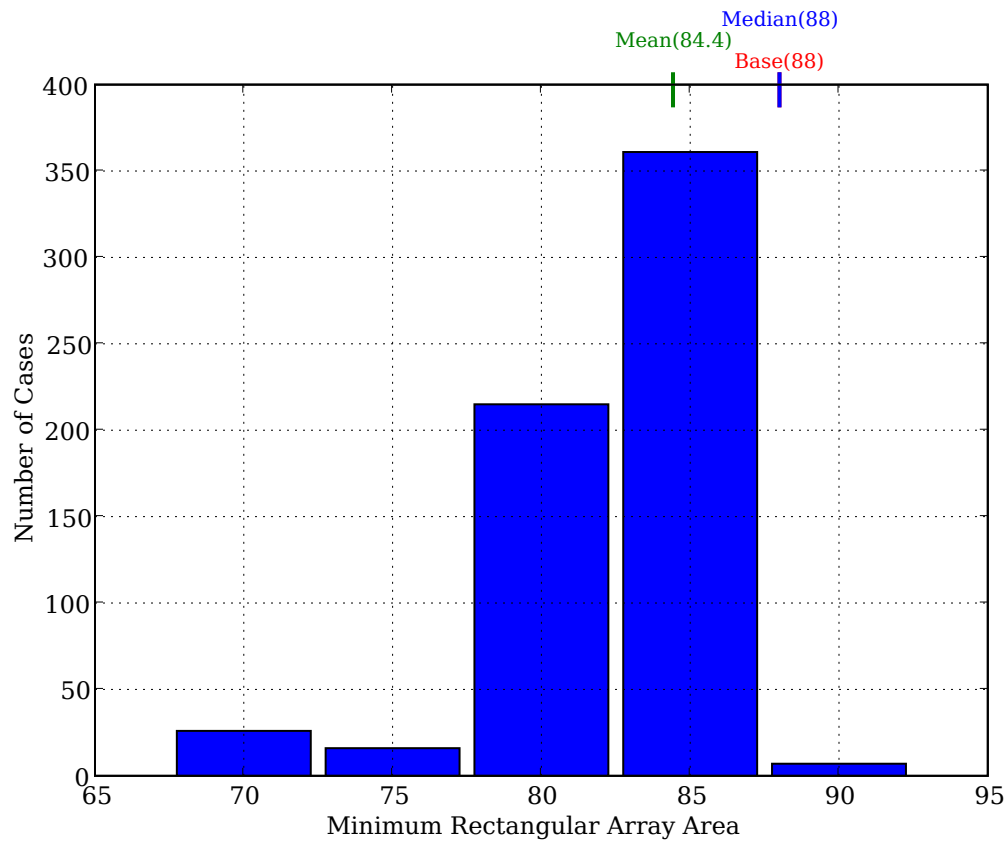


Figure 6.28: Histogram of the minimum rectangular array area, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the multi-app application targeting the second version of AsAP. The statistics for this test regarding the minimum rectangular array area are: minimum = 72; maximum = 90; mean = 84.4; median = 88; base = 88.

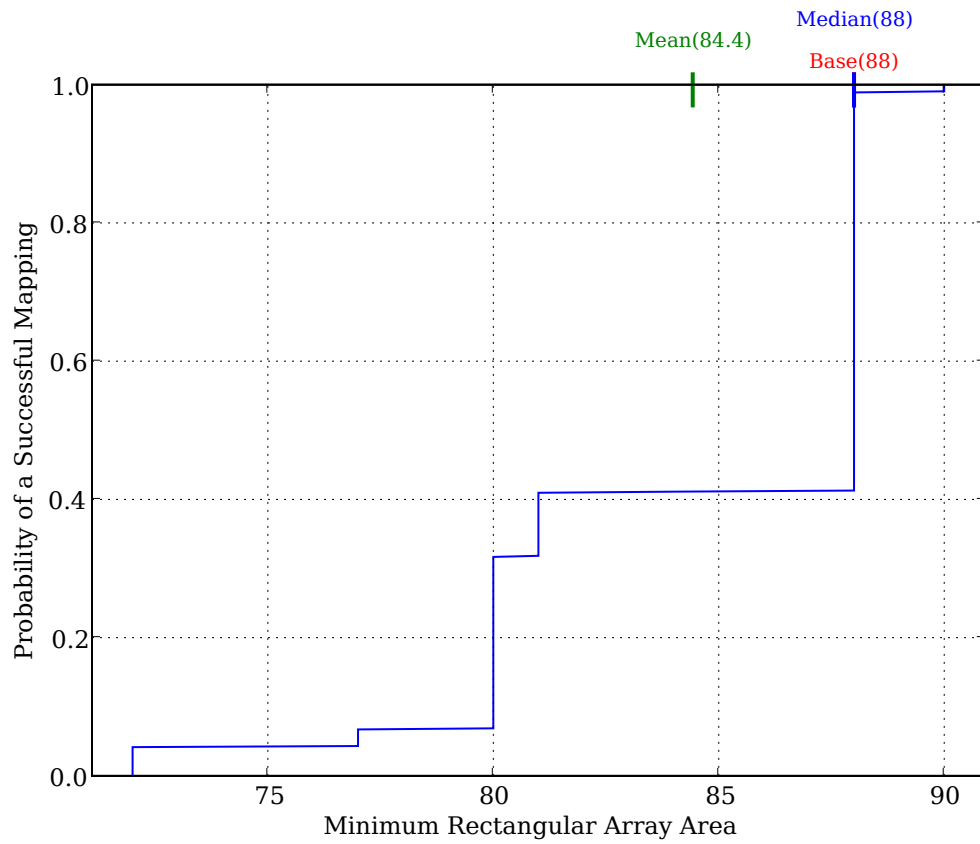


Figure 6.29: Cumulative Distribution Function for the minimum rectangular array area, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the multi-app application targeting the second version of AsAP. The statistics for this test regarding the minimum rectangular array area are: minimum = 72; maximum = 90; mean = 84.4; median = 88; base = 88.

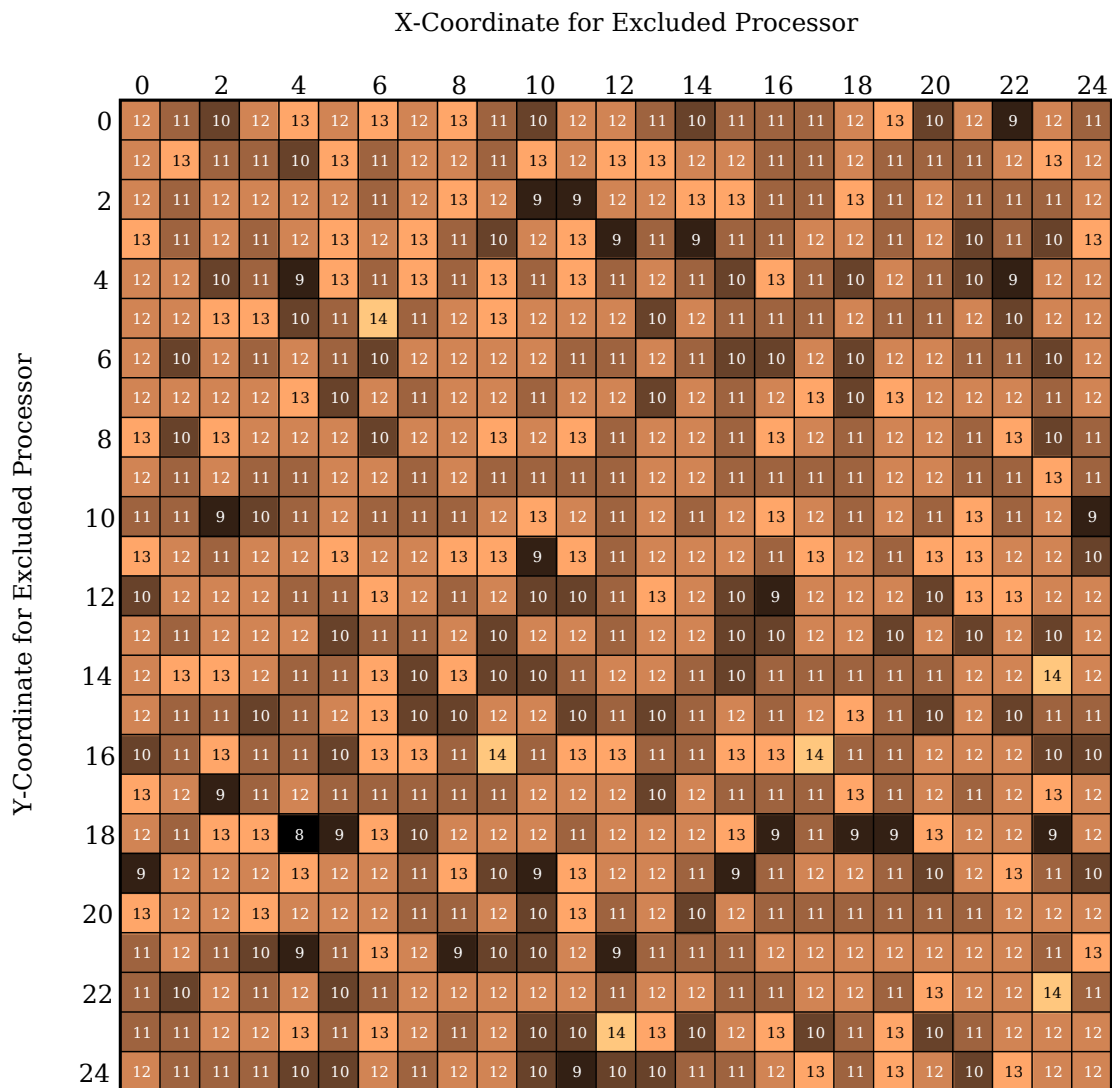


Figure 6.30: 2D-plot of the minimum number of long-distance interconnects when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the multi-app application targeting the second version of ASAP. The square with the darkest color has the lowest number of long-distance interconnects. The number of long-distance interconnects increases as the square lightens in color. The statistics for this test regarding the minimum number of long-distance interconnects are: minimum = 8; maximum = 14; mean = 11.5; median = 12; base = 13.

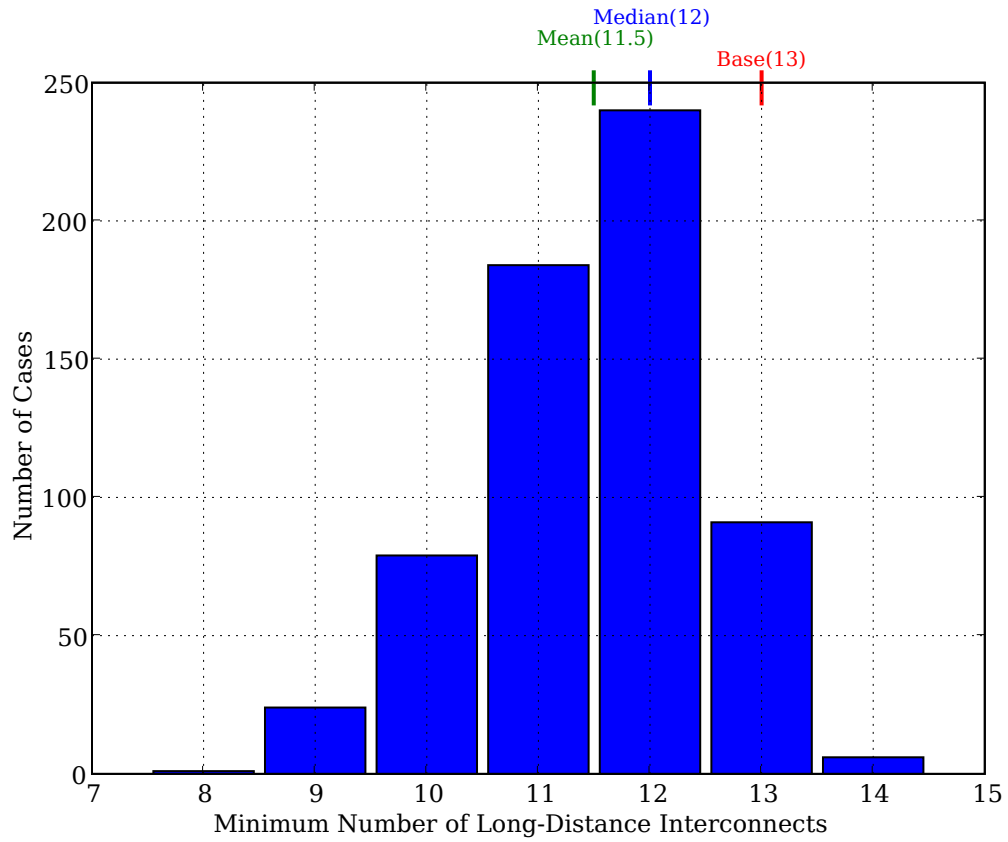


Figure 6.31: Histogram of the minimum number of long-distance interconnects, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the multi-app application targeting the second version of AsAP. The statistics for this test regarding the minimum number of long-distance interconnects are: minimum = 8; maximum = 14; mean = 11.5; median = 12; base = 13.

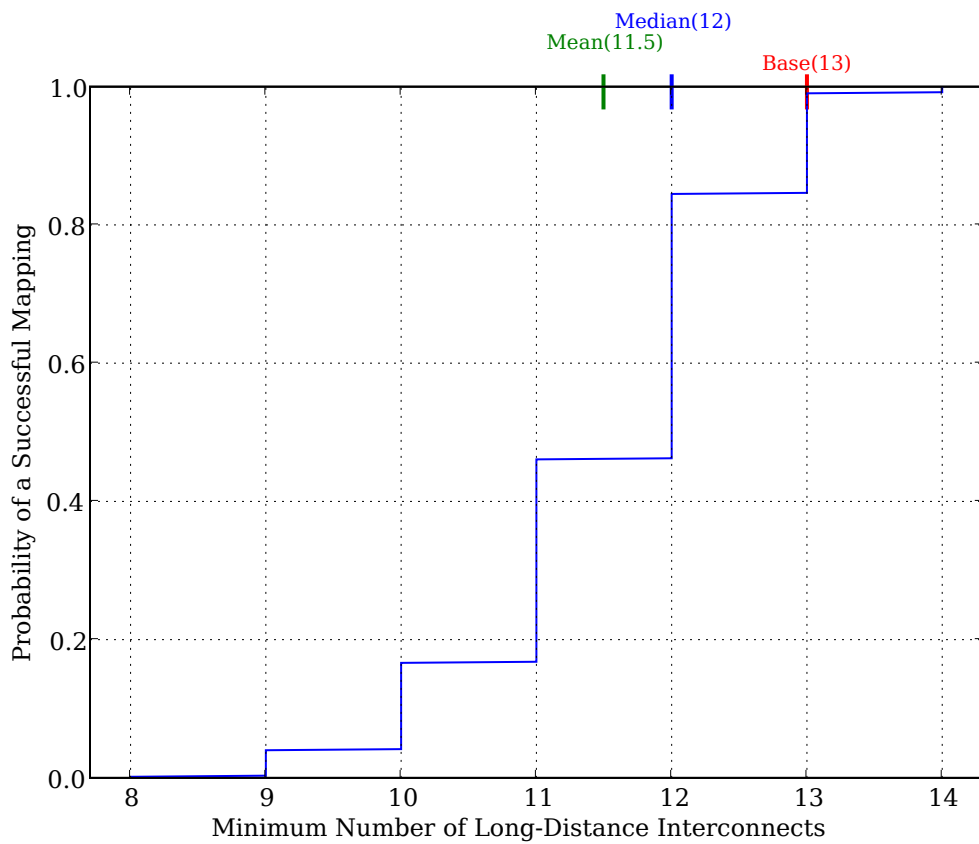


Figure 6.32: Cumulative Distribution Function for the minimum number of long-distance interconnects, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the multi-app application targeting the second version of AsAP. The statistics for this test regarding the minimum number of long-distance interconnects are: minimum = 8; maximum = 14; mean = 11.5; median = 12; base = 13.

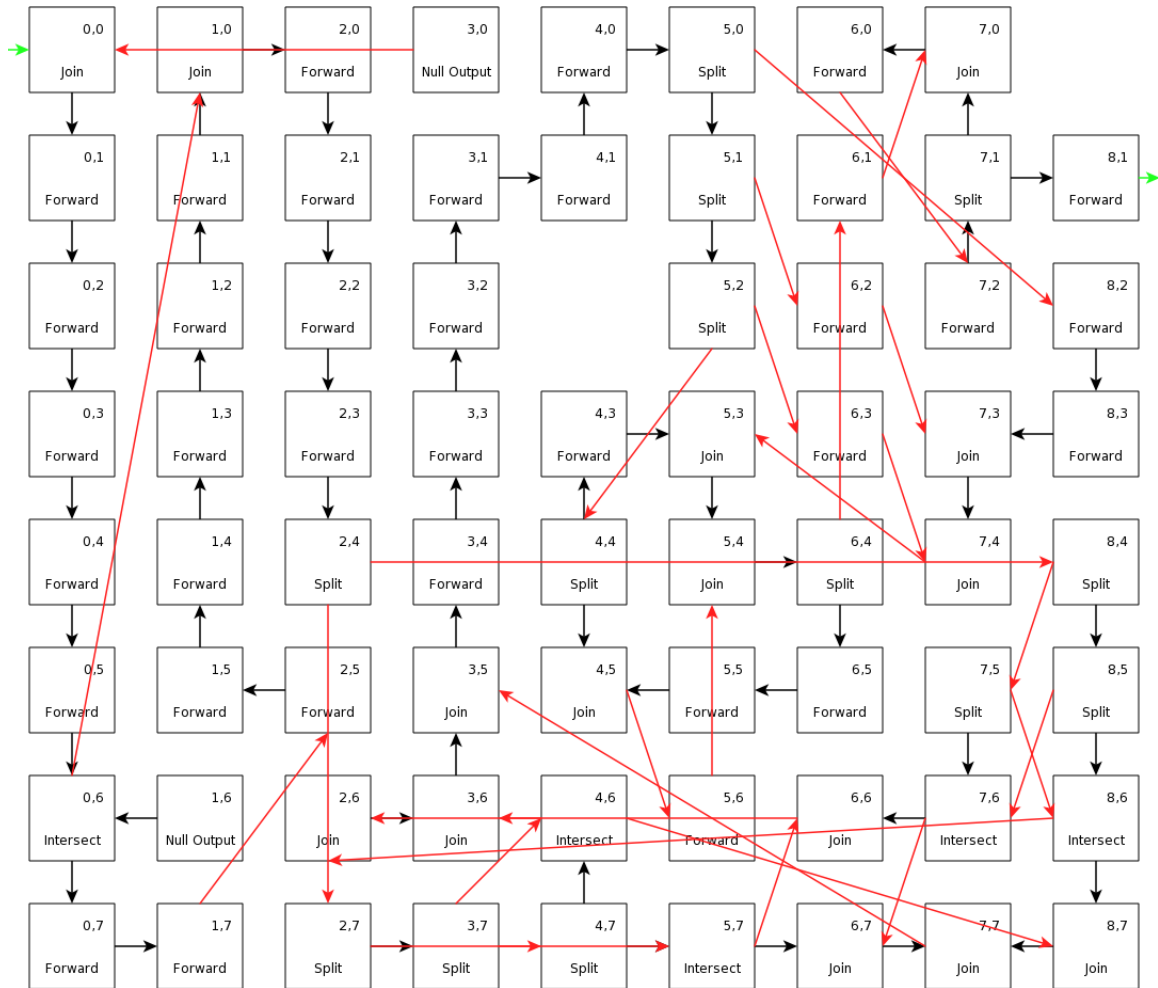


Figure 6.33: Best automatic mapping for the multi-app application after excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) while targeting the second version of AsAP. This automated mapping excludes processor (13, 2), has a rectangular array area of 72 (9x8), and uses 30 long-distance interconnects.

6.4.2 100 Random Nodes

Settings: Defaults + “InputEdge = Left”

This is the same 100 random nodes application used in the previous section (6.3). The principle behind using a random application is to avoid tuning the mapping algorithm for a specific type of application. The 100 random nodes application may be larger, and therefore take more time to map than the multi-app application, but its complexity is somewhat lower. This means a nearest neighbor only solution can be found more often. This test is setup exactly the same way as the previous single exclusion test that targeted the first version of AsAP. Each processor from an array of 25x25 is excluded in sequence and 100 trials are performed for each excluded processor. The base and minimum values are obtained first, followed by computing the mean and median values from the minimum values. The array input is again allowed to float along the left edge to lower the mapping complexity.

Figure 6.34 and Figure 6.37 are color-coded 2D-array plots of the minimum rectangular array area and the minimum number of routing processors. These are similar to the 2D-array plots from the previous tests. Figure 6.35 and Figure 6.36 plot the histogram and the cumulative distribution function for the minimum rectangular array area and include value tick marks. Figure 6.38 and Figure 6.39 plot the histogram and the cumulative distribution function for the minimum number of routing processors and also include value tick marks. Figure 6.40 shows the best automated mapping for the 100 random nodes application, while targeting the first version of AsAP, which excludes processor (14, 9). Plots are not needed for the minimum number of long-distance interconnects since this value must always be zero for the first version of AsAP.

The 2D-array plots for this test are somewhat different than the 2D-array plots from the previous tests. When excluding processors near the center of the array the minimum rectangular array area and the minimum number of routing processors decreases. The explanation for this lies in how the space insertion component works. This application, being somewhat easier to map than the previous application, does not always require additional space for routing. Excluded processors sometimes block the expansion that occurs when adding space for routing. The result is a decrease in rectangular array area. However, the mean and base values are still very close, as shown in the histograms for the minimum rectangular array area and the minimum number of routing processors. This indicates that the application is tolerant of single processor exclusions. As mentioned before, the base value is greater than or equal to the mean and median values because an insufficient number trials were performed, or possibly a more highly optimized algorithm is needed. The cumulative

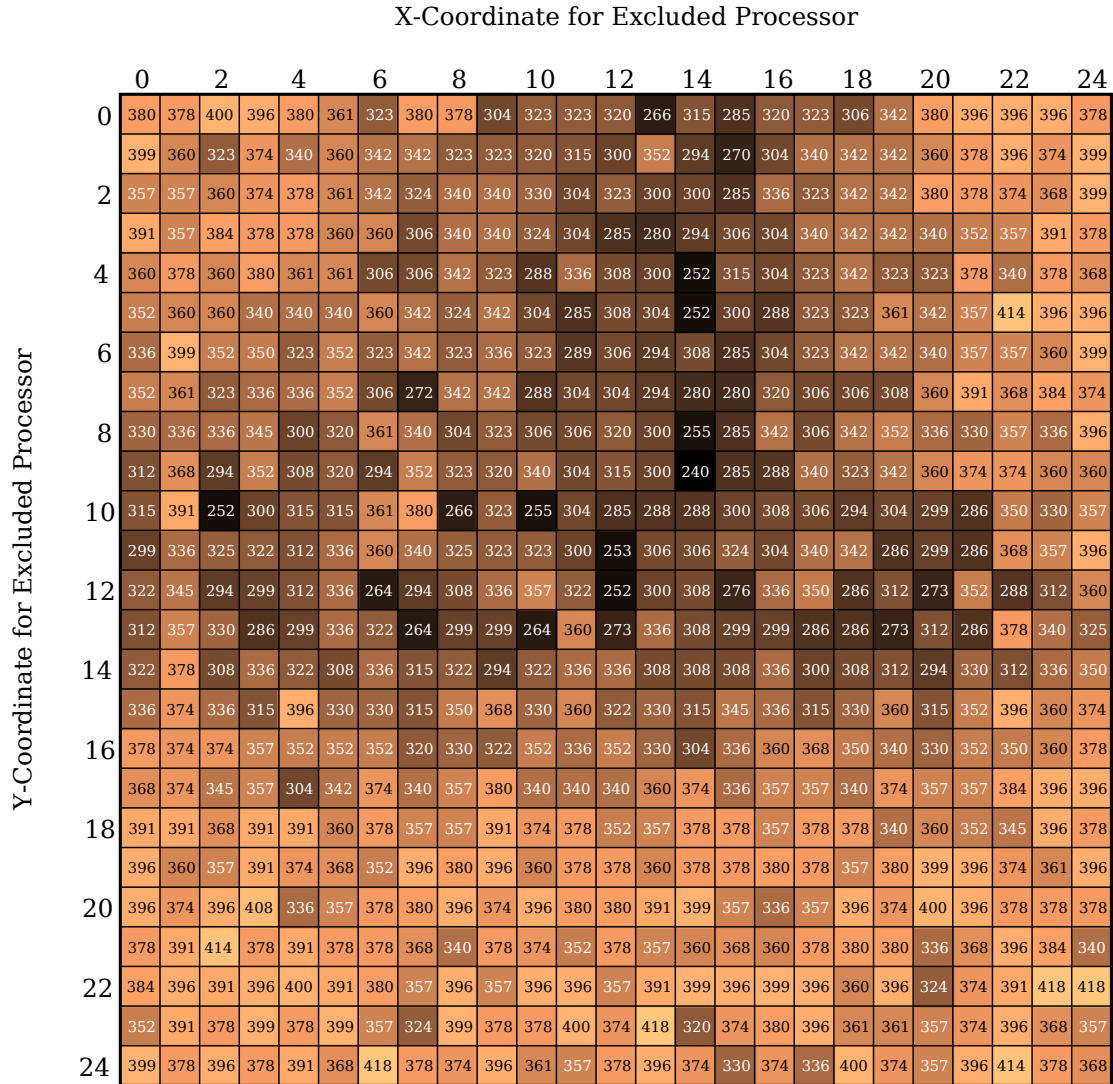


Figure 6.34: 2D-plot of the minimum rectangular array area when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the 100 random nodes application targeting the first version of AsAP. The square with the darkest color has the lowest rectangular array area. The rectangular array area increases as the square lightens in color. The statistics for this test regarding the minimum rectangular array area are: minimum = 240; maximum = 418; mean = 345.2; median = 350; base = 357.

distribution functions show that by using a rectangular array area of 396 there is a 95% chance that the mapping will be successful with one excluded processor.

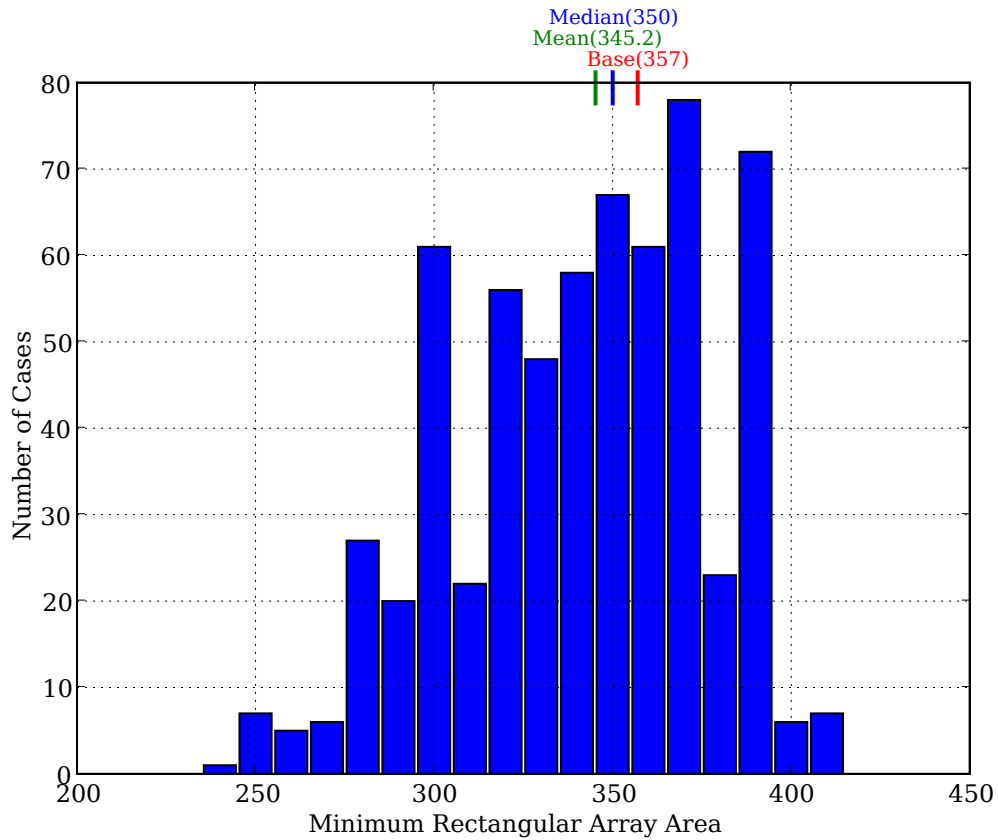


Figure 6.35: Histogram of the minimum rectangular array area, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the 100 random nodes application targeting the first version of AsAP. The statistics for this test regarding the minimum rectangular array area are: minimum = 240; maximum = 418; mean = 345.2; median = 350; base = 357.

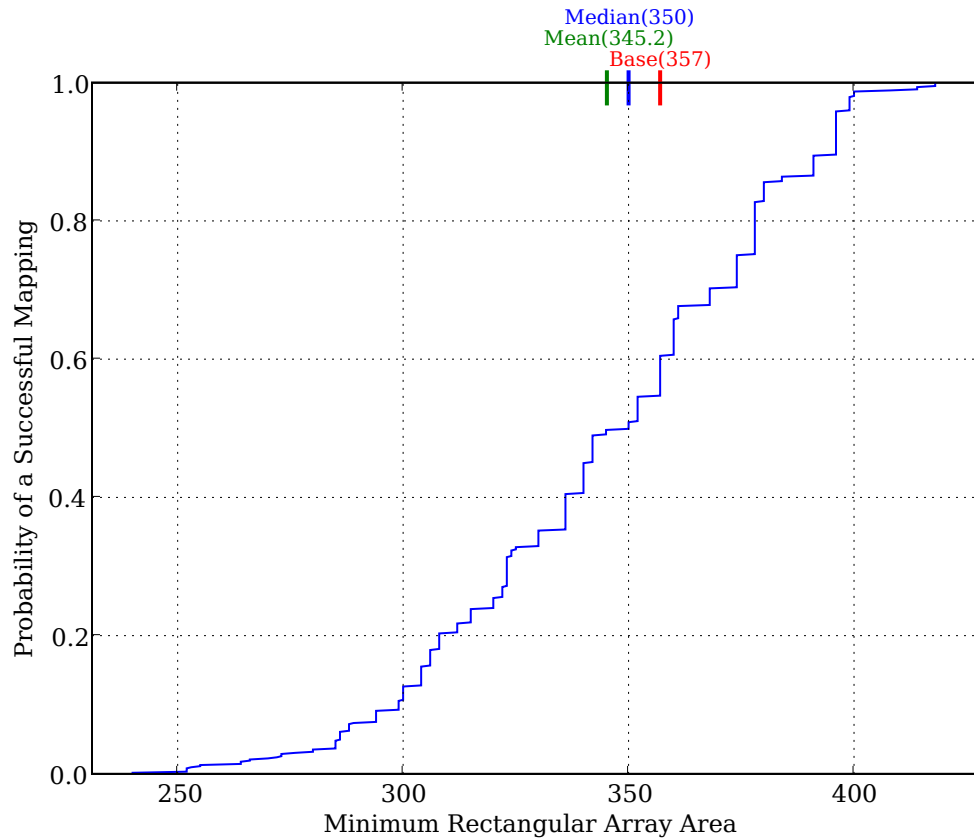


Figure 6.36: Cumulative Distribution Function for the minimum rectangular array area, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the 100 random nodes application targeting the first version of AsAP. The statistics for this test regarding the minimum rectangular array area are: minimum = 240; maximum = 418; mean = 345.2; median = 350; base = 357.

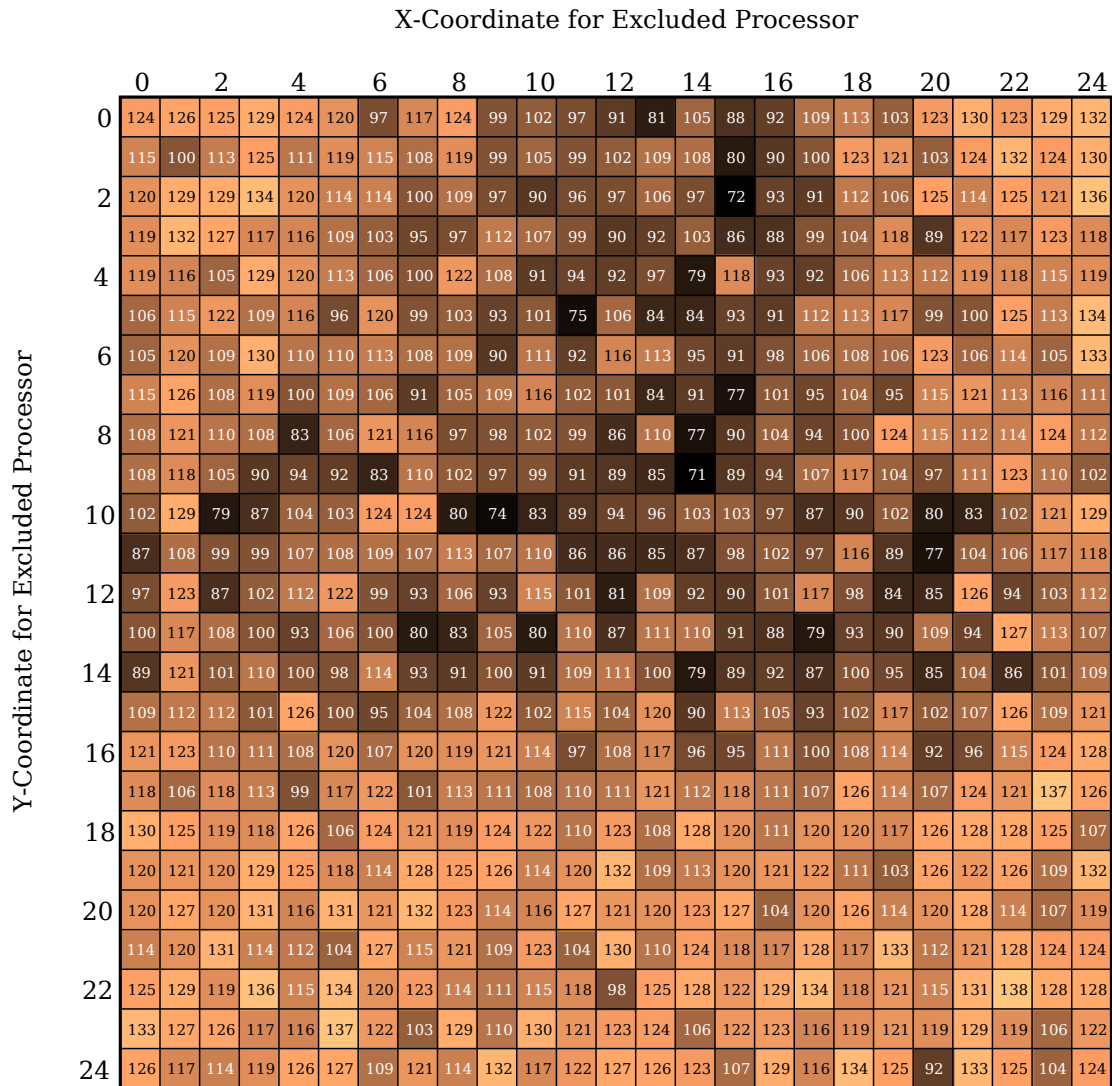


Figure 6.37: 2D-plot of the minimum number of routing processors when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the 100 random nodes application targeting the first version of AsAP. The square with the darkest color has the lowest number of routing processors. The number of routing processors increases as the square lightens in color. The statistics for this test regarding the minimum number of routing processors are: minimum = 71; maximum = 138; mean = 110; median = 111; base = 110.

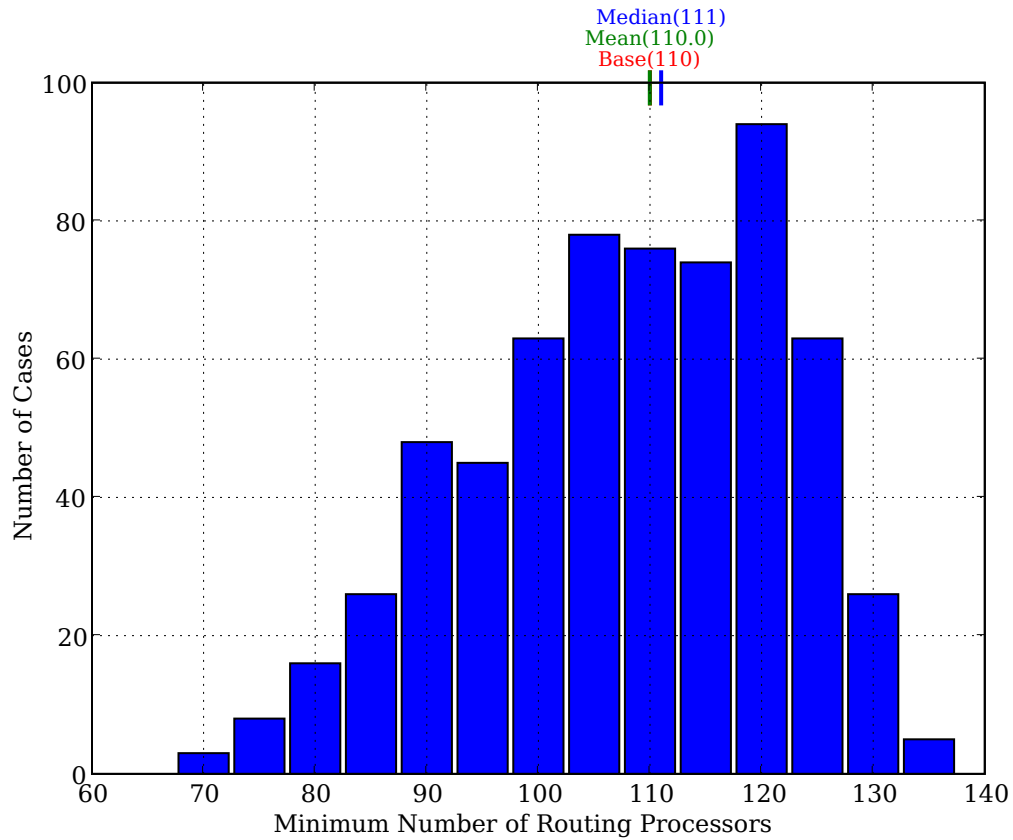


Figure 6.38: Histogram of the minimum number of routing processors, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the 100 random nodes application targeting the first version of ASAP. The statistics for this test regarding the minimum number of routing processors are: minimum = 71; maximum = 138; mean = 110; median = 111; base = 110.

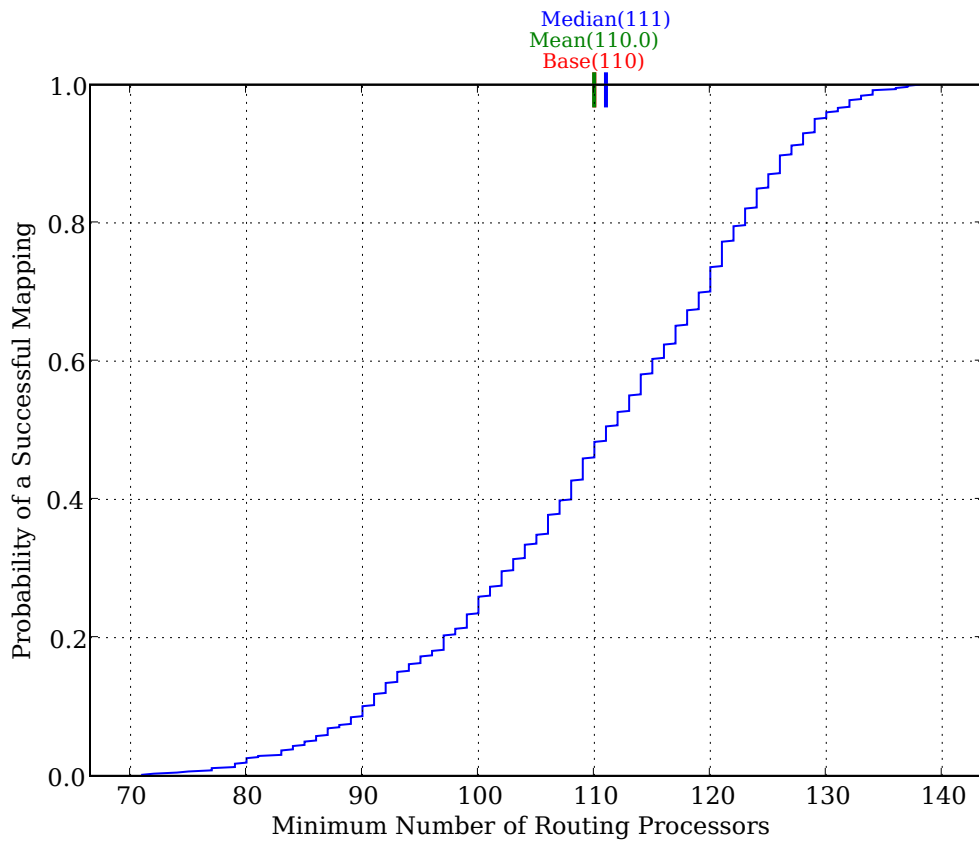


Figure 6.39: Cumulative Distribution Function for the minimum number of routing processors, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the 100 random nodes application targeting the first version of ASAP. The statistics for this test regarding the minimum number of routing processors are: minimum = 71; maximum = 138; mean = 110; median = 111; base = 110.

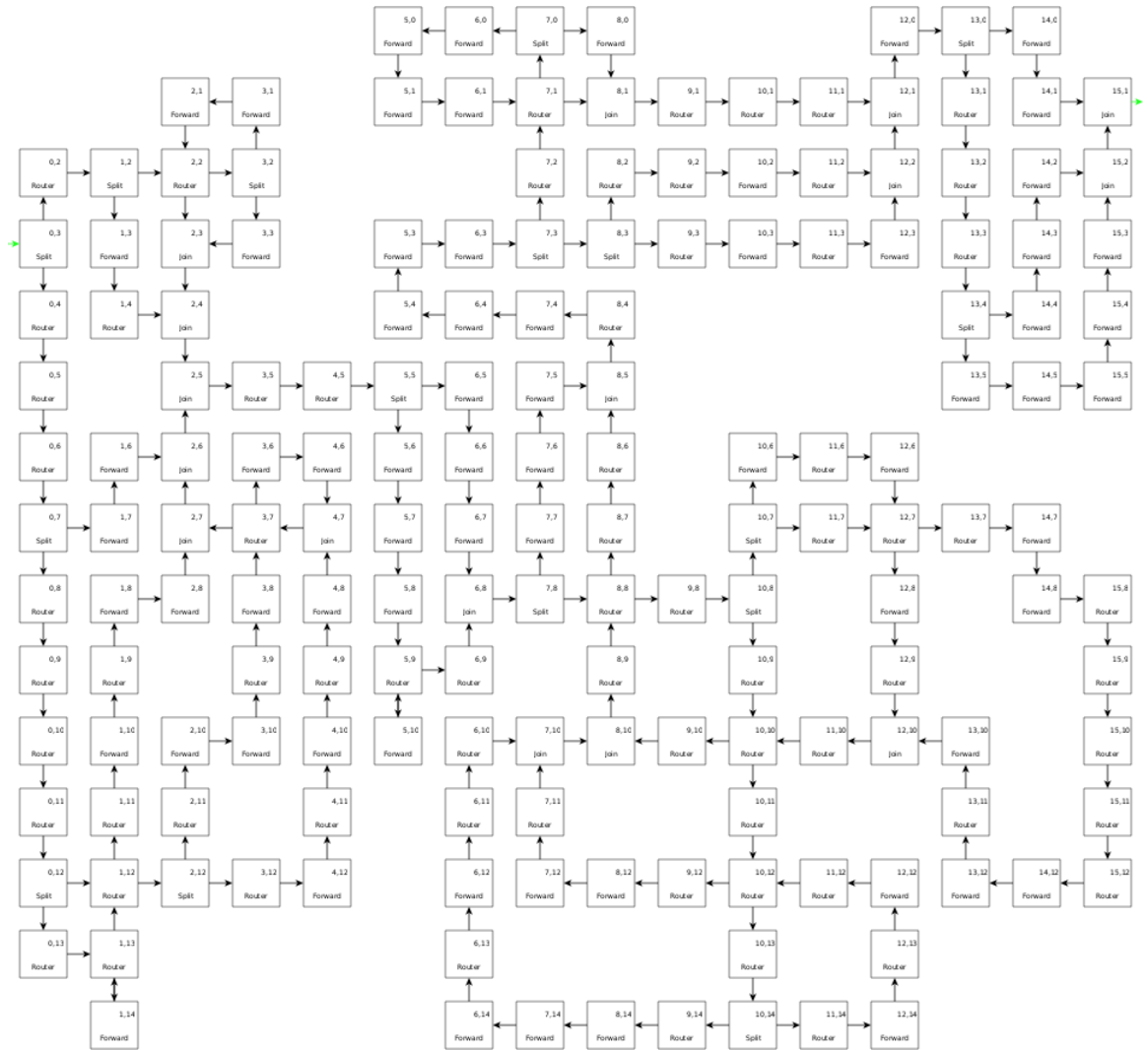


Figure 6.40: Best automatic mapping for the 100 random nodes application after excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) while targeting the first version of AsAP. This automated mapping excludes processor (14, 9), has a rectangular array area of 240 (16x15), and uses 71 routing processors.

AsAP Version 2.0

**Settings: Defaults + “InputEdge = Left” + “UseRouting = False”
+ “CostArraySize = 2X”**

The setup for this test is identical to the setup used for the previous single processor exclusion test that targeted the second version of AsAP. Each processor from an array of size 25x25 is excluded in sequence and 100 trials are performed for each excluded processor. The base and minimum values are obtained first, followed by computing the mean and median values from the minimum values. Just like before, the array input is allowed to float along the left edge of the array, the routing phase is disabled, and the cost for increasing the array size is doubled to target the second version of AsAP.

Similar to the other tests, Figure 6.41 and Figure 6.44 are color-coded 2D-array plots of the minimum rectangular array area and the minimum number of long-distance interconnects, respectively. Figure 6.42 and Figure 6.45 plot the minimum rectangular array area and the minimum number of long-distance interconnects as histograms, respectively, and include tick marks for the mean, median, and base values. Figure 6.43 and Figure 6.46 plot the minimum rectangular array area and the minimum number of long-distance interconnects as cumulative distribution functions, respectively, and again include tick marks for the mean, median, and base values. Figure 6.47 again shows the best automated mapping for the 100 random nodes application, but this time targeting the second version of AsAP and excludes processor (0, 0). Plots are not needed for the minimum number of routing processors since routing was disabled and therefore this value is always zero.

The 2D-array plots for this test are very similar to the 2D-array plots for the previous single exclusion test targeting the second version of AsAP. There are no regions within the 2D-array plots that are particularly more or less difficult to map because of the excluded processors. This is again because long-distance interconnects are more flexible than routing processors. The histograms for the minimum rectangular array area and the minimum number of long-distance interconnects both show that the mean and base values for these two metrics are quite close. This confirms that the application is mostly unaffected by single processor exclusions. Surprisingly, the base value is less than the mean and median values for the minimum rectangular array area, though not by much. Yet, the base value is greater than the mean and median values for the minimum number of long-distance interconnects likely due to the number of trials that were performed, or possibly a more highly optimized algorithm is needed. The cumulative distribution functions indicate that by using a rectangular array area of 130 there is a 95% chance that the mapping will be successful with one

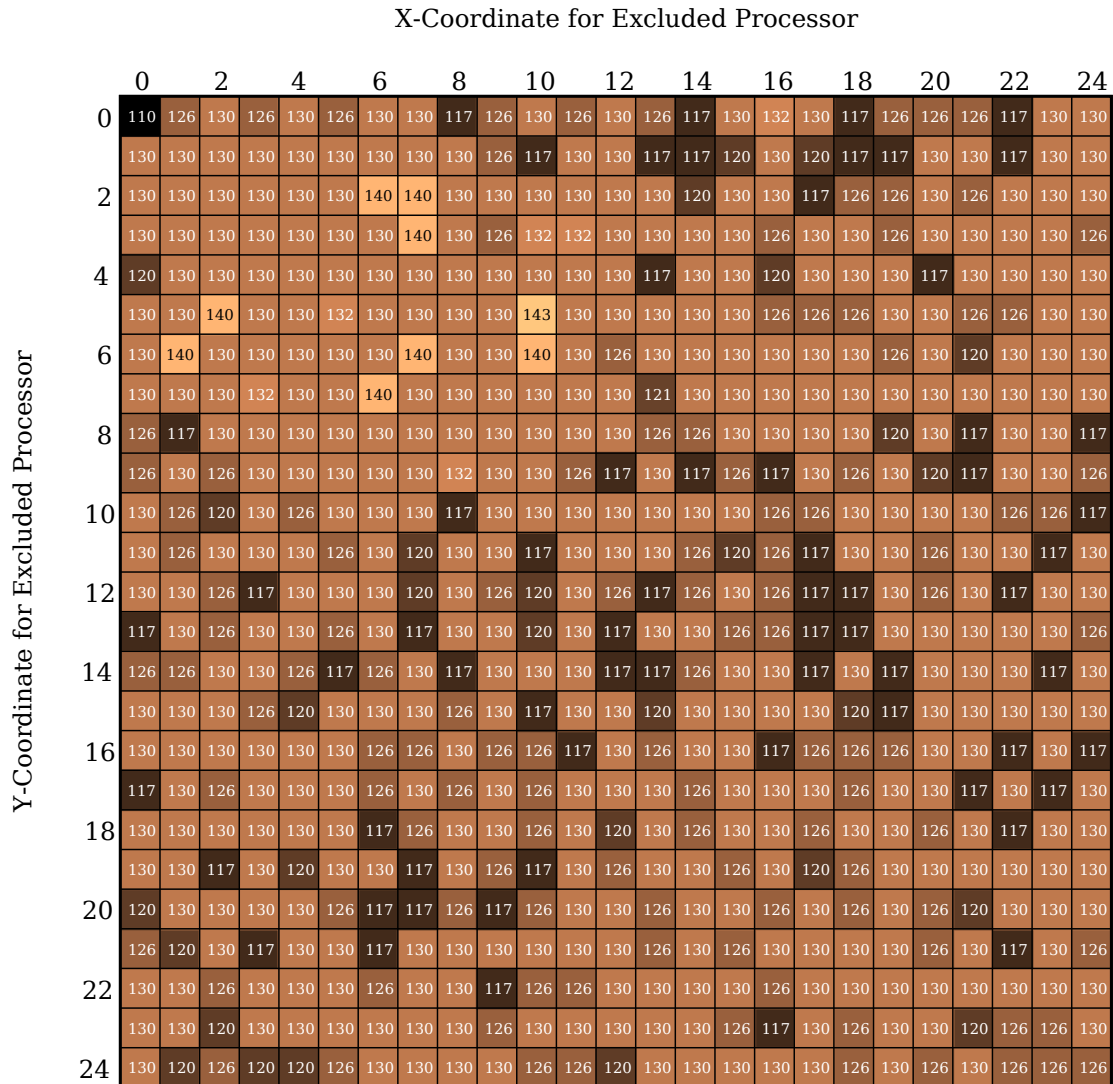


Figure 6.41: 2D-plot of the minimum rectangular array area when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the 100 random nodes application targeting the second version of AsAP. The square with the darkest color has the lowest rectangular array area. The rectangular array area increases as the square lightens in color. The statistics for this test regarding the minimum rectangular array area are: minimum = 110; maximum = 143; mean = 127.6; median = 130; base = 120.

excluded processor. Comparing the automated mappings for the first and second versions of AsAP we can see that again the rectangular array area decreased substantially ($11 \times 10 < 16 \times 15$). This is due to the trade-off between routing processors and long-distance interconnects.

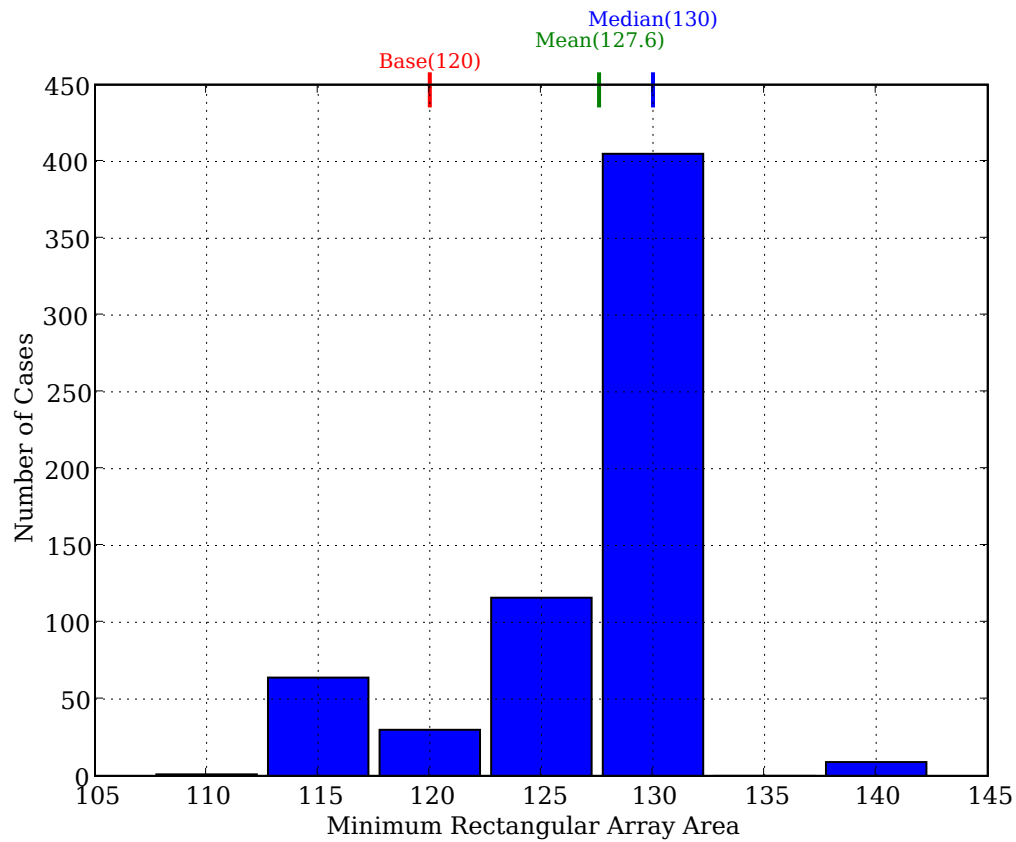


Figure 6.42: Histogram of the minimum rectangular array area, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the 100 random nodes application targeting the second version of AsAP. The statistics for this test regarding the minimum rectangular array area are: minimum = 110; maximum = 143; mean = 127.6; median = 130; base = 120.

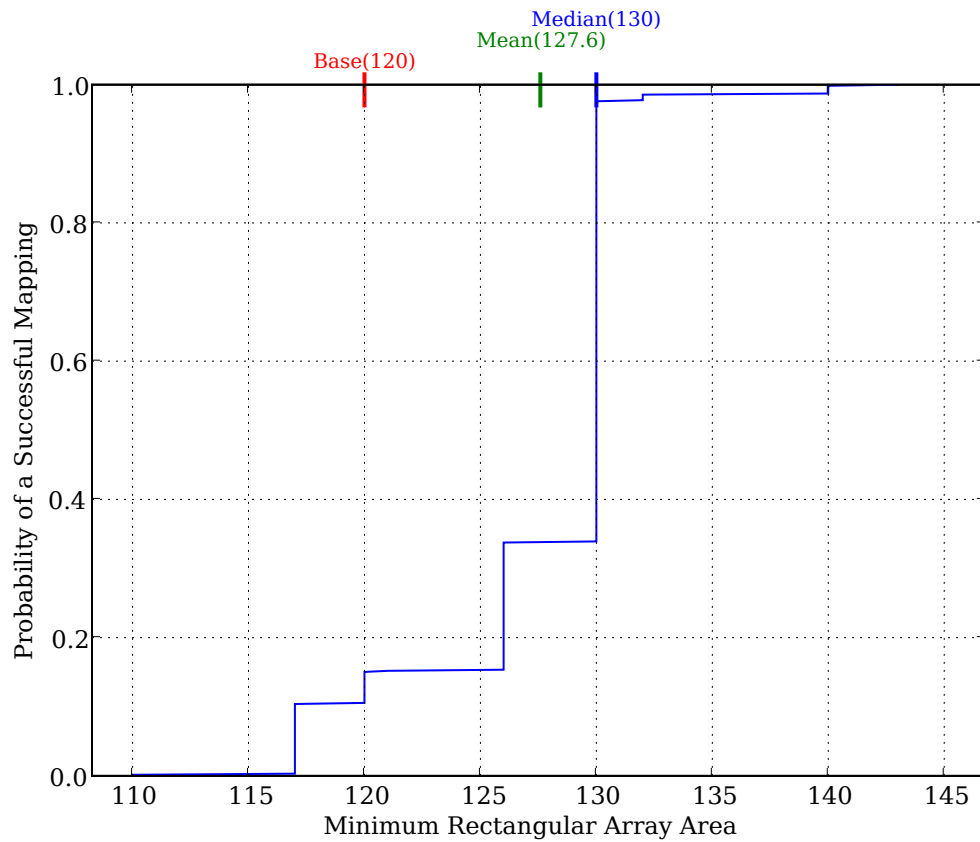


Figure 6.43: Cumulative Distribution Function for the minimum rectangular array area, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the 100 random nodes application targeting the second version of AsAP. The statistics for this test regarding the minimum rectangular array area are: minimum = 110; maximum = 143; mean = 127.6; median = 130; base = 120.

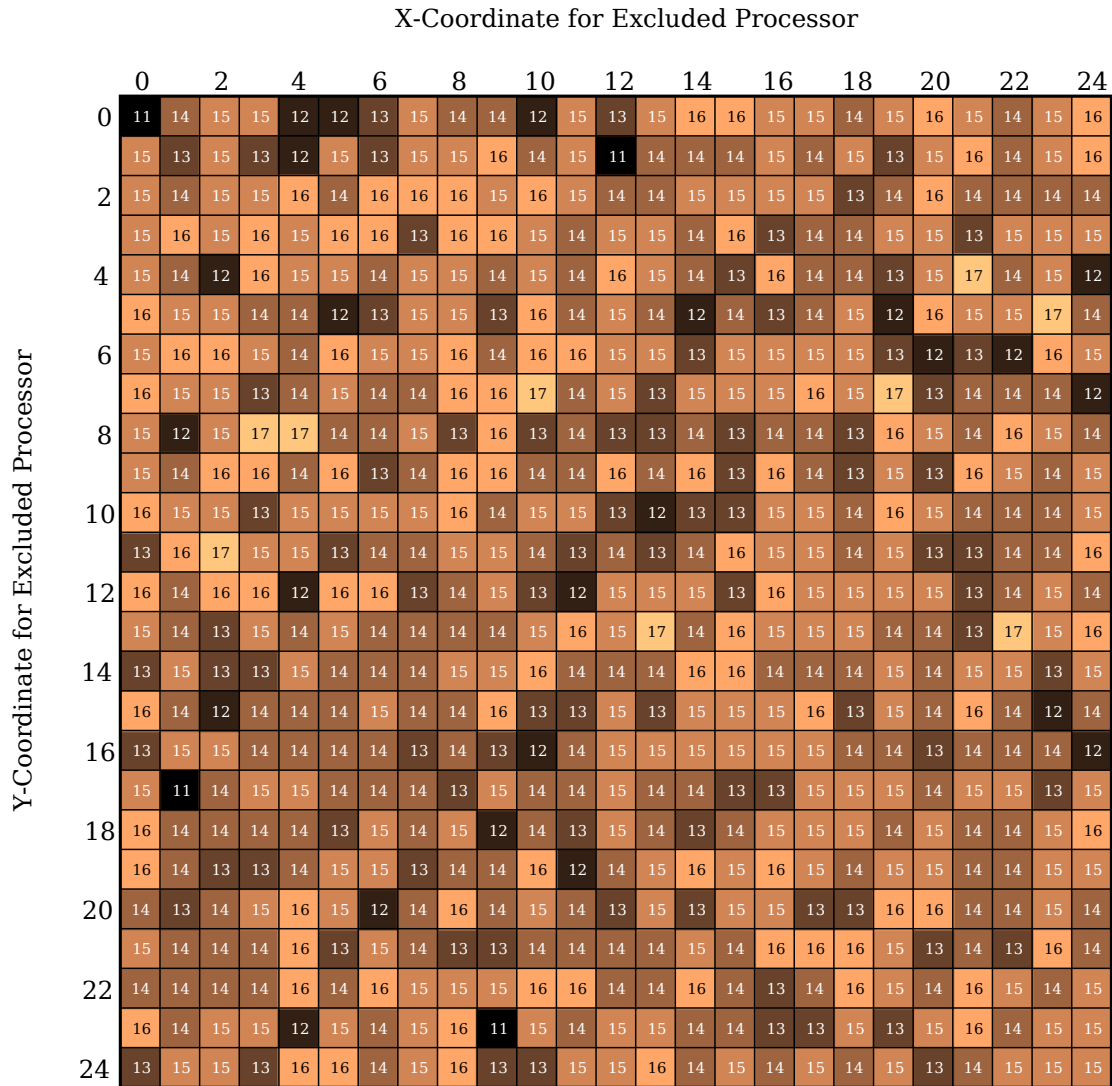


Figure 6.44: 2D-plot of the minimum number of long-distance interconnects when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the 100 random nodes application targeting the second version of AsAP. The square with the darkest color has the lowest number of long-distance interconnects. The number of long-distance interconnects increases as the square lightens in color. The statistics for this test regarding the minimum number of long-distance interconnects are: minimum = 11; maximum = 17; mean = 14.5; median = 15; base = 15.

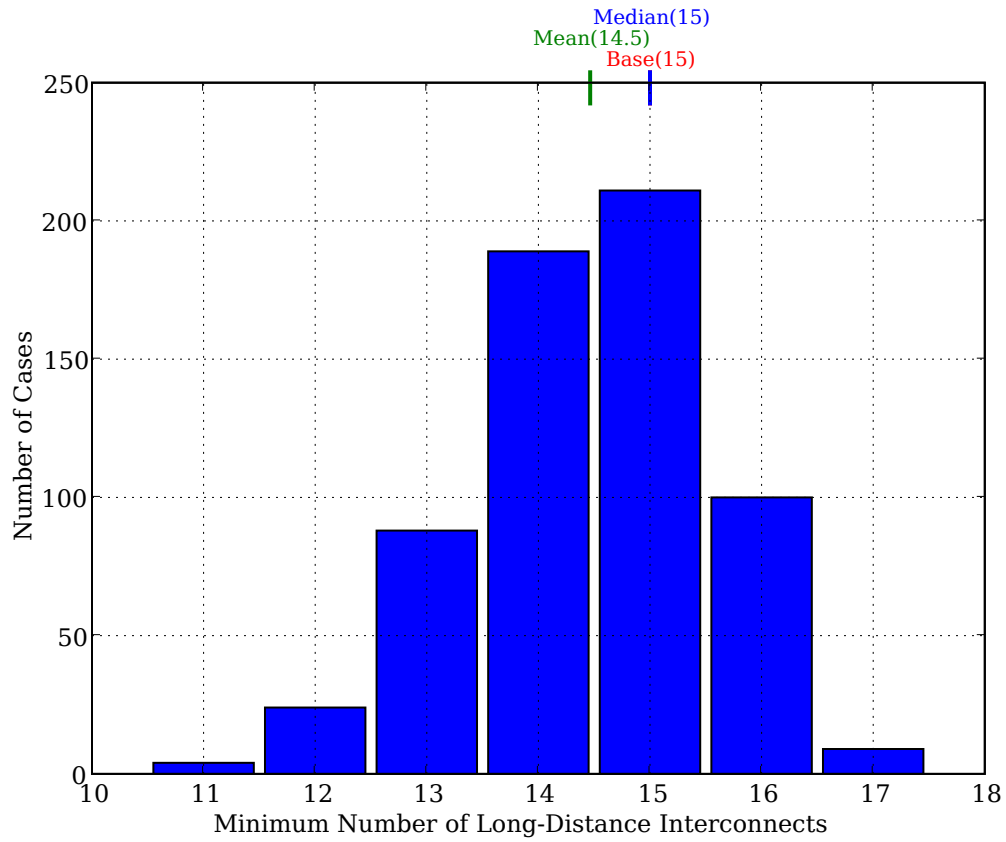


Figure 6.45: Histogram of the minimum number of long-distance interconnects, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the 100 random nodes application targeting the second version of AsAP. The statistics for this test regarding the minimum number of long-distance interconnects are: minimum = 11; maximum = 17; mean = 14.5; median = 15; base = 15.

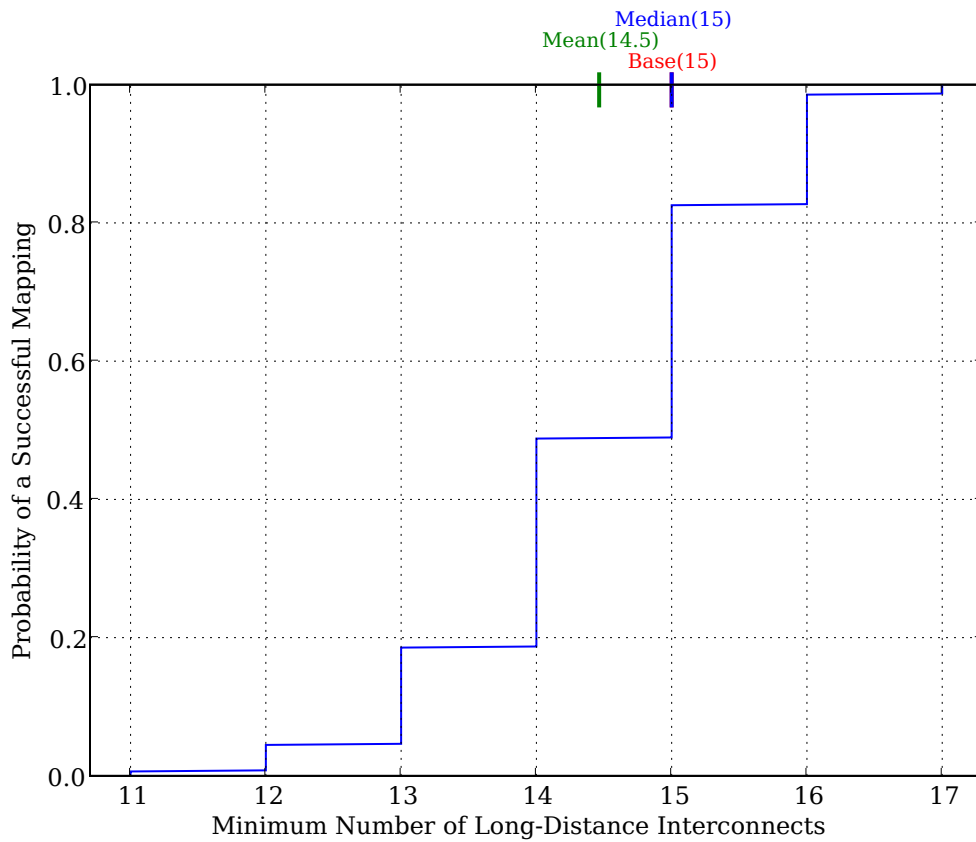


Figure 6.46: Cumulative Distribution Function for the minimum number of long-distance interconnects, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) for the 100 random nodes application targeting the second version of ASAP. The statistics for this test regarding the minimum number of long-distance interconnects are: minimum = 11; maximum = 17; mean = 14.5; median = 15; base = 15.

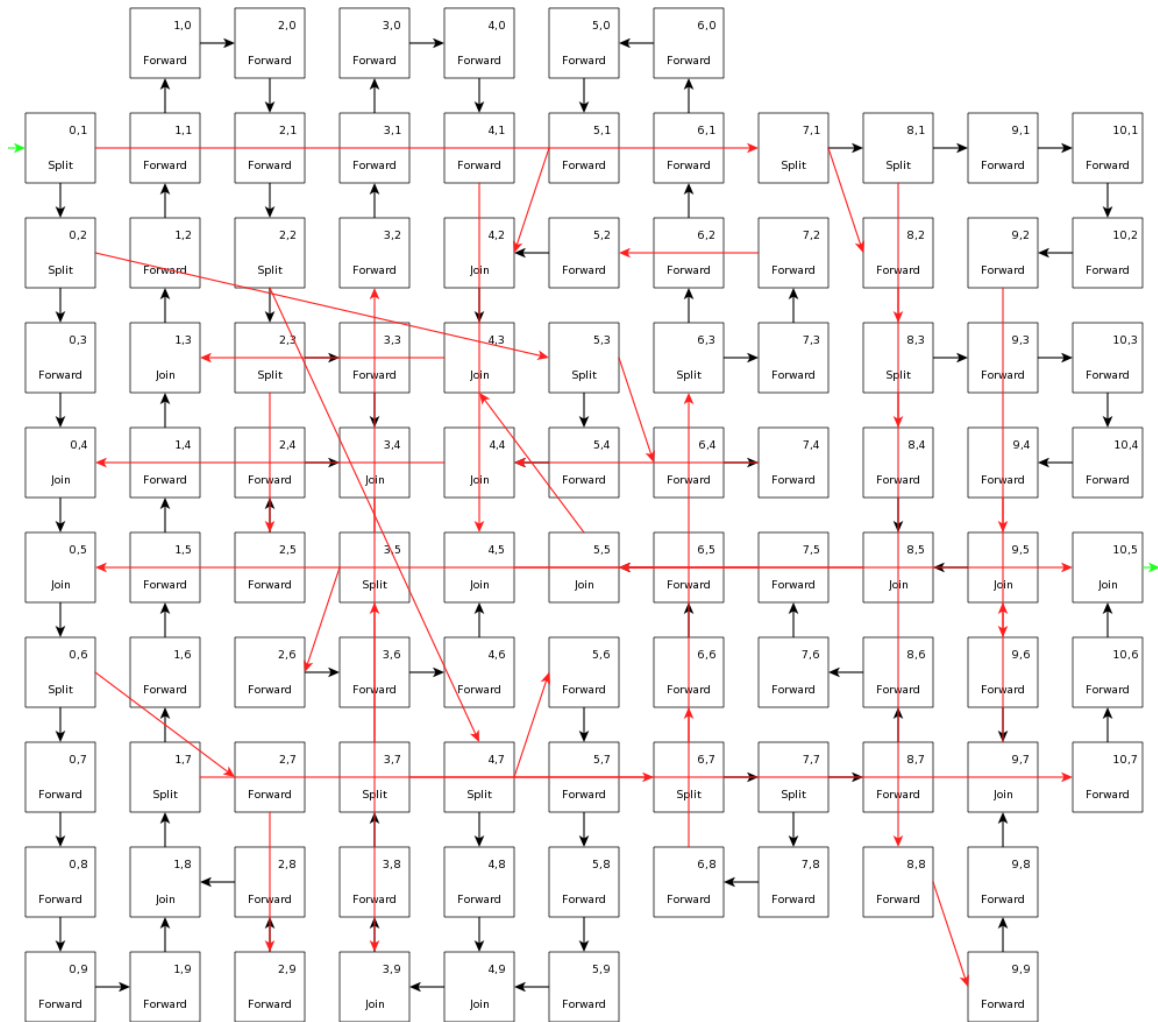


Figure 6.47: Best automatic mapping for the 100 random nodes application after excluding each processor individually from the target 25x25 array (using 100 trials for each excluded processor) while targeting the second version of AsAP. This automated mapping excludes processor (0, 0), has a rectangular array area of 110 (11x10), and uses 34 long-distance interconnects.

6.4.3 802.11a Wireless Transmitter

Settings: Defaults + “InputEdge = Left”

The objective for this test is to perform more trials on a smaller application in order to obtain higher resolution plots. These higher resolution plots show how the mapping algorithm will converge when a very large number of trials are performed. For this test each processor is again excluded one-by-one in sequence except that this time the array size is 8x8 instead of 25x25. Also for each excluded processor 1000 trials are performed instead of just 100 trials. The base and minimum values are instead obtained from these blocks of 1000 trials. Next, the mean and median values are calculated from the minimum values. Like before the array input is allowed to float along the left edge of the array to lower the mapping complexity.

Figure 6.48 shows the minimum rectangular array area for each excluded processor plotted as a color-coded 2D-array. Figure 6.49 plots the minimum rectangular array area as a histogram and includes tick marks for the mean, median, and base values. Figure 6.50 plots the cumulative distribution function for the minimum rectangular array area and again includes tick marks for the mean, median, and base values. Figure 6.51 shows the best automated mapping for the 802.11a wireless transmitter application, while targeting the first version of AsAP, and excludes processor (7, 6). Plots are not necessary for the minimum number of long-distance interconnects since this test targets the first version of AsAP, which does not allow long-distance interconnects. Also plots are not needed for the minimum number of routing processors since the application can be mapped without routers (in at least one trial) for every excluded processor.

The results from this test are quite a bit different than the results from the previous tests. The 2D-array plot looks more like what we expect. When the excluded processor is near the upper-left corner of the array the application is more difficult to map. When the excluded processor is near the bottom and right perimeters of the array the excluded processor is far enough out that it doesn't interfere with the mapping. The histogram shows that the mean and base values are again very close, but this time the base value is the absolute lowest value. This confirms that the application is mostly unaffected by a single excluded processor and also that the best mapping is the one with no excluded processors. The cumulative distribution function shows that by using a rectangular array area of 30 there is a 100% chance that the mapping will be successful with one excluded processor. Comparing this automated mapping to the previous automated mapping in Figure 6.1 we notice that the layout is somewhat different but both automated mappings have the same rectangular array area and both use no routing processors.

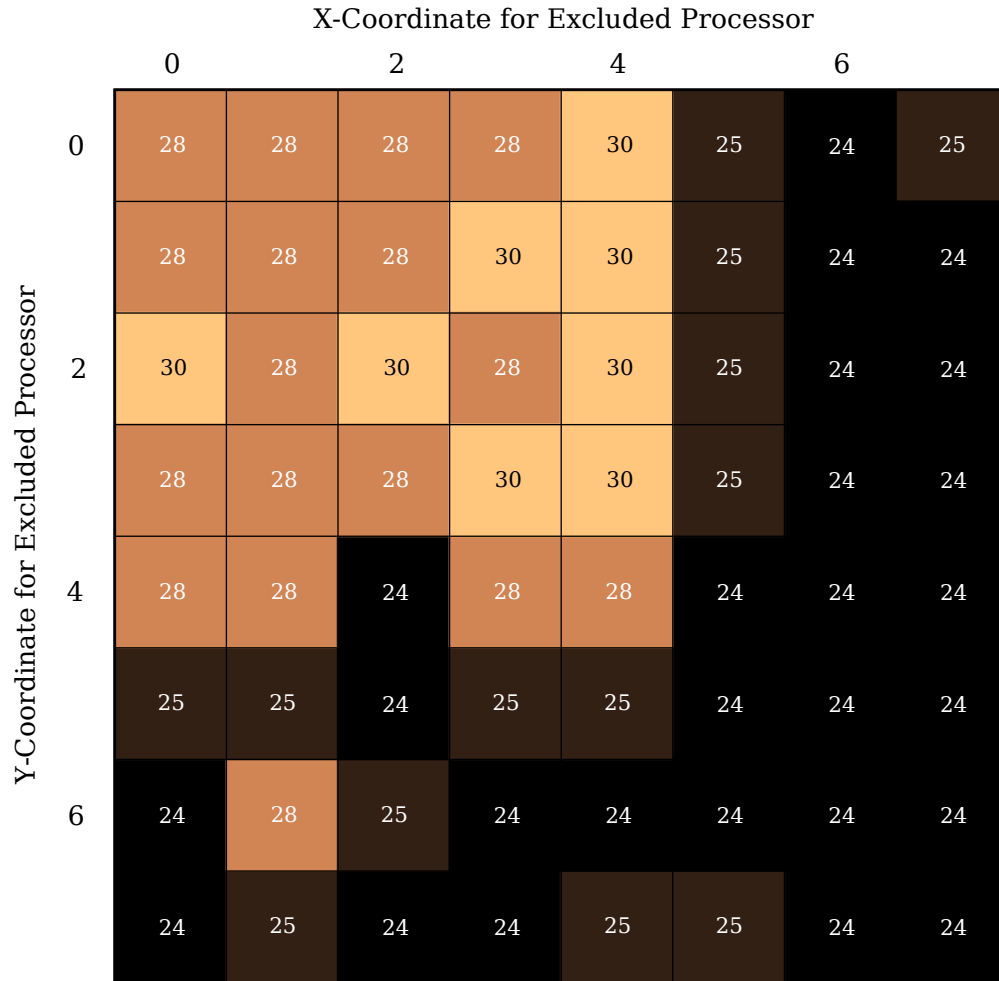


Figure 6.48: 2D-plot of the minimum rectangular array area when excluding each processor individually from the target 8x8 array (using 1000 trials for each excluded processor) for the 802.11a wireless transmitter application targeting the first version of AsAP. The square with the darkest color has the lowest rectangular array area. The rectangular array area increases as the square lightens in color. The statistics for this test regarding the minimum rectangular array area are: minimum = 24; maximum = 30; mean = 26; median = 25; base = 24.

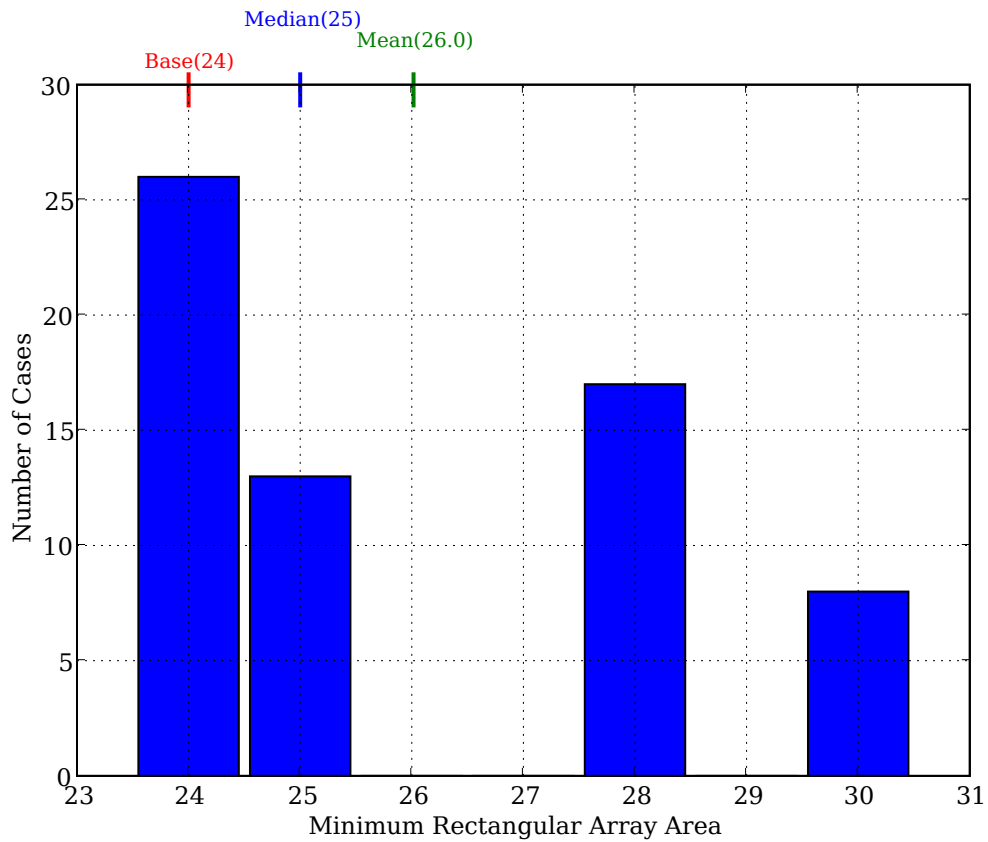


Figure 6.49: Histogram of the minimum rectangular array area, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 8x8 array (using 1000 trials for each excluded processor) for the 802.11a wireless transmitter application targeting the first version of AsAP. The statistics for this test regarding the minimum rectangular array area are: minimum = 24; maximum = 30; mean = 26; median = 25; base = 24.

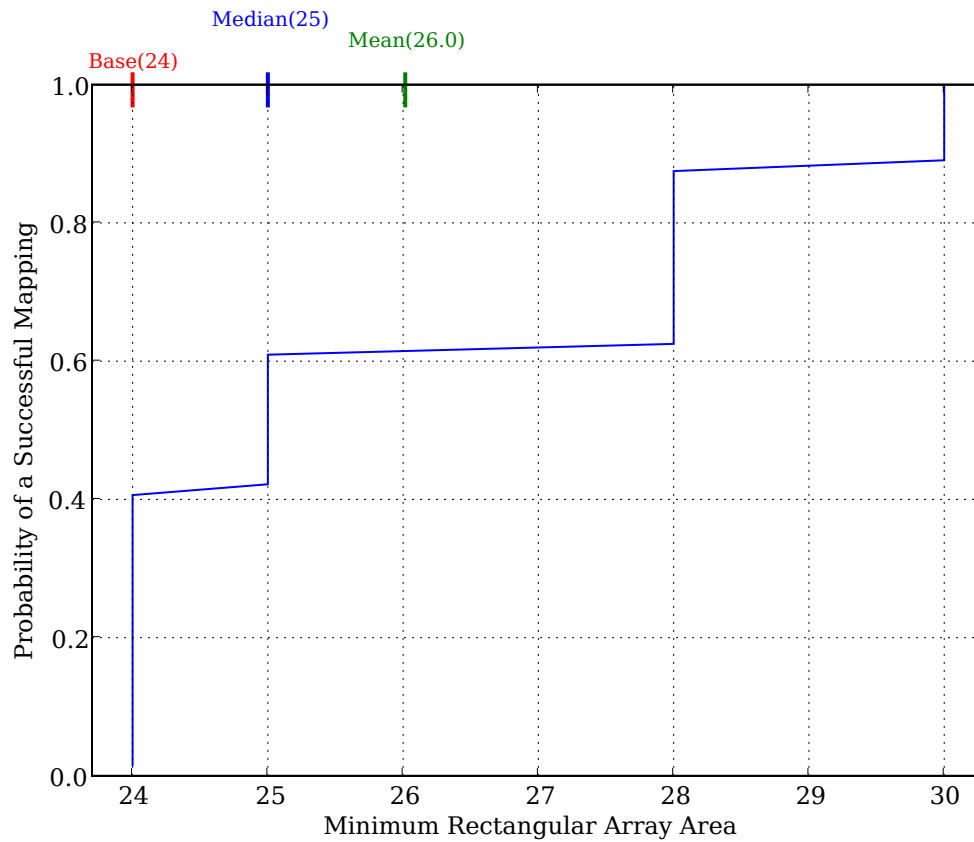


Figure 6.50: Cumulative Distribution Function for the minimum rectangular array area, along with the base value (no excluded processors), the median value, and the mean value, when excluding each processor individually from the target 8x8 array (using 1000 trials for each excluded processor) for the 802.11a wireless transmitter application targeting the first version of AsAP. The statistics for this test regarding the minimum rectangular array area are: minimum = 24; maximum = 30; mean = 26; median = 25; base = 24.

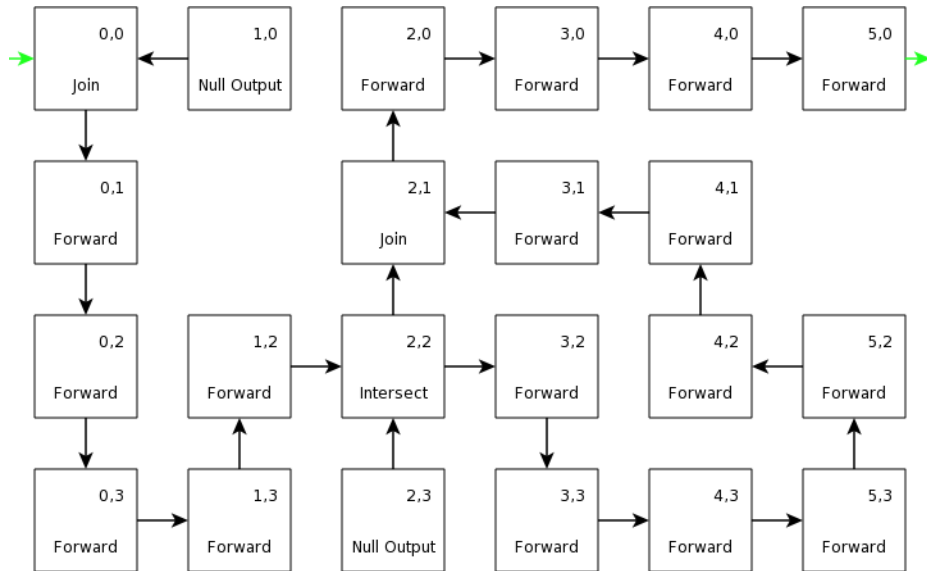


Figure 6.51: Best automatic mapping for the 802.11a wireless transmitter application after excluding each processor individually from the target 8x8 array (using 1000 trials for each excluded processor) while targeting the first version of AsAP. This automated mapping excludes processor (7, 6) and has a rectangular array area of 24 (6x4).

6.4.4 Multiple Exclusions

Settings: Defaults + “InputEdge = Left”

The 802.11a wireless transmitter application was chosen for the multiple exclusion tests because it’s the easiest to map and can therefore tolerate the most processor failures. Three tests are performed on this application. The first test contains 100 unique sets of 10 randomly selected excluded processors. The second test contains 100 unique sets of 20 randomly selected excluded processors. Finally, the third test contains 100 unique sets of 30 randomly selected excluded processors. Excluded processors are chosen randomly from an array of size 10x10. For each set of excluded processors 100 trials are performed. From these blocks of 100 trials the base and minimum values are obtained. The mean and median values are calculated from the minimum values just like before. Routing processors are used, but their exact number is not so important. We are more interested in the rectangular array area. Since these tests target the first version of AsAP all the mappings must be nearest neighbor only and therefore use no long-distance interconnects.

Figure 6.52 shows that the mean rectangular array area with 10 excluded processors is about 36. Figure 6.53 shows that the mean rectangular array area with 20 excluded processors is about 46. Finally, Figure 6.54 shows that the mean rectangular array area with 30 excluded processors is about 59. From these three figures we can also see that the rectangular array area

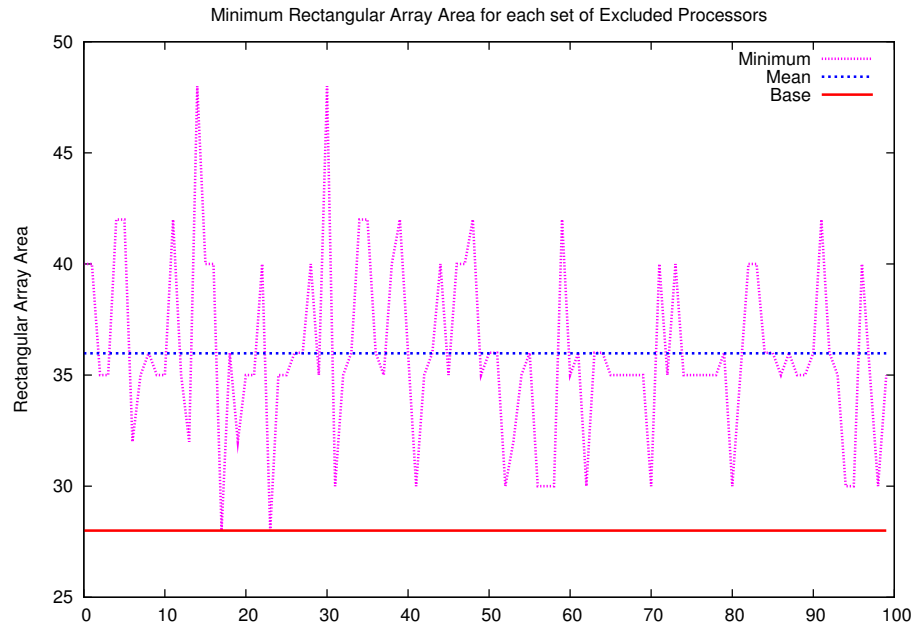


Figure 6.52: Plot of the minimum rectangular array area for each set of 10 excluded processors, the mean rectangular array area across all minimums, and the base rectangular array area (where no processors are excluded) for the 802.11a wireless transmitter targeting the first version of AsAP. The statistics for this test regarding the minimum rectangular array area are: mean = 36.0; base = 28.

without any excluded processors, also called the base rectangular array area, is equal to 28. The mean rectangular array area increases each time the number of excluded processors increases. This is of course expected since more routing processors are required to get around the faulty processors, thereby increasing the rectangular array area. The range of numbers along the x-axis decreases as the number of excluded processors increases, indicating that fewer mappings were successful as the number of excluded processors increased. When testing sets of 10 excluded processors the application was mappable using all 100 sets. When testing sets of 20 excluded processors the application was mappable using only 94 of the 100 sets. When testing sets of 30 excluded processors the application was mappable using only 73 of the 100 sets. If significantly more trials were performed it's possible that all 100 sets in all three tests would be mappable. The mean values would also likely decrease as better solutions would be available for each set of excluded processors.

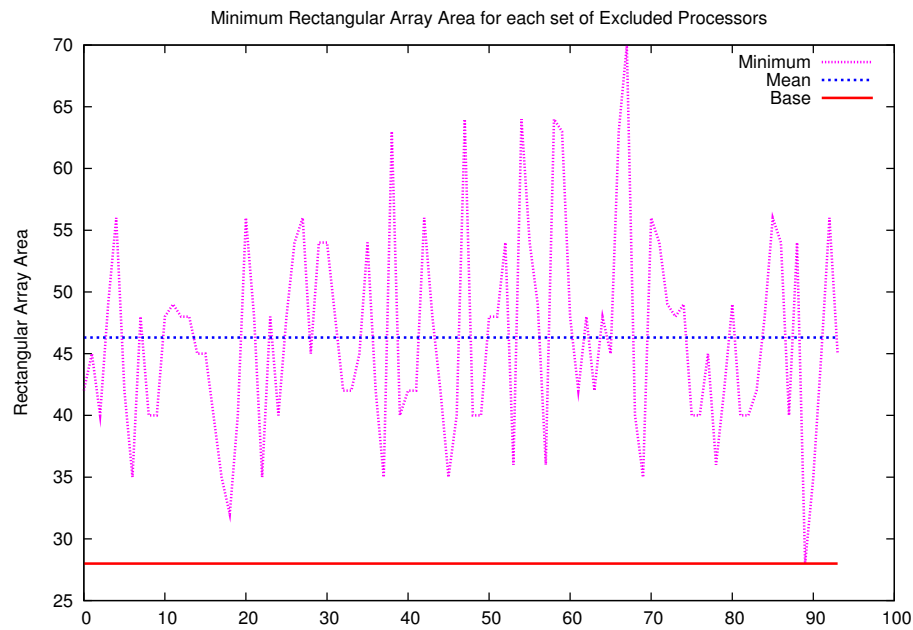


Figure 6.53: Plot of the minimum rectangular array area for each set of 20 excluded processors, the mean rectangular array area across all minimums, and the base rectangular array area (where no processors are excluded) for the 802.11a wireless transmitter targeting the first version of AsAP. The statistics for this test regarding the minimum rectangular array area are: mean = 46.3; base = 28.

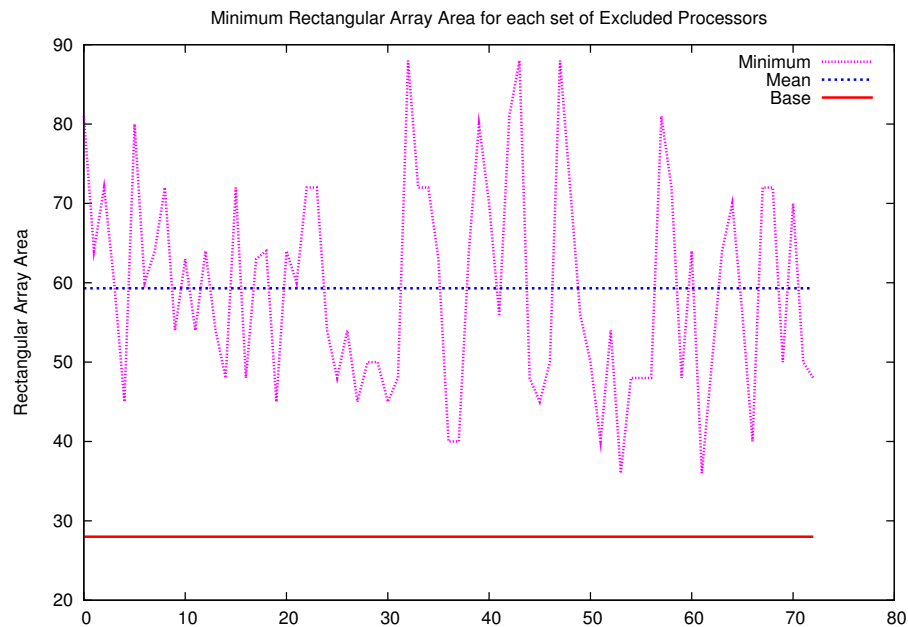


Figure 6.54: Plot of the minimum rectangular array area for each set of 30 excluded processors, the mean rectangular array area across all minimums, and the base rectangular array area (where no processors are excluded) for the 802.11a wireless transmitter targeting the first version of AsAP. The statistics for this test regarding the minimum rectangular array area are: mean = 59.3; base = 28.

6.4.5 Summary

Upon analyzing the results from the five single exclusion tests, the mapping algorithm was able to tolerate minor fabrication errors with the exception of just a few cases. This is based on the observation that the mean values were close to the base values in every test. There was a noticeable decrease in rectangular array area for applications that were mapped to both the first and second versions of AsAP. This was due to a desirable trade-off between routing processors and long-distance interconnects. For the two larger applications only 100 trials were performed instead of the desired 1000 trials that were performed on the smaller application. This was primarily done to save time (which already took two weeks using 30 CPUs) but still allowed us to estimate how the mapping algorithm would handle fabrication errors when mapping larger applications. When 1000 trials were actually performed, the patterns observed were what we expected. An excluded processor near the periphery of the array is nearly identical to having no excluded processors at all. Also the base rectangular array area is equal to the lowest rectangular array area, which occurs when no processors are excluded. The reason why these observations do not hold true in some cases, and why we see the base value above the mean and median values, is with lower trial counts only a fraction of the solution space is explored relative to higher trial counts. Also larger more complex applications require additional trials in order to find mappings comparable in quality to the mappings found for smaller simpler applications. The solution is to perform more trials or possibly implement a more highly optimized algorithm.

After analyzing the results from the three multiple exclusion tests, the mean rectangular array area is indeed quite stable despite the large number of excluded processors. In the last test there are more excluded processors than nodes in the application. In Figure 6.55, which plots the mean rectangular array area with respect to the number of excluded processors, the mean rectangular array area grows linearly as opposed to exponentially. This indicates that the growth is stable. The coefficient for the best-fit line shows an increase in mean rectangular array area of about 1.04 nodes for each additional processor failure. This basically indicates that no additional routing processors are need to handle a new processor failure (up to some limit of course). Based on the previous efficiency results when mapping the 802.11a wireless transmitter application, 100 trials could be executed in approximately one minute. The mapping tool could quickly re-map this application if a new processor failure were to occur. This would be very challenging to do manually in less than one minute. These observations indicate that the mapping algorithm can tolerate numerous fabrication errors when working with simple applications.

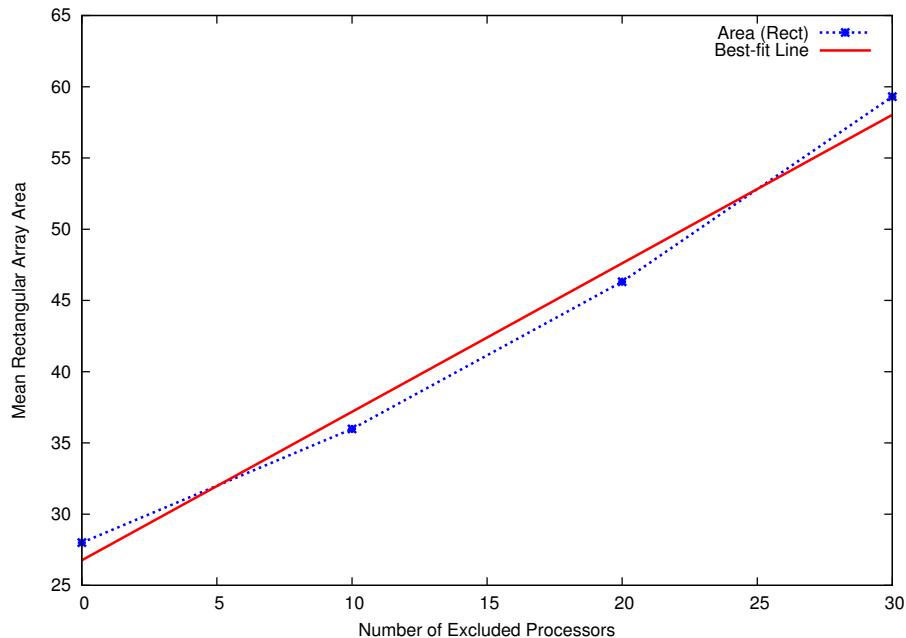


Figure 6.55: Plot of the minimum rectangular array area with respect to the number of excluded processors for the 802.11a wireless transmitter

6.5 Fabrication Differences

Different processors from the same chip will perform differently due to variations in transistor characteristics as a result of fabrication. These fabrication differences affect the mapping problem in ways similar to fabrication errors. For fabrication errors the goal is to avoid placing tasks onto certain processors. For fabrication differences the goal is to place certain tasks onto certain processors to improve performance or reduce power consumption. Placing tasks with a greater workload onto processors that have a higher maximum frequency may provide opportunities to improve the performance of an application. Similarly, placing less active tasks onto higher leakage current processors may provide opportunities to reduce the power consumption for an application. The goal is to quantify these scenarios, associate them with a cost, then minimize this cost in order to improve the application's performance and power consumption. This is done by adding annotation values to each task and each processor, which help guide the user cost function.

The two applications tested in this section are the 802.11a wireless transmitter application and the Viterbi decoder application, because of their simplicity. Unlike previous tests the desired array size is pre-defined instead of being calculated automatically. The target array is defined to be 10x10 even though the required rectangular array area is known to be only 6x6. This was done intentionally to increase task flexibility, allowing tasks to migrate to their optimal location.

To evaluate the effectiveness of optimizing for speed and power, each application is mapped both with and without annotations. Each application is first mapped without annotations then mapped three times with annotations. Each time different annotations are enabled. First just speed related annotations are enabled, next just power related annotations are enabled, and finally both speed and power related annotations are enabled. For each test 1000 trials are performed (random seeds from 1 to 1000). The quality of the unannotated mapping is then compared to the quality of the three annotated mappings. Mappings are compared both visually and numerically. Mappings are compared visually by checking the placement of key tasks and the use of key processors. Mappings are compared numerically using simple equations (Equation 6.1 and Equation 6.2) discussed later in this section.

6.5.1 Value Annotations

Before we can calculate performance and power consumption costs, annotation values must be added to each processor within the target array and each task within the application. Two annotation values are added to each processor within the target array. The first value is the *maximum frequency*, which estimates a processor's maximum clock frequency. The second value is the *leakage current*, which estimates a processor's leakage current. Two annotation values are added to each task within the application. The first value is the *load average*, which estimates the number of operations performed for each sample, or unit, processed. A task with a main loop containing 50 instructions would have a load average equal to 50. The second value is the *activity level*, which estimates the percentage of time a task is active. A task that is active 100% of the time would have an activity level equal to 100. A custom user function reads in these annotation values and calculates the cost associated with the current positioning of tasks relative to the processors they have been assigned.

Due to physical differences created during fabrication, regions of faster and slower maximum frequency, and regions of higher and lower leakage current, are formed in various parts of the target array. The annotation values entered for the target array are not based on real measured data. These annotation values were instead generated by hand (as they were typed into the datafile) to demonstrate some of the benefits of optimizing for fabrication differences. The hand generated maximum frequency map in Figure 6.56 shows a cluster of faster processors in the center of the array, capable of 500 MHz, and various clusters of slower processors around the border of the array, capable of only 400 MHz. The hand generated leakage current map in Figure 6.57 shows that processors in the upper-left and lower-right regions tend to leak more current, 30 mA, while processors near the

lower-left tend to leak less current, 10 mA. The data from these two figures is stored in a datafile processed by the ASAP mapping tool and passed to the custom user function. Using real measured data and verifying mappings against the physical chip are left as future work.

Each task must be annotated in order to determine which task is best suited for each processor. The annotation values entered for each task are not based on real simulation data. These values were instead generated by hand using general assumptions and visually inspecting the code inside each task. The hand generated map in Figure 6.58 shows the load average, which is the top color, and the activity level, which is the bottom color, for each task in the 802.11a wireless transmitter application. Each task has been numbered to make visual comparisons easier. For the 802.11a wireless transmitter application the IFFT, which includes tasks 3 - 6 and tasks 8 - 11, performs the bulk of the work. These tasks remain active once started and have the highest throughput requirements. The hand generated map in Figure 6.59 shows the load average and activity level for each task in the Viterbi decoder application. For the Viterbi decoder application the traceback cycle, which includes tasks 14 - 18, performs the bulk of the work and runs all the time once started. The data from these two figures is stored in the module files, which are processed by the ASAP mapping tool and passed to the custom user function. Using real simulated data is left as future work.

Processors with a higher maximum frequency typically have higher leakage currents since these two characteristics are typically associated with transistors that have a higher V_t (threshold voltage). Making matters worse, tasks with a higher load average typically have higher activity levels since these tasks usually have the most data to process. This causes somewhat of a dilemma when trying to optimize for both speed and power simultaneously. Moving a task with a higher load average and higher activity level to a processor with a higher maximum frequency and higher leakage current will reduce performance costs but increase power consumption costs. Moving this same task to a processor with a lower maximum frequency and lower leakage current will reduce power consumption costs but increase performance costs. These types of tasks are best handled by *critical processors*. A critical processor is a processor that has a higher maximum frequency but also has a lower leakage current. The critical processors for the target array are processors (3, 5), (3, 6), (4, 5), and (4, 6) from Figure 6.56 and Figure 6.57. These processors are of course very desirable, but they should ideally be reserved for *critical tasks*. A critical task is a task that has a higher load average and also a higher activity level. The critical tasks for the 802.11a wireless transmitter application are tasks 4, 6, 8, and 10 from Figure 6.58. The critical tasks for the Viterbi

Maximum Frequency Map

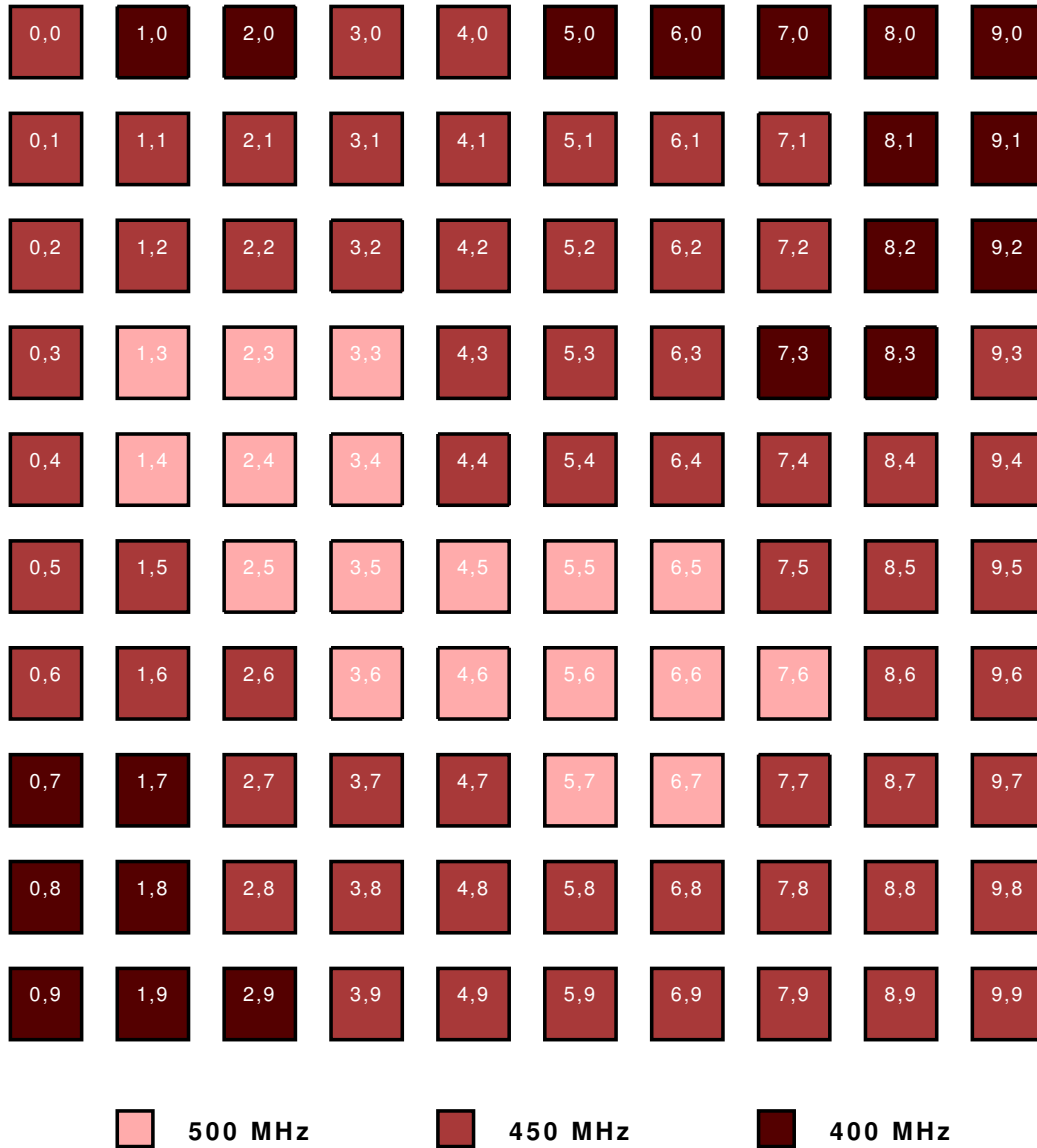


Figure 6.56: Maximum frequency value for each processor in the target 10x10 array

decoder application are tasks 14, 15, 16, 17, and 18 from Figure 6.59. The mapping with the highest performance and the lowest power consumption will likely have every critical processor assigned a critical task.

Leakage Current Map

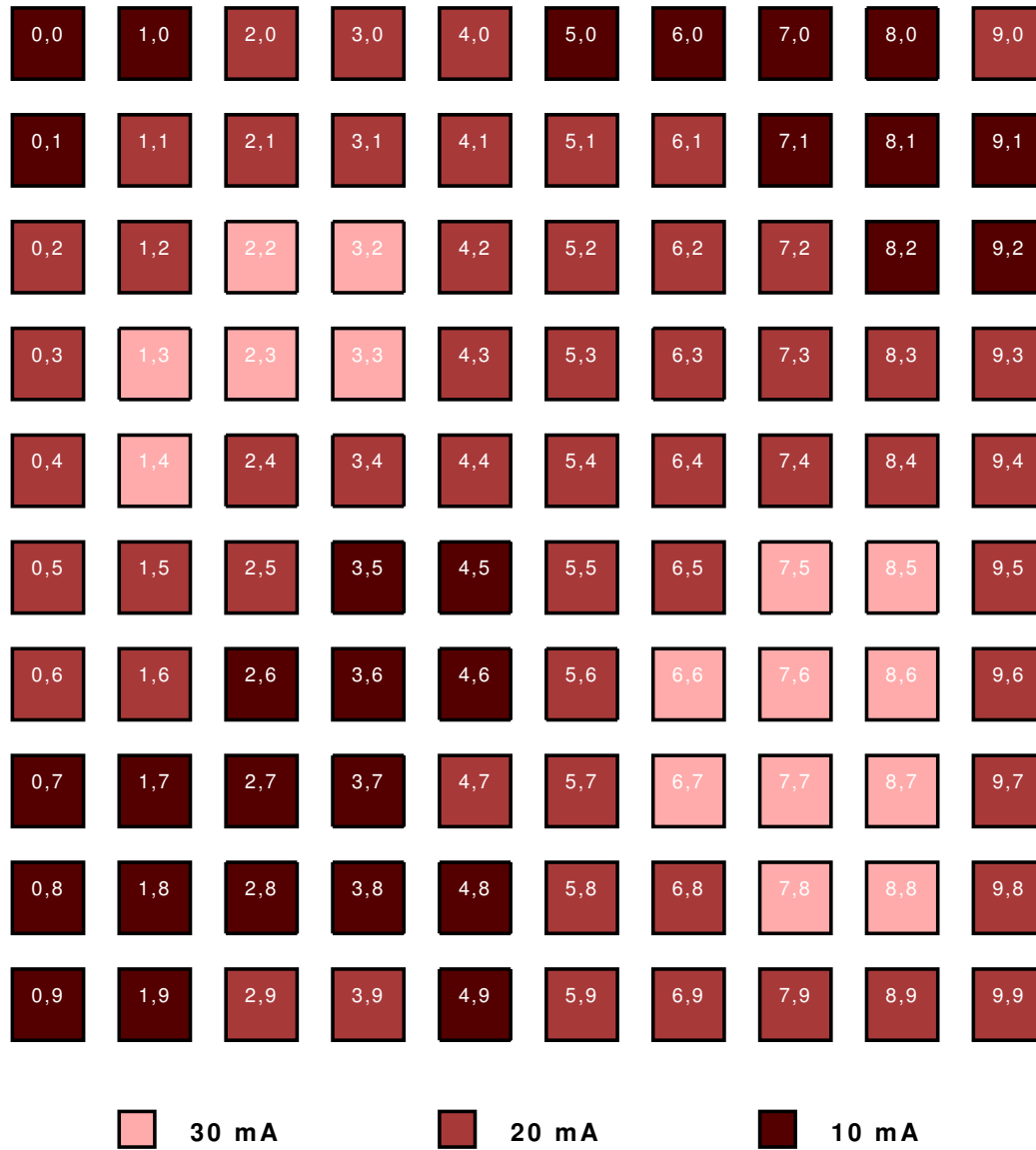


Figure 6.57: Leakage current value for each processor in the target 10x10 array

802.11a Tx LoadAvg/Activity Map

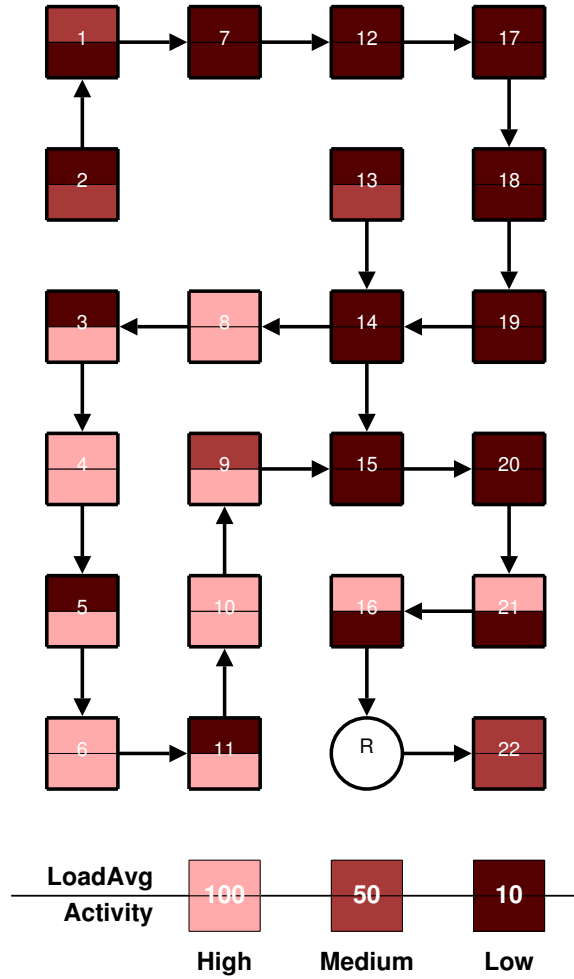


Figure 6.58: Load average and activity level for each task in the 802.11a wireless transmitter application

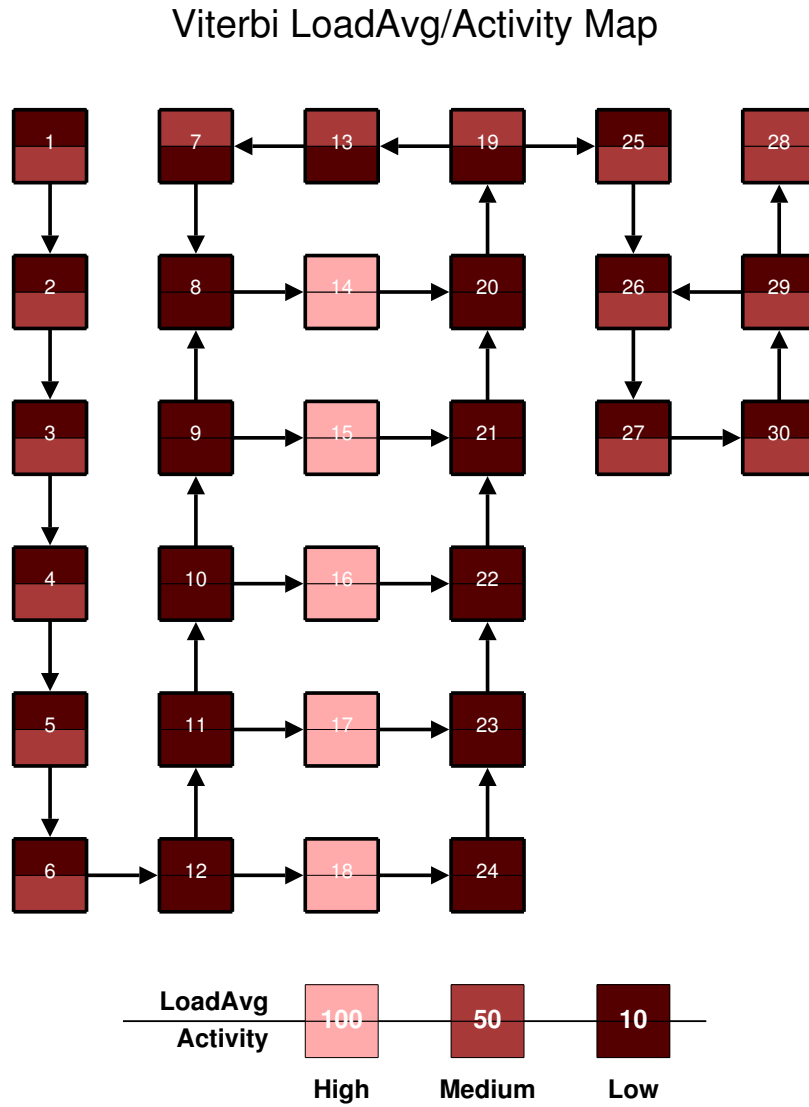


Figure 6.59: Load average and activity level for each task in the Viterbi decoder application

6.5.2 Custom User Function

The custom user function is an advanced feature of the mapping tool. The custom user function allows a programmer to augment the configuration cost, which is used extensively during the placement phase to optimize mappings. An unlimited number of values can be associated with each task and each processor. The meaning of these values is determined by their use within the custom user function. The custom user function implemented in this work has two goals. The first goal is to place tasks with a higher load average onto processors that have a higher maximum frequency. The second goal is to place tasks with a higher activity level onto processors that have a lower leakage current. These two goals are achieved using specialized equations that produce lower values when the configuration is more desirable. The contents of this custom user function are listed in Algorithm 6.1 using the same typefaces and notations as the code listings in Chapter 3. The value returned by the custom user function is added to the final configuration cost near the end of the `ConfigCost` function (listed in Algorithm 3.4). The equations on lines 12 and 13 of Algorithm 6.1 perform the actual optimizations using the annotation values. The equation on line 12 optimizes for speed, and the equation on line 13 optimizes for power. These two equations produce whole numbers in the range of 1 to 100 for the annotation values shown previously. Line 14 determines which optimizations are enabled by choosing which results from line 12 and 13 are added to the total cost. The custom user function is integrated into the mapping algorithm so applications are optimized for both fabrication differences as well as the other primary metrics.

Algorithm 6.1 UserCost

```

1: let Cost ← 0
2: for each Vertex in Graph do
3:   let Coord ← Vertex.Coordinate
4:   if Coord.X ≥ Graph.Size.X or Coord.Y ≥ Graph.Size.Y then
5:     let Cost ← Cost + 1000
6:   next iteration
7: end if
8: let LoadAvg ← load average user data for Vertex
9: let Activity ← activity level user data for Vertex
10: let Frequency ← maximum frequency user data for Coord
11: let Leakage ← leakage current user data for Coord
12: let Costspeed ← (LoadAvg × 30) / (Frequency - 350)
13: let Costpower ← (Activity × Leakage) / 50
14: let Cost ← Cost + Costspeed + Costpower
15: end for
16: return Cost

```

6.5.3 Numerical Evaluation

Performance and power consumption are evaluated numerically using two equations, which roughly estimate the *sample latency* and *leakage power*. The sample latency is the time required to process a sample assuming the sample is passed to every task only once in sequence (similar to a software pipeline). The leakage power is the amount of power consumed due to leakage currents assuming each task is active a constant percentage of time and power is removed from a task when it's inactive. These two values help to compare different mappings and also show more concrete benefits to mapping an application with annotations. The first equation, Equation 6.1, computes the sample latency for a mapping. The assumption is that the load average is the number of instructions executed for each sample processed. The time required to process a sample can be estimated by dividing the number of instructions by the maximum clock frequency then doing this for every task and summing the values. The second equation, Equation 6.2, computes the leakage power for a mapping. The voltage is assumed to be 1.8 V since that's the nominal operating voltage for the first version of AsAP (the nominal operating voltage for the second version of AsAP is not yet known). The leakage power can be estimated by multiplying together the voltage, the leakage current, and the percentage of time the task is active then doing this for every task and summing the values. Mappings can be compared numerically using these two values in addition to being compared visually using critical processor and critical task locations.

$$\text{Sample Latency} = 1000 \times \text{LoadAvg} / \text{Frequency} \quad ns \quad (6.1)$$

$$\text{Leakage Power} = 1.8 \times \text{Leakage} \times (\text{Activity} / 100) \quad mW \quad (6.2)$$

6.5.4 802.11a Wireless Transmitter

Settings: Defaults + “AddSpacing = False”

The 802.11a wireless transmitter application has only 22 processors, which provides plenty of free space within the target array to optimize the application for speed and power. The spacing insertion flag has been disabled to keep the array from shifting unexpectedly during the routing phase, which would leave the processors misaligned. The annotation values used for these tests are shown in Figure 6.58. When mapping the application without annotations, the mapping chosen for comparisons is the mapping with the lowest *optimization cost*, similar to the previous sections. The optimization cost is calculated by Equation 5.4, which evaluates a mapping using the metrics: area, communication, and utilization. Since the optimization cost equation doesn't take into ac-

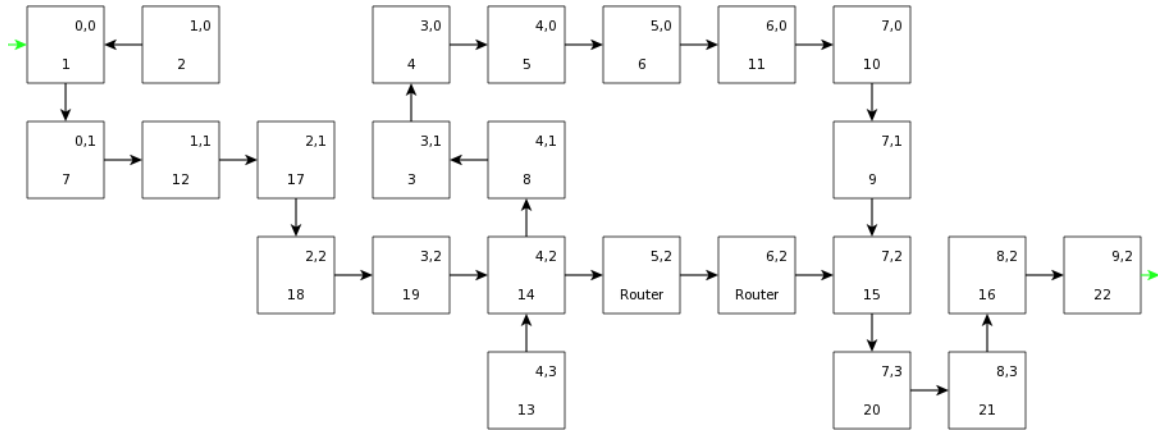


Figure 6.60: Automatic mapping without using any annotations for the 802.11a wireless transmitter

count annotations, it's effectively useless when comparing annotated mappings. When mapping the application with annotations, the mapping chosen for comparisons is the mapping with the lowest *configuration cost*. The configuration cost is calculated by the `ConfigCost` function (listed in Algorithm 3.4), which does take into account annotations. The final configuration cost is display by the mapping tool upon completion of the placement phase, which can be extracted from the log file. For all tests, both with and without annotations, 100 trials are first executed before selecting the best mapping for comparisons.

For this first mapping no annotations were included. The mapping shown in Figure 6.60 has a reasonably small rectangular array area (10x4) and uses only 2 routing processors. Though upon visual inspection none of the tasks, let alone critical tasks, were placed onto critical processors. None of the higher maximum frequency processors were even used. In terms of leakage power this mapping was decent. Only 2 tasks were assigned to higher leakage current processors. Upon numerical inspection the estimated sample latency was 3000 ns and the estimated leakage power was 440 mW. These numbers don't mean much by themselves but will help to compare this unannotated mapping to the following annotated mappings.

For this second mapping only speed related annotations were included, in particular the maximum frequency and the load average. With speed related annotations included we should notice an increase in performance. The mapping shown in Figure 6.61 has a larger rectangular array area (10x9), compared to the previous mapping, and uses 6 routing processors. Upon visual inspection every critical processor has been assigned a task. Even better, critical processor (3, 6) was assigned critical task 8. Approximately 75% of the tasks were assigned to higher maximum frequency processors, which is expected when trying to optimize for speed. Although 6 tasks were

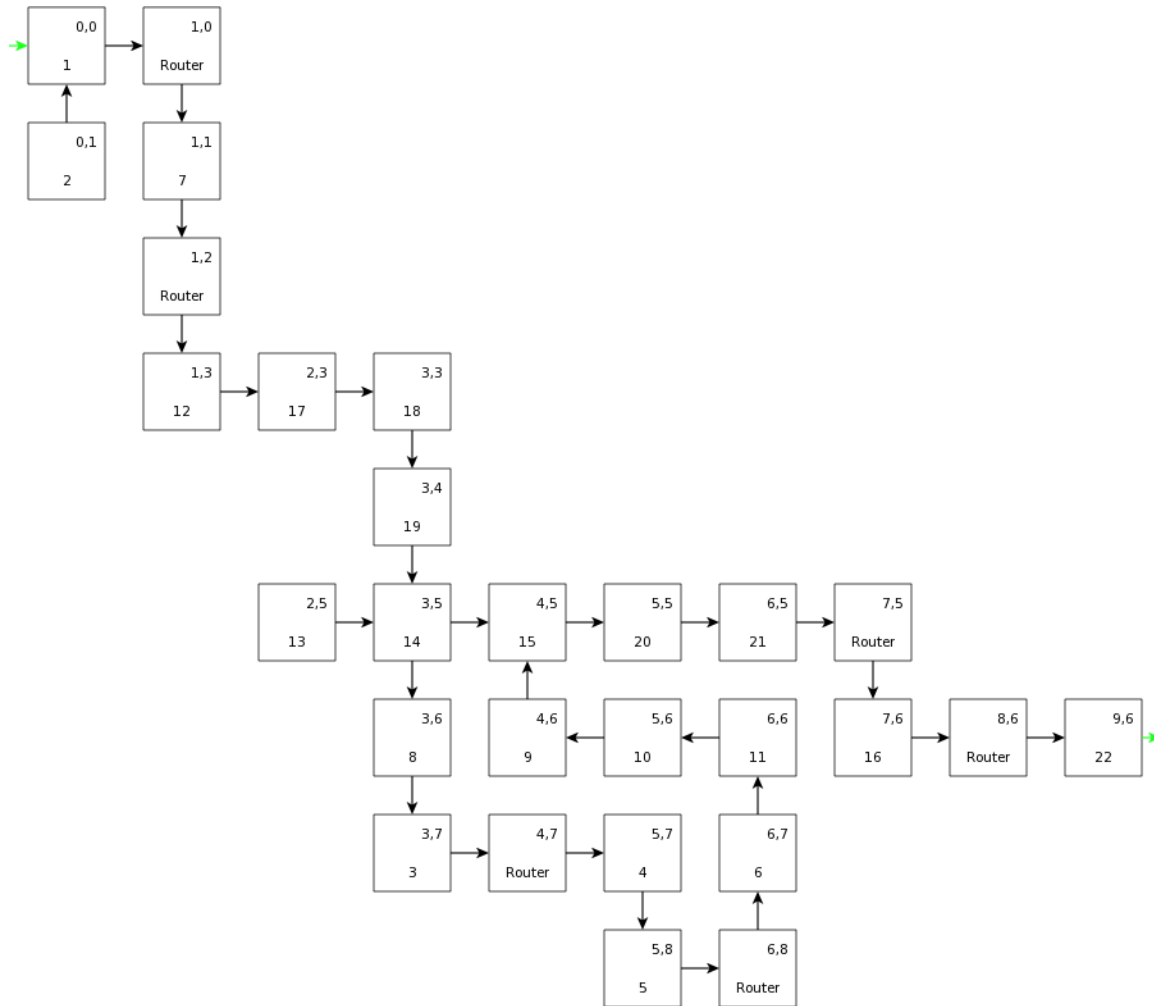


Figure 6.61: Automatic mapping using just speed related annotations for the 802.11a wireless transmitter

assigned to higher leakage current processors. Upon numerical inspection the estimated sample latency was 2610 ns and the estimated leakage power was 490 mW. This mapping, being about 13% faster, clearly has better performance than the unannotated mapping, but the power consumption has increased by about 11%.

For this third mapping only power related annotations were included, in particular the leakage current and the activity level. With power related annotations included we should notice a decrease in power consumption. The mapping shown in Figure 6.62 has a rectangular array area of 10x10, which is the maximum size allowed, and uses 12 routing processors. Upon visual inspection 3 of the 4 critical processors were assigned tasks, but none of these were critical tasks. A little over 80% of the tasks were assigned to lower leakage current processors, which is very important when optimizing for power. Although 7 tasks were assigned to lower maximum frequency processors.

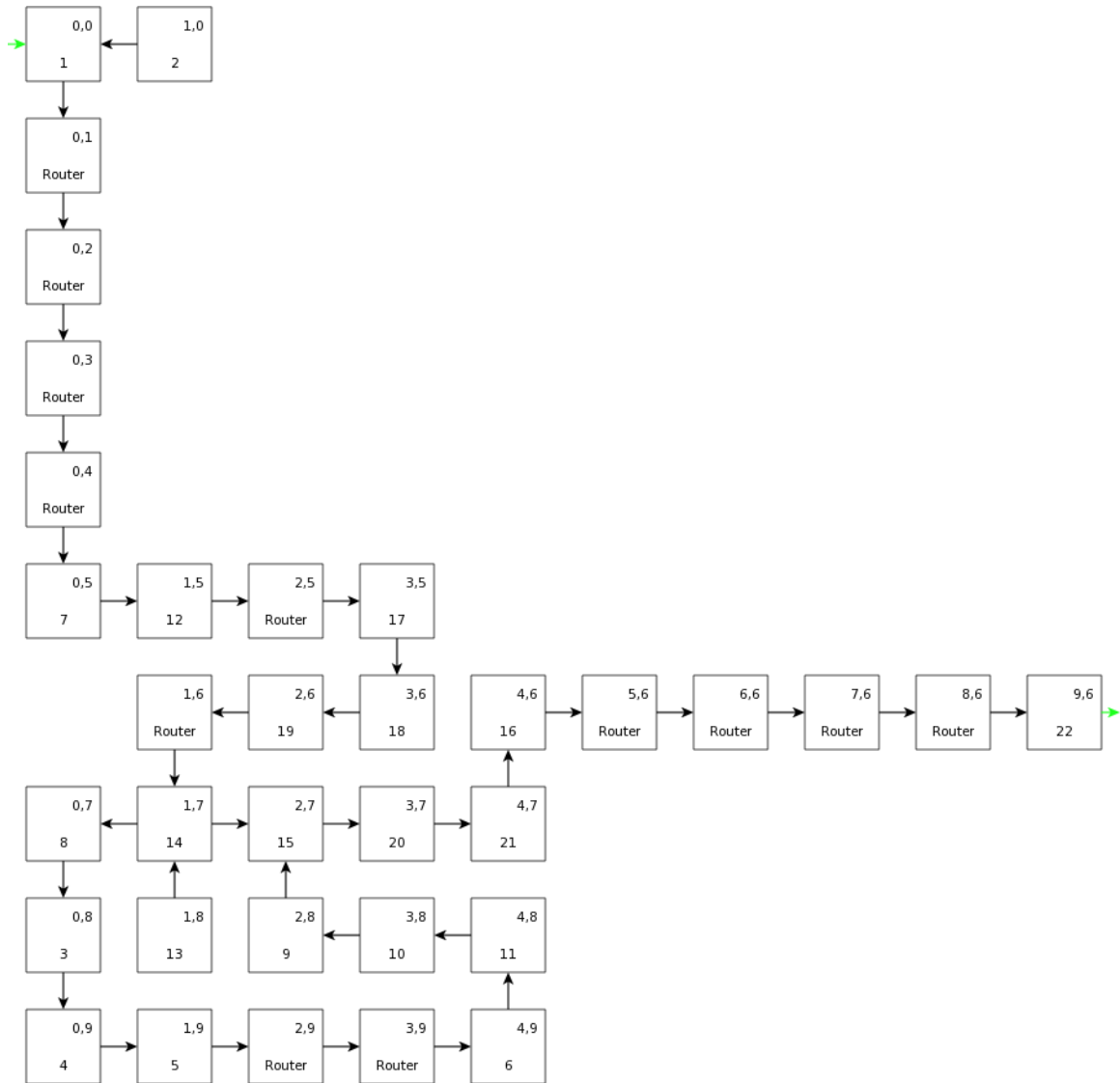


Figure 6.62: Automatic mapping using just power related annotations for the 802.11a wireless transmitter

Upon numerical inspection the estimated sample latency was 2930 ns and the estimated leakage power was 280 mW. With a decrease in leakage power of about 36%, this mapping clearly consumes less power than the unannotated mapping. In addition it's even slightly faster.

For the final mapping both speed and power related annotations were included. With annotations fully included we should notice improvements in both performance and power consumption. The mapping shown in Figure 6.63 has a rectangular array area of 10x9 and uses 7 routing processors. Upon visual inspection every critical processor was assigned a task. Critical processor (3, 5) was even assigned critical task 8. None of the tasks were assigned to lower maximum frequency processors and 9 tasks were assigned to higher maximum frequency processors. Also none

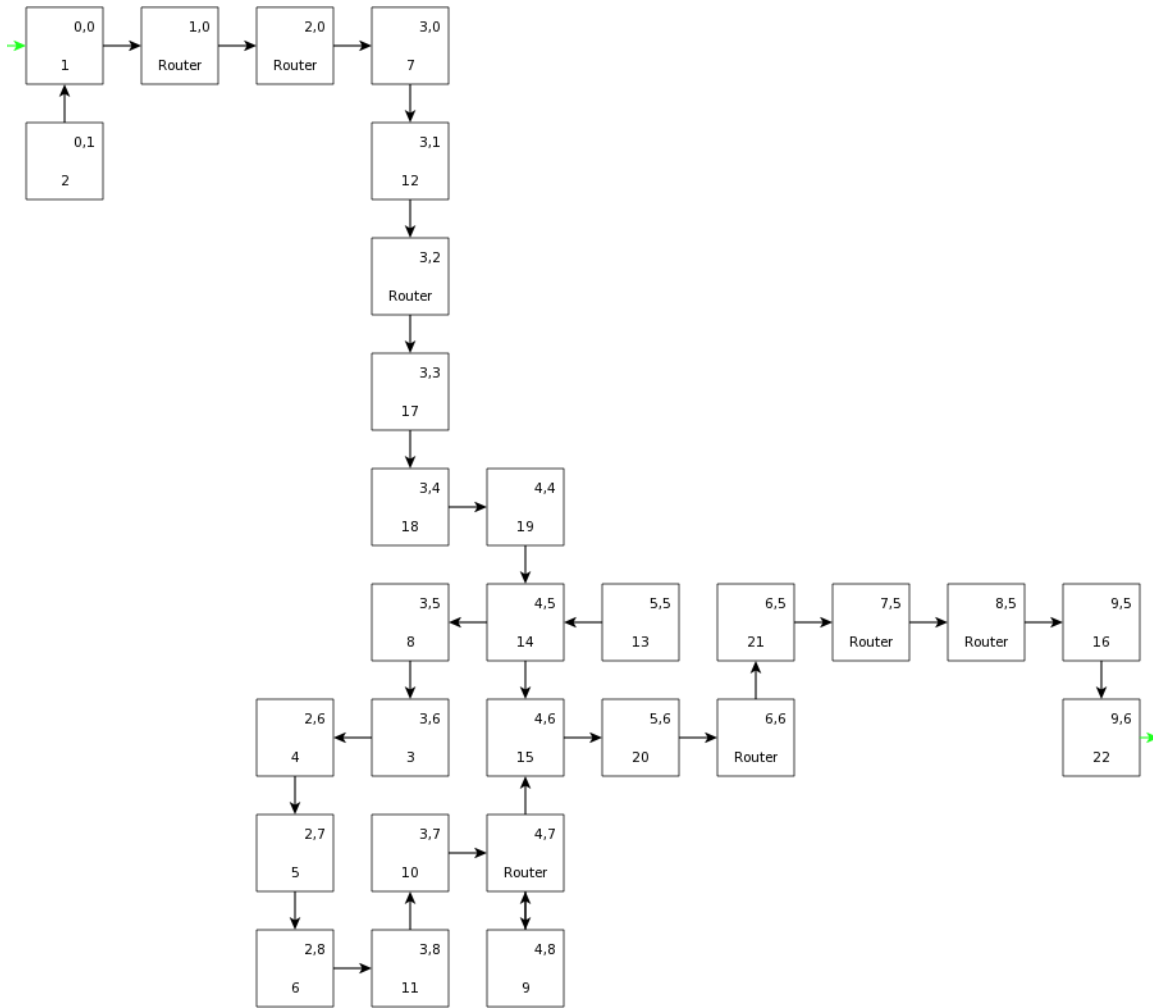


Figure 6.63: Automatic mapping using both speed and power related annotations for the 802.11a wireless transmitter

of the tasks were assigned to higher leakage current processors and 12 tasks were assigned to lower leakage current processors. Upon numerical inspection the estimated sample latency was 2720 ns and the estimated leakage power was 330 mW. This mapping is not only about 9% faster than the unannotated mapping, but it also consumes 25% less power due to the annotations.

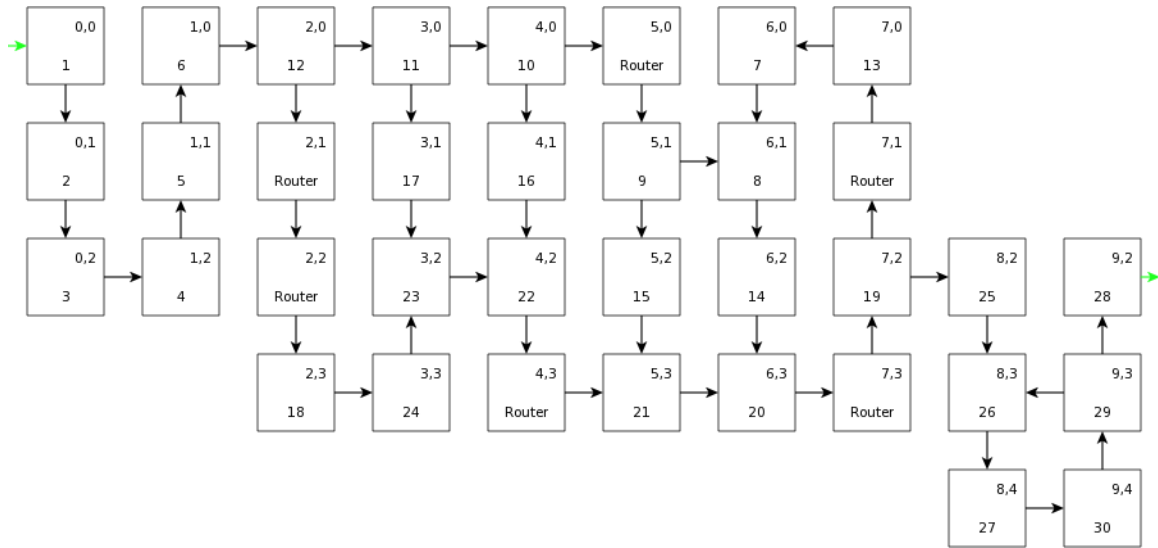


Figure 6.64: Automatic mapping without using any annotations for the Viterbi decoder

6.5.5 Viterbi Decoder

Settings: Defaults + “AddSpacing = False”

The Viterbi decoder application, having 30 processors, still has plenty of free space within the target array to optimize the application for speed and power. Once again the spacing insertion flag has been disabled to keep processors from becoming misaligned during the routing phase. The annotation values used for these tests are shown in Figure 6.59. When mapping the application without annotations, the mapping chosen for comparisons is again the mapping with the lowest optimization cost. Due to deficiencies in the optimization cost equation, the configuration cost must be used when comparing annotated mappings. When mapping the application with annotations, the mapping chosen for comparisons is once again the mapping with the lowest configuration cost. For every test 100 trials are first executed before selecting the best mapping for comparisons.

For this first mapping no annotations were included. The mapping shown in Figure 6.64 has a reasonably small rectangular array area (10x5), and uses only 6 routing processors. Though upon visual inspection none of the tasks, let alone critical tasks, were placed onto critical processors. Only 2 of the higher maximum frequency processors were used. However, only 3 tasks were placed onto higher leakage current processors, which is quite impressive. Upon numerical inspection the estimated sample latency was 3560 ns and the estimated leakage power was 470 mW. Like before, these numbers mean very little by themselves but will help to compare this unannotated mapping to the following annotated mappings.

For this second mapping only speed related annotations were included, so we should notice an increase in performance. The mapping shown in Figure 6.65 has a larger rectangular array area (10x10), compared to the previous mapping, but uses only 4 routing processors. Upon visual inspection every critical processor was assigned a task and two of these tasks were the critical tasks 16 and 17. Over half of the tasks were assigned to higher maximum frequency processors and only 1 higher maximum frequency processor went unused. Although about one third of the tasks were assigned to higher leakage current processors, which is acceptable since we are optimizing for speed. Upon numerical inspection the estimated sample latency was 3300 ns and the estimated leakage power was 460 mW. This mapping is only about 7% faster than the unannotated mapping, but the goal for this test was to increase performance, which was accomplished. As an added bonus, this mapping also consumes less power despite the large number of higher leakage current processors that were used.

For this third mapping only power related annotations were included, so we should notice a decrease in power consumption. The mapping shown in Figure 6.66 has a rectangular array area of 10x10, and uses 12 routing processors. Upon visual inspection every critical processor was assigned a task and two of these tasks were the critical tasks 14 and 15. Exactly 60% of the tasks were assigned to lower leakage current processors, which includes every lower leakage current processor in the bottom-left region. However, 6 tasks were assigned to lower maximum frequency processors, which is not so important when optimizing for power. Upon numerical inspection the estimated sample latency was 3500 ns and the estimated leakage power was 280 mW. This mapping consumes about 40% less power than the unannotated mapping, which is significantly lower. This mapping is even marginally faster.

For the final mapping both speed and power related annotations were included, so we should notice improvements in both performance and power consumption. The mapping shown in Figure 6.67 has a rectangular array area of 10x10, which is the maximum size allowed, and uses 8 routing processors. Upon visual inspection every critical processor was assigned a task, and two of these tasks were the critical tasks 16 and 17. A majority of the higher maximum frequency processors were used and only 2 tasks were assigned to lower maximum frequency processors. Also 5 tasks were assigned to higher leakage current processors but exactly half of the tasks were assigned to lower leakage current processors. Upon numerical inspection the estimated sample latency was 3360 ns and the estimated leakage power was 310 mW. This mapping consumes about 34% less power than the unannotated mapping and is even about 6% faster than the unannotated mapping,

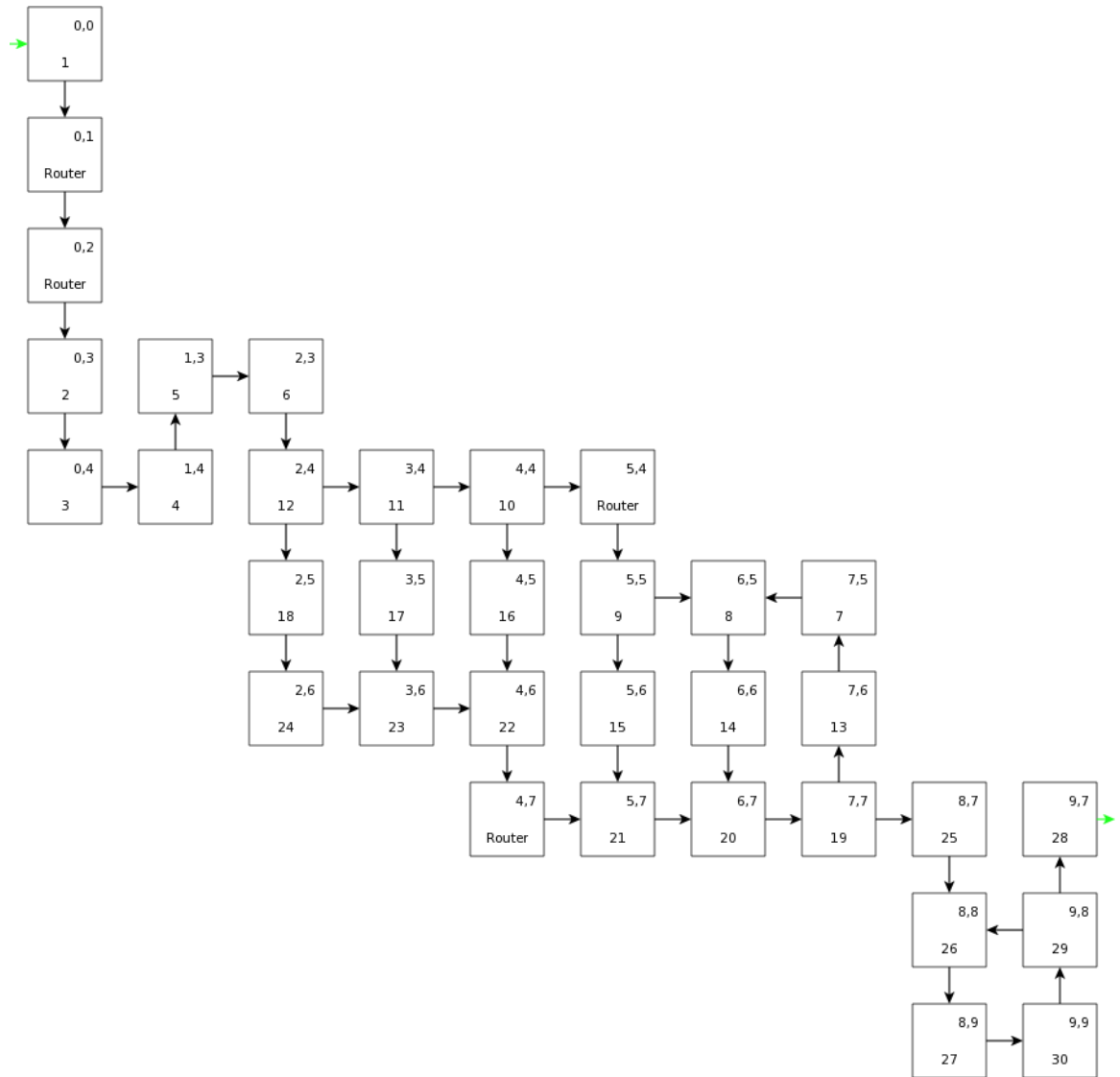


Figure 6.65: Automatic mapping using just speed related annotations for the Viterbi decoder

which clearly shows that both performance and power consumption have been improved.

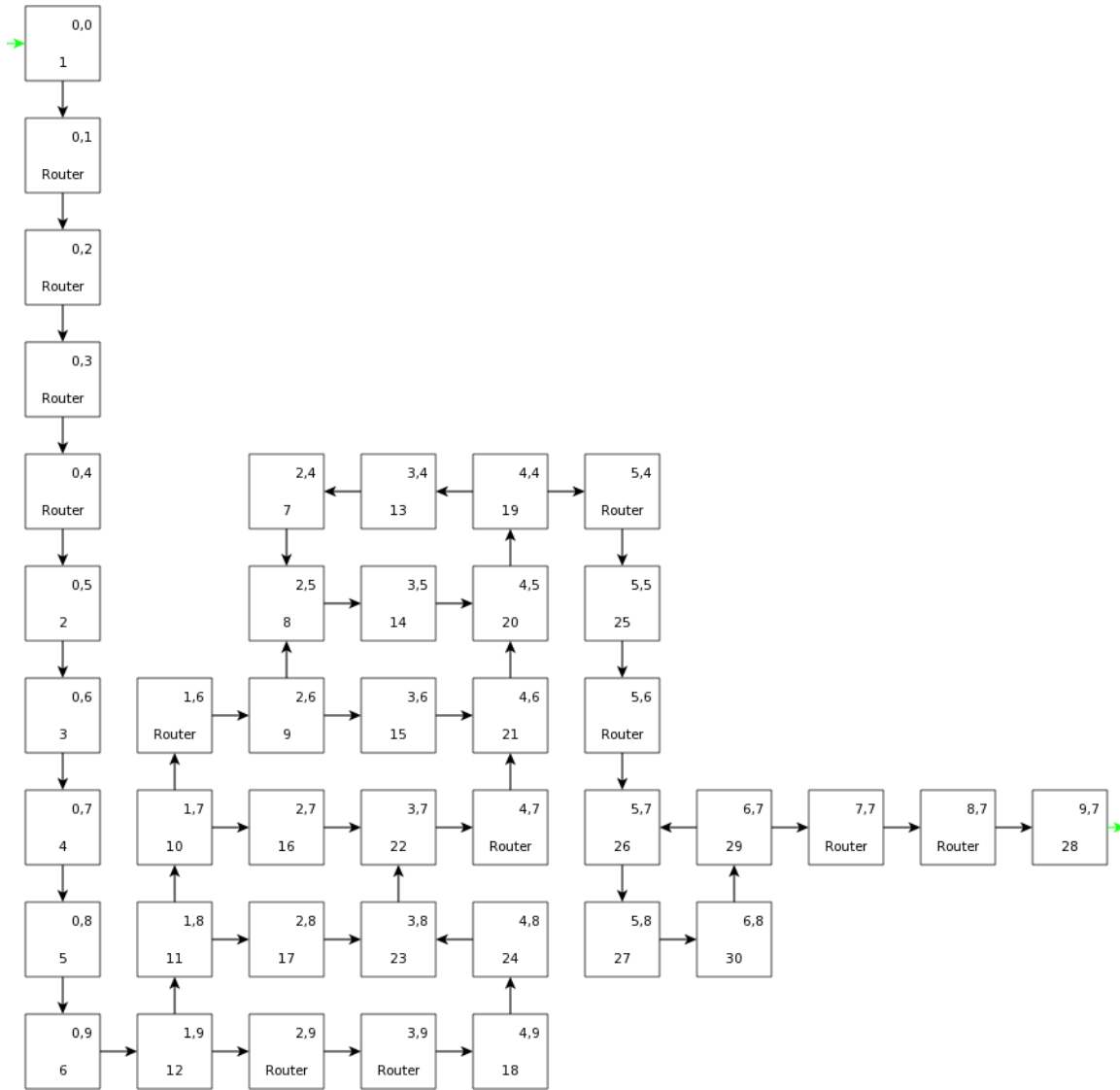


Figure 6.66: Automatic mapping using just power related annotations for the Viterbi decoder

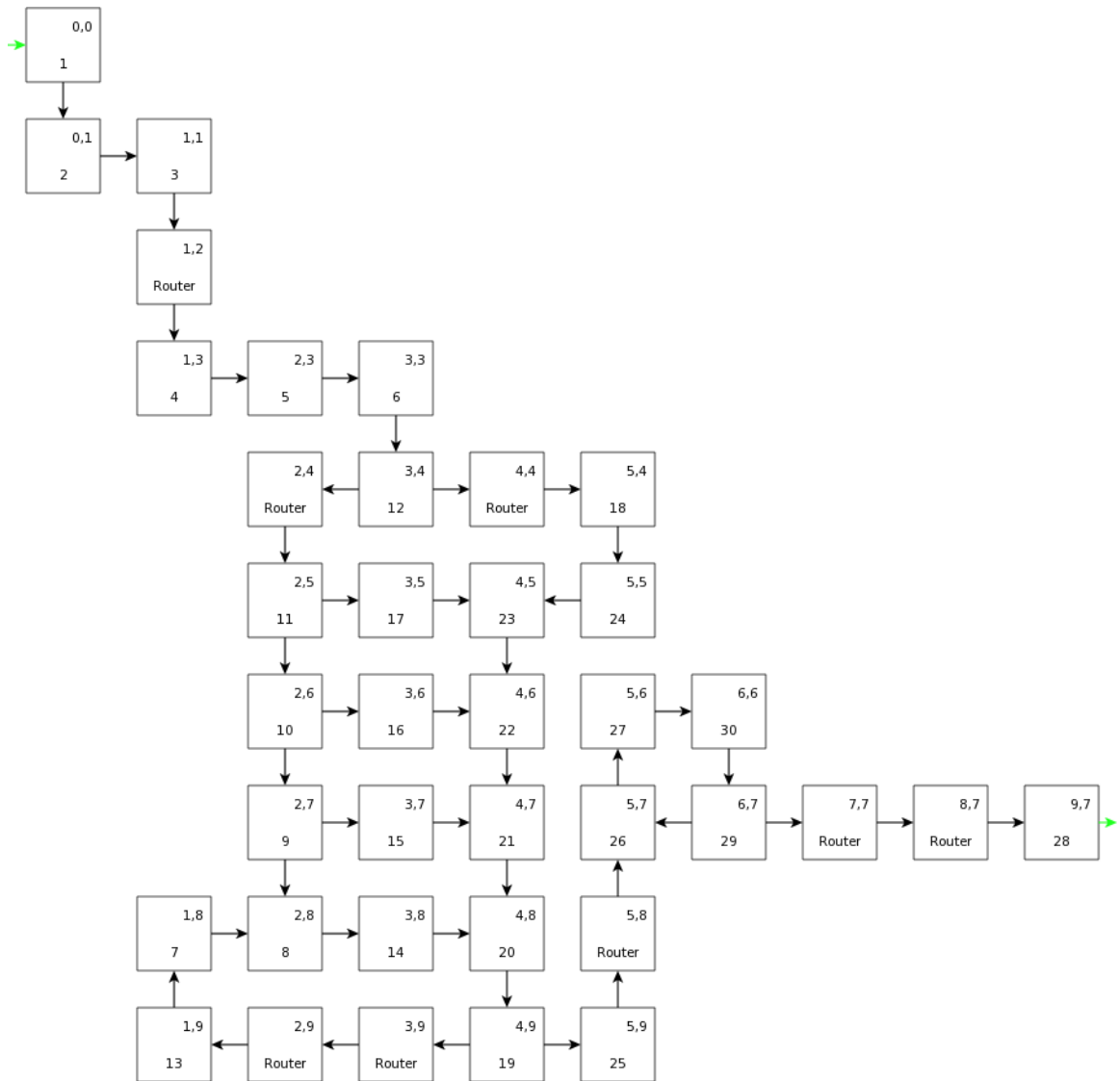


Figure 6.67: Automatic mapping using both speed and power related annotations for the Viterbi decoder

Test (Wireless)	Sample Latency		Leakage Power	
Unannotated	3000 ns	Ref	440 mW	Ref
Speed Annotations	2610 ns	13%	490 mW	-11%
Power Annotations	2930 ns	2%	280 mW	36%
All Annotations	2720 ns	9%	330 mW	25%

Table 6.7: The estimated sample latency (in nanoseconds) and the estimated leakage power (in milliwatts) for automatic mappings that use no annotations, use speed related annotations, use power related annotations, and use both speed and power related annotations, for the 802.11a wireless transmitter application

Test (Viterbi)	Sample Latency		Leakage Power	
Unannotated	3560 ns	Ref	470 mW	Ref
Speed Annotations	3300 ns	7%	460 mW	2%
Power Annotations	3500 ns	2%	280 mW	40%
All Annotations	3360 ns	6%	310 mW	34%

Table 6.8: The estimated sample latency (in nanoseconds) and the estimated leakage power (in milliwatts) for automatic mappings that use no annotations, use speed related annotations, use power related annotations, and use both speed and power related annotations, for the Viterbi decoder application

6.5.6 Summary

After mapping these two applications both with and without annotations, there are clearly advantages to including specialized annotations. By adding annotations that account for fabrication differences, tasks with higher demands are placed onto processors with better characteristics, thus improving performance and lowering power consumption. Figure 6.68 shows an outline of the three annotated mappings for the 802.11a wireless transmitter application overlaid on top of the annotated target array. Figure 6.69 similarly shows the three annotated mappings for the Viterbi decoder application overlaid on top of the annotated target array. The sample latency and leakage power results for the 802.11a wireless transmitter application have been summarized in Table 6.7. Similarly, the sample latency and leakage power results for the Viterbi decoder application have been summarized in Table 6.8. If the goal is to produce a mapping with the smallest possible rectangular array area, the least number of routing processors, or the least number of long-distance interconnects then annotations should not be used. When processor characteristics for the target array vary substantially and a sufficient number of extra processors are available, then annotations can be used to improve performance and power consumption. Adding annotations is similar to enabling any other optimization. How and when annotations are used depends upon the application and the target platform.

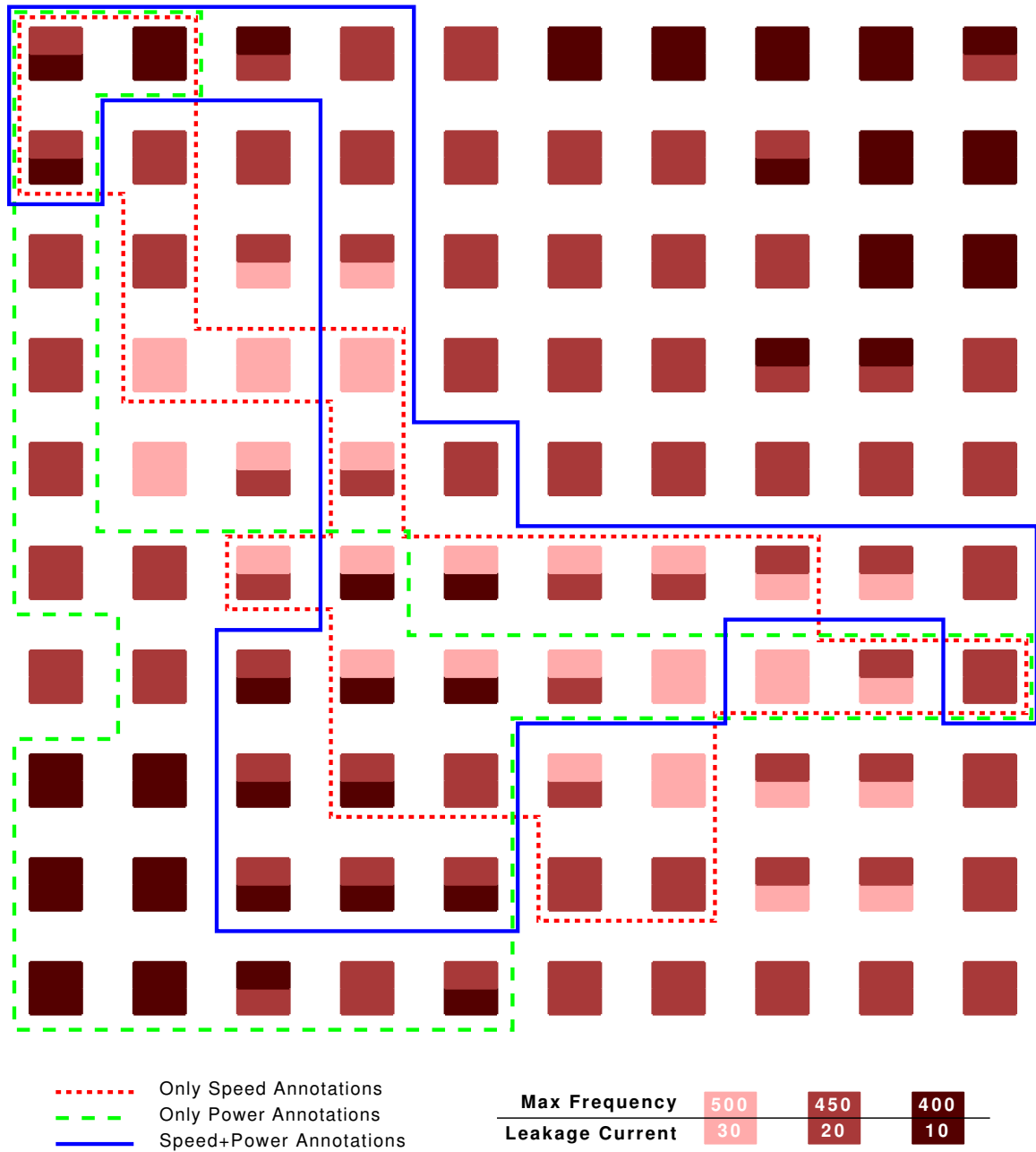


Figure 6.68: Visual comparison between the three annotated mappings (using only speed annotations, using only power annotations, and using both speed and power annotations) for the 802.11a wireless transmitter application. Also included is the maximum frequency and leakage current for each processor in the target array.

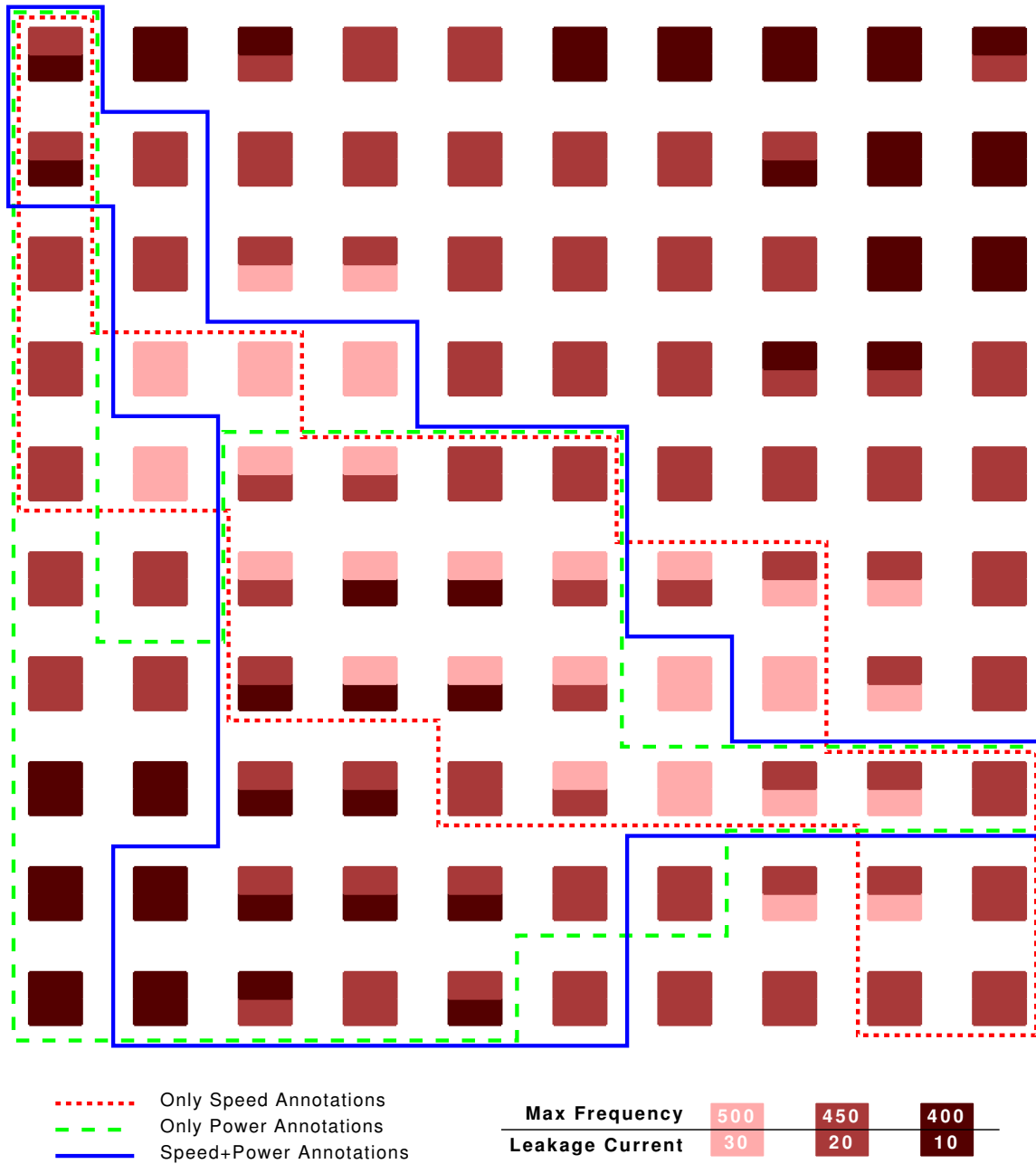


Figure 6.69: Visual comparison between the three annotated mappings (using only speed annotations, using only power annotations, and using both speed and power annotations) for the Viterbi decoder application. Also included is the maximum frequency and leakage current for each processor in the target array.

6.6 Conclusion

In summary the mapping algorithm is efficient, scalable, tolerant of processor failures, and able to optimize for fabrication differences. The tests performed throughout this chapter demonstrate these various qualities for the applications that were selected. These applications attempt to represent common dataflow patterns found in many DSP algorithms to provide well-rounded testing that is not application specific. In general the mapping algorithm is very flexible and when combined with the graphical user interface provides a invaluable tool that saves time when optimizing applications for AsAP and other parallel array architectures.

Chapter 7

Related Work

Systems with a large number of processing elements are not a new concept to the computing community. There are a number of subfields in computer engineering that deal with a large numbers of elemental units. Sometimes these elemental units are complete computing systems such as in super-computing clusters. Other times these elemental units are binary logic gates such as in VLSI design. These systems have applications in digital signal processing, physics simulations, and sensor networks, among others. Researchers in these fields have developed tools and algorithms for handling such a large number of units. This chapter will discuss previous work that is closely related to the AsAP architecture and the mapping problem addressed in this work.

7.1 Parallel Processor Arrays

Recently there has been a trend towards developing chip-multiprocessors. This includes multi-core and multiprocessor chips. This increase in interest is not because chip-multiprocessors are a recent innovation but because improvements in VLSI technology have made them realizable. This section discusses other recently developed parallel array processors that are similar to AsAP. I will point out any key differences from AsAP and how these differences affect the mapping problem.

7.1.1 RAW

The RAW architecture, designed by a research group at MIT, is a tile-based parallel array architecture [4, 33]. The first RAW chip contains 16 tiles arranged in a 4x4 grid. Each RAW tile contains a processor core and a dedicated network router. The processor core, which is similar to a standard MIPS processor, contains an 8-stage pipeline, large caches, and performs floating-point

arithmetic operations. Communication between tiles is performed using 4 full-duplex channels, one to each nearest neighbor, which have only a 1-cycle delay. Long-distance communication is less efficient and is performed by hopping through intermediate processors. Each processor contains two static routers, configured at compile time, and two dynamic routers, configured at runtime. The RAW team has developed a C compiler based on the GNU tool chain. The compiler not only produces instructions for the compute processors but also produces instructions for the network routers. A commercial product, named Tiler, was recently introduced that is based heavily on the RAW architecture [11].

The primary differences between AsAP and RAW are the size of the processor cores and the communication infrastructure. For AsAP the goal is to have small and simple processors so a large number of cores can be placed on a single chip. For RAW the goal was to create a general purpose architecture, which requires a flexible communication infrastructure. Just like in AsAP, the FIFOs in RAW perform flow-control and stall when writing to a full FIFO or when reading from an empty FIFO. Communication in AsAP may be nearest neighbor only, but processors are so small they can be used purely for routing without wasting much die area.

7.1.2 Smart Memories

The Smart Memories architecture, developed by a research group at Stanford, is a tile-based architecture designed for 0.1 μm technology generations and below [24]. The architecture is highly configurable which allows it to mimic other parallel architectures. Each tile contains local memory, local interconnects, and a processor core. To increase communication efficiency each block of 4 tiles is grouped into a quad. Tiles in a quad share a local data bus. Each quad connects to a global data bus. The local memory inside a tile is broken down into 16 independent 8 KB memory mats. The local interconnects, also called the crossbar, can dynamically route data between the processor core and the quad interface. The crossbar can process up to 8 requests at one time. Each processor core contains a 64-bit processing engine and uses a 256-bit microcode instruction format. Instructions can be packed into a 128-bit VLIW format, for applications that contain high instruction-level parallelism, or a 32-bit RISC format, for general purpose applications.

Very little was mentioned about how the Smart Memories architecture is programmed. The architecture appears to be based on Tensilica technology and uses the Tensilica XCC compiler. A related software project, developed by the same research group, called the Stream Virtual Machine (SVM), is an API for writing stream-based applications [20]. It uses a C-like language with SVM

API extensions to produce C code that can be compiled to different parallel architectures, Smart Memories being one of them. Further details on the high-level and low-level language components of the Stream Virtual Machine were not given.

Smart Memories is similar to AsAP in that local communication is highly efficient. Although a key difference between Smart Memories and AsAP is that AsAP doesn't have a global communication bus. Smart Memories also uses dynamic routing so dedicated routing processors are not needed. AsAP may be less flexible, but the trade-off is energy efficiency due to simpler processors and less memory. The applications that were tested on Smart Memories were mapped manually to the best of my knowledge. The high-level compiler for the Stream Virtual Machine is responsible for the mapping phase but specific details were not given. Overall the goal is the same, designing an architecture for future VLSI technologies.

7.1.3 iWarp

The iWarp architecture is a computing platform for high performance image and signal processing [8]. Up to 1024 iWarp components can be combined to form a single system. An initial prototype had 64 iWarp components connected in an 8x8 torus. Each iWarp component is an ASIC consisting of a computation agent and a communication agent. The computation agent consists of a processor core capable of 20 megaflops with a throughput of 320 MB/s. The communication agent has 4 inputs and 4 outputs that can access other iWarp cells or the outside world. Data can be routed in a number of ways including message passing, where data is occasionally exchanged, or systolic communication, where data flows in a stream.

One of the tools used for programming iWarp is the ASSIGN parallel program generator [27]. ASSIGN works in two phases by first partitioning the application into cells then placing and routing these cells. The output of ASSIGN is a set of C files, which can also be compiled on a typical PC. When partitioning an application, tasks are combined using load balancing until the number of partitions equals the number of cells. No details were given as to how partitions are assigned to cells. Since the iWarp system can be configured in almost any way possible, tasks could simply be assigned to cells sequentially.

The communication infrastructure for iWarp is very different than AsAP where communication can only be point-to-point and nearest neighbor. iWarp can be configured in almost any way desired and can even represent AsAP. This extra flexibility requires more complexity in each node. The reduced complexity of AsAP combined with using asynchronous communication make

AsAP more efficient for applications that map well to the architecture. The benefit of this additional complexity is that iWarp supports a larger number of applications.

7.1.4 Imagine Stream Processor

The Imagine platform, developed by researchers at Stanford, is comprised of 48 parallel ALUs running at 400 MHz [18]. It has a peak performance of 16 gigaflops for single-precision floating-point data and 32 gigaops for 16-bit fixed point data. For general purpose applications it would be difficult to continuously supply data to 48 ALUs simultaneously so Imagine classifies itself as a media processor. Imagine focuses on applications that follow a stream-based programming model and utilize producer-consumer locality. Imagine requires a host controller to program the on-board stream controller, which orchestrates dataflow to and from the ALUs. The Imagine Stream Processor also includes an off-board memory controller, streaming register file, and an instruction dispatch micro-controller. Data is loaded from an external SDRAM into the stream register file then into the local register file of each ALU. When the computation is complete data can be transferred back into SDRAM through the streaming register file. Alternatively, the ALUs can communicate between clusters via an 8x8 inter-cluster crossbar and also within clusters via an intra-cluster crossbar that connects all functional unit outputs to all register file inputs. VLIW instructions are broadcast to the ALUs from the instruction dispatch micro-controller, which also keeps track of the program counter. A commercial entity was recently formed called Stream Processors Inc., which is based around the Imagine Stream Processor.

The StreamC and KernelC languages are used for programming the Imagine Stream Processor [17, 25]. StreamC is used for scheduling data transfers to and from the ALUs while KernelC is used to describe the operations performed by the ALUs. StreamC programs are written in C++ and linked to a special library that performs stream operations and host PC communication. A primary goal when orchestrating dataflow is taking advantage of intra-cluster bandwidth, which is 10x faster than passing data through the streaming register file. KernelC programs are written in a subset of C and operate on single data elements.

Imagine and AsAP both work very well for applications that exploit cascading task-level parallelism. Cascading task-level parallelism can be found in applications that use sequential operation and process continuous datastreams. For Imagine, one such application is graphics processing. For AsAP, a few of these applications include an FFT and multi-pass filters. A major distinction between Imagine and AsAP is the communication infrastructure. In AsAP communication must be

between processors while in Imagine communication can either be performed globally, through the stream register file, or locally between ALUs. AsAP can perform more complex operations, while keeping the data local, by having full RISC-based CPUs in each processing unit.

7.1.5 Explicit Data Graph Execution (TRIPS)

The Explicit Data Graph Execution (EDGE) Instruction Set Architecture (ISA) is an instruction set architecture for a new generation [9]. The TRIPS prototype is the first implementation of an EDGE ISA, developed by a team at the University of Texas, Austin. In typical ISAs data is read from the register file and written back to the register file. In an EDGE ISA data transfers between instructions are performed explicitly. The TRIPS prototype has two cores, each with 16 ALUs arranged in a 4x4 grid, running at 366 MHz. Each ALU is connected by a lightweight network to a bank of 4 register files. Instructions are dispatched in parallel and stored in small local caches inside each ALU. The chip consumes 36 W when operating at a clock frequency of 366 MHz and has an on-chip bandwidth of 4.7 GB/s [30].

The TRIPS team has developed a specialized compiler for programming the array of ALUs [32]. The TRIPS compiler is designed to balance the trade-off between work done in hardware and work done in software. The goal of the compiler is to find concurrency in the application to assure that data is always flowing through the ALUs. The compiler works by creating blocks of 128 instructions then allocating registers and ALUs to each block. Each block can only contain 32 loads and 32 stores since each operation is given a Load-Store ID (LSID). LSID values are used to ensure that memory is accessed in the correct order across ALUs.

The EDGE architecture and the AsAP architecture are somewhat dissimilar. The goal for both architectures is to maximize all three forms of parallelism, instruction-level parallelism, data-level parallelism, and thread-level parallelism. Both architectures have multiple execution units but in EDGE these units are only the ALU component and the memory is shared between all units. In AsAP each unit has its own pipeline and its own memory. AsAP uses static task allocation so tasks are available the entire life of the program. EDGE uses dynamic task allocation but decides these allocations at compile time.

7.2 Parallel Programming Tools

Various methods for programming parallel architectures have been studied over the past couple decades, but only recently have these methods been applied to real applications and physical

devices. New languages and programming tools are being developed to make it easier for novice programmers to utilize this new type of hardware. Below I will discuss some of these new tools and how they are related to this work as well as how they differ.

7.2.1 StreamIt

StreamIt, developed by researchers at MIT, is a portable framework for programming stream-based architectures [14, 15]. The StreamIt language and compiler are designed to overcome many of the shortcomings associated with using C to program stream-based architectures. The StreamIt language has a syntax very similar to C and Java. The StreamIt compiler performs all the tasks necessary to generate code for the target architecture. Although the primary target for StreamIt is the RAW architecture, it can be adapted to other communication exposed architectures.

The StreamIt language has 4 basic constructs: the filter, for serial computation; the split and join, for parallel computation; and the feedback loop, for creating cycles. When compiling code StreamIt language files are first converted into StreamIt IR (Intermediate Representation). StreamIt IR contains dataflow and hierarchy information for the application. This is followed by partitioning, layout, and communication scheduling which are more architecture dependent. For the partitioning phase the compiler uses a series of fission and fusion operations that combine lightly loaded nodes and split heavily loaded nodes until the number of nodes matches the number of tiles. For the layout phase the compiler uses simulated annealing. The configuration cost is derived from the number of items being transferred and the number of hops along the path. For the communication scheduling phase the compiler must assure that no deadlocks will occur. Deadlocks are avoided by interleaving read and write operations with the computation code.

StreamIt and this work are closely related and share many common interests. The StreamIt compiler deals with every aspect of parallel programming while this work focuses on layout and communication. Comparing layout algorithms both use simulated annealing but this work requires additional optimizations since AsAP is more constrained than RAW. This work also allows the programmer to optimize for a specific chip by configuring parameters for fault tolerance and physical differences. Comparing communication algorithms this work has to explicitly insert additional nodes for routing since AsAP requires nearest neighbor communication. RAW has a dedicated routing network that simplifies communication scheduling. This work uses special routing code similar to StreamIt that detects deadlocks before they occur and continues with other operations first if possible. This work was somewhat inspired by StreamIt, which is the reason for many of the

similarities.

7.2.2 OREGAMI

The OREGAMI project, developed by researchers at the University of Oregon, is a collection of tools for programming parallel architectures [23]. The project contains 3 components: LaRCS, a graph description language; MAPPER, a library of mapping algorithms; and METRICS, an interactive graphics tool. OREGAMI is intended to be a front-end for existing parallel languages and supports architectures with a regular communication network, such as a hypercube or mesh. A primary interest of the OREGAMI project is to exploit regularity in both structure and communication. OREGAMI is based on a new graph theoretical method called Temporal Communication Graphs (TCG). TCGs are similar to static dataflow graphs except they provide information about communication patterns over time.

Applications in OREGAMI are described using LaRCS. A LaRCS file contains two parts, the static structure of the application, and the temporal communication behavior of the application. The LaRCS representation of an application and a description of the target architecture are then passed to MAPPER, which performs the mapping. If MAPPER recognizes regularities in both the application and the target architecture then MAPPER uses a canned solution from its library. If no canned solution was found then MAPPER uses an arbitrary contraction, embedding, and routing algorithm. For arbitrary graphs MAPPER first executes the contraction phase, which performs clustering through load-balancing. This is followed by the embedding phase, which places the nodes with the most communication adjacent to each other. Finally the routing phase is executed, which minimizes contention and produces source, destination, and intermediate route points. The results from MAPPER are analyzed using METRICS, which also provides an estimate of the mapping quality.

The OREGAMI project and this work have some similar components. Comparing components, LaRCS serves a similar purpose as XML module files, MAPPER serves a similar purpose as libamap, and METRICS serves a similar purpose as asapmap. The primary difference between OREGAMI and this work is that OREGAMI requires regularity in the application for it to really work, while this work assumes the application is entirely arbitrary. Even though both tools target homogeneous architectures this work optimizes for nearest neighbor parallel array architectures while OREGAMI requires a less restrictive architecture, which allows it to use canned solutions. This work uses simulated annealing so the algorithm is more flexible and can deal with fault toler-

ance and the physical properties of the target. OREGAMI uses a more theoretical approach that produces results closer to optimal for applications and architectures with known regularities.

7.2.3 CASCH

The CASCH (Computer Aided SCHEDuling) project, developed by researchers at the University of Hong Kong and the State University of New York at Buffalo, is a complete parallel programming environment [5]. Their tools perform all the steps necessary to convert a sequential program into a parallel program. This includes code generation for the Intel Paragon. The major components are: a code lexer and parser; a node and edge weight estimator; a dataflow graph generator; a scheduling and mapping tool; and a communication insertion and code generation tool. The project also includes some graphical analysis tools. The CASCH project is mostly an integration of related research into a unified framework.

Mapping an application with CASCH primarily involves scheduling. Only a minimal amount of effort is applied to placing the application onto physical processors. The user can select which scheduling algorithm to use from a collection of scheduling algorithms that are each designed for a specific architectural environment. There are three categories of scheduling algorithms. The first category, called UNC (Unbounded Number of Clusters), assumes that the network is fully connected and there are an unlimited number of processing elements. The second category, called BNC (Bounded Number of Clusters), also assumes that the network is fully connected but there are a limited the number of processing elements. The third category, called APN (Arbitrary Processor Network), assumes that there is an arbitrary network and a limited number of processing elements. Once scheduling is finished tasks appear to be placed onto processing elements sequentially, starting with the highest priority task first. Different scheduling algorithms can be tested from their graphical user interface in order to find the best schedule.

The primary difference between CASCH and this work is that CASCH focuses on partitioning and scheduling while this work focuses on mapping and communication. The reason CASCH doesn't exert much effort when placing an application onto processors is that the scheduling algorithms assume that processors can talk to each other either directly or through a routing network. For this work all communication is assumed to be nearest neighbor only and scheduling is handled by the asynchronous FIFOs. Despite these differences both tools are very useful for programming parallel architectures.

7.2.4 PYRROS

The PYRROS project, developed by researchers at Rutgers University, contains an automatic scheduler and a code generation tool [36]. These tools focus on MIMD message passing architectures such as the nCUBE-1, the nCUBE-2, and the Intel iPSC/2. The project contains the following components: a task graph language; a scheduling system; a code generator; and some graphical tools. The application must already be partitioned into parallel tasks prior to using these tools.

The PYRROS tools take as input a task graph described using a simple C-like language. Scheduling is performed in four phases. First tasks are clustered assuming an unbounded number of processors. Next clusters are merged using load balancing until the number of clusters is equal to the number of processors. Next clusters are placed onto physical processors by first performing an initial placement followed by incremental improvements to reduce communication delay. Finally the tasks within merged clusters are reordered to reduce data dependency delays. After scheduling, the code generation tool produces both communication code and computation code. The communication code includes code for deadlock avoidance and message broadcasting.

The PYRROS project, much like the CASCH project, is focused more on scheduling applications then mapping them onto physical processors. PYRROS and this work may both target parallel architectures, but the communication infrastructures are vastly different. The first version of AsAP is nearest neighbor only while the nCube for example is connected in a hypercube. For this work communication is scheduled using software based routing tasks that are separate from computation tasks. For PYRROS communication is scheduled during code generation and integrated into computation tasks. This work is somewhat unique since it takes into account a number of other factors besides just communication delay, such as fabrication errors and differences in processor characteristics. One advantage the PYRROS project has over this work is that it can manipulate the code in ways not possible with this work.

7.2.5 HyperTool

The HyperTool project, developed by researchers at the University of California, Irvine, consists of a tool for scheduling parallel programs and inserting communication primitives [35]. Hypertool has been designed for message passing architectures such as the Intel iPSC and the nCUBE. Hypertool is very similar to other parallel programming tools and includes similar components. These components are: a C-code lexer and parser; a dataflow graph generator; an execution sched-

uler; a physical processor mapping tool; a communication synchronization tool; and a code generator. This tool does not partition sequential code into parallel code since the authors believe that code partitioning is best handled by the user. This tool takes as input a C program with one main procedure and many parallel sub-functions. The call graph is converted into a DFG used for scheduling, mapping, and code generation.

The scheduling component begins by estimating the number of virtual processors required to fully maximize parallelism. Tasks are then scheduled onto these virtual processors using both as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) scheduling techniques. Tasks that have the same value for ASAP scheduling as ALAP scheduling are given the highest priority. The mapping component then tries to minimize the communication distance between tasks. An optimal mapping is one that uses only nearest neighbor communication. Hypertool uses a previously developed mapping algorithm that starts with an initial placement then iteratively swaps tasks to try and improve the mapping. The synchronization component then inserts send and receive commands into processing elements that pass data for non-nearest neighbor connections.

Hypertool and this work have a similar goal, which is to decrease the length of communication channels. Both algorithms start with an initial configuration then iteratively improve the configuration to reduce communication delay. One primary difference between Hypertool and this work is that Hypertool integrates communication code with computation code where this work relies on dedicated routing processors. Integrating communication code with computation code helps to decrease the number of processing elements. For AsAP the instruction space is very limited so using extra processors for routing is preferred since they are considered throw-away processors anyway. This work also performs other optimizations besides just minimizing communication delay.

7.2.6 Energy-Aware Mapping Algorithm

Researchers at Carnegie Mellon University have developed a mapping algorithm for tile-based NoC architectures [16]. The mapping algorithm is energy-aware, which means communication energy is minimized. The communication energy includes both the number of hops and the energy consumed per transmission. Architectures can be heterogeneous with a mix of ASICs, DSPs, and CPUs, but each tile must have an interface compatible static router. It's assumed that each tile is connected to its four nearest neighbors and uses XY routing. This branch-and-bound based mapping algorithm efficiently explores the solution space by intelligently deciding which configurations to explore. IP cores are first sorted and placed into a priority queue. Next a solution space tree is

constructed based on the many ways IP cores can be placed onto processors. Upper and lower bounds are computed for each level in the solution space tree. Part of the solution space tree can be trimmed using the values for the upper and lower bounds, narrowing the solution space.

The authors compared their algorithm to a simulated annealing based algorithm and noticed significant speed improvements while obtaining similar mapping quality. This is likely true for the simple cost equation they were using. Though for this work the cost equation is quite complex and accounts for many other aspects of the mapping. For example this energy-aware mapping algorithm relies on dedicated routers so it doesn't have to worry about inserting routing processors or communication code. Since this energy-aware algorithm only minimizes communication energy it's likely to be faster. This work is more feature rich but the trade-off is increased runtime.

Chapter 8

Conclusion

The automated mapping algorithm presented in this work has been shown to be an efficient tool for programming large scale parallel arrays. By applying the mapping algorithm to the AsAP architecture, programmers are able to design applications in less time and often with better performance. One of the goals for AsAP, and other parallel arrays, is to adapt to future VLSI technologies. This includes an increasing number of processing elements and an increasing number of fabrication errors as feature sizes approach an atomic level. With the help of the mapping tool, applications have been mapped to AsAP that were so large they would have been nearly impossible to map by hand. Not only does the mapping tool overcome fabrication problems but it takes advantage of these problems to increase performance. The benefits of a rapid and fully automated mapping tool that takes into account faulty and varying-performance processors, makes a strong case for the usage of automated mapping tools for programming large scale parallel arrays.

A framework has been developed for mapping applications to 2D-mesh nearest neighbor dominated parallel arrays. The framework is modular and can be adapted to other large scale parallel arrays with little work. An automated mapping algorithm and an intuitive graphical user interface have been created that initially target the AsAP architecture. The mapping algorithm is highly configurable and capable of mapping applications that require over a dozen optimization factors at one time. The mapping algorithm has been shown to be efficient, scalable, tolerant of processor failures, and able to optimize for fabrication differences. The graphical user interface explores a new technique for programming large scale parallel arrays by allowing applications to be created based on their dataflow. The graphical user interface allows programmers to quickly develop new applications and experiment with different configuration parameters in an entirely visual way. The

most important attribute of this framework is the ability to improve and adapt the framework to new architectures.

8.1 Lessons Learned

While implementing and testing the mapping algorithm I ran into a number of problems. The biggest mistake I made was programming the graphical user interface before completing the mapping algorithm. This delayed my initial results by a few months and the graphical user interface later needed to be frequently revised, especially in the mapping algorithm setting dialog. Although the upside to this approach was applications were easy to create and my initial results were simple to analyze. In the first implementation of the mapping algorithm the placement and routing phases were combined. Routing processors were periodically inserted along the longest edge. The mapping quality for this first implementation was terrible and the mapping algorithm was unnecessarily complex. Results improved after splitting the mapping algorithm into two phases but complex applications, such as the large Clos network, were still unmappable when targeting the first version of AsAP. To overcome this problem the space insertion component was added to the routing phase. This was mostly a hack. This component degraded the mapping quality for some applications but allowed complex applications, such as the large Clos network, to be mapped using nearest neighbor communication only. The edge-to-vertex ratio, or space threshold, determines when this hack is needed. This threshold unreliably estimates the complexity of an application so sometimes good mappings get trashed by the space insertion component. Given more time I would have liked to explore other methods for determining when space insertion is needed. From each of these problems valuable lessons were learned that should be avoided by anyone improving this framework or doing similar work.

If tomorrow I had to start this project all over again, I would have done a number of things differently. Currently the graphical user interface is limited to point-to-point connections. The graphical user interface should allow an user to input one-to-many and many-to-one connections and handle fan-in/fan-out appropriately. Implementing this change would be far from trivial and it would likely be easier to do a complete rewrite. After reading about the energy-aware mapping algorithm it brought back notions of doing an intelligent full-search. To do this, a solution space tree is created starting with the input vertex then branches are removed that clearly lead to an inferior solution. This would result in an optimal solution. Of course the runtime would also increase, but hopefully not by too much. Another modification I would have liked to try is replacing the

initial placement function by a graph planarization process. This would have improved the mapping quality and lowered the initial temperature, thus saving time. If this work was started at a later date, the second version of AsAP would have been fabricated. I believe it's necessary to follow up this work with some simple performance and power measurements after taking into account fabrication differences. With multi-core processors on the rise this problem is unlikely to be sufficiently solved in the near future.

8.2 Future Work

It has been demonstrated that the presented mapping algorithm efficiently produces quality mappings for most of the applications tested. However, as with any tool dealing with large NP-complete problems, there is certainly room for improvement. In addition, there are still a number of valuable tools missing from the parallel programming tool-chain that must be implemented. As a starting point, the following list contains suggestions for future work:

- Develop a serial to parallel code partitioning algorithm that compliments the AsAP mapping tool
- Develop an instruction scheduling algorithm for optimizing global throughput that compliments the AsAP mapping tool
- Modify the AsAP mapping tool and the XML module format to work directly with C code
- Explore using various graph planarization techniques to improve the initial placement function
- Modify the mapping algorithm so the array input and the array output are allowed to float along any combination of the four edges
- Explore new ways to insert additional routing space to reduce array area and routing conflicts
- Explore ways to reorder the list of edges given to the maze routing algorithm to reduce routing conflicts
- Run some complex applications on the physical chip and measure the difference in power and throughput as a result of the user cost function
- Put the mapping algorithm into a co-processor that will automatically detect processor failures and remap applications on-the-fly

- Integrate canned mapping solutions into the mapping algorithm to optimize common dataflow patterns found in many DSP applications
- Explore using min-cut in conjunction with hierarchical data structures to improve the mapping quality

Appendix A

Readme - Mapping Library

Contained within this appendix is the contents of the `Readme` file associated with the mapping library. This file can be found inside the ASAP mapping tool source code archive. This `Readme` file was written to provide the end user with instructions on compiling the mapping library and the programming API necessary to embed the mapping library into an application. A general overview of the placement and routing phases is also provided along with a brief description of the contents within each source code file.

=== Documentation for the ASAP Mapping Library ===

Author: Eric Work
E-mail: ewwork@ucdavis.edu
Modified: February 2007

***** Overview *****

The primary objective of the mapping library is to assign interconnected tasks to locations in a 2D-mesh array of processors. These assignments are optimized to maximize nearest neighbor communication, minimize overall array area, as well as allow for some customization. A few of the customizations possible are fixing the location of certain tasks and excluding certain processors from being assigned.

The mapping library works in two phases. The first phase is the placement phase, which is based on simulated annealing. The second phase is the routing phase, which is based on maze routing. Each phase has been optimized to increase placement quality for an ASAP nearest-neighbor architecture. Since every application is different the algorithm has a number of configuration parameters that can be changed to improve placement quality for specific applications. For further details refer to my thesis on the mapping algorithm.

The following is a break-down of the source file contents:

```

algorithm.cpp - algorithm entry point and configuration validation
algorithm.hpp - configuration data-types and their default values
graph.hpp     - graph data-types with related descriptors and iterators
gridmap.cpp   - routing gridmap implementation details
gridmap.hpp   - routing gridmap data-types with helpers
perturb.cpp   - placement perturb implementation details
perturb.hpp   - placement perturb data-types with helpers
placement.cpp - placement entry point and simulated annealing framework
routing.cpp   - routing entry point and maze routing framework
testbench.cpp - testbench implementation for debugging the library
usercost.cpp  - user cost function and data file parsing
utility.cpp   - utility functions for both placement and routing
utility.hpp   - utility function declarations used everywhere

```

***** Compiling the Library *****

To compile the mapping library you will need to have the boost libraries installed. The boost libraries are a collection of generic programming libraries designed using C++ templates. The boost libraries can be obtained from <http://www.boost.org>. In particular you will need the following libraries: graph, property_map, format, random, date_time, and string_algo. The filesystem library is also needed when building the library in debug mode. The resulting binaries are statically linked and require no other runtime libraries beyond the basic libstdc++.

If you want to compile the optional testbench application you will need to build an additional library for reading graphviz files. The source code for this library is included with the boost graph library source code. This can be found inside the following directory of the boost source code, './libs/graph/src/'. The i686 version has been included with this package. This pre-compiled graphviz library was compiled with '-march=i686 -O2' using GCC 4.1.X. Use the following steps to build the static library for a different architecture or platform:

1. g++ -ftemplate-depth-50 -O2 -I../.. -c graphviz_*.cpp
2. ar -rc libbgl-'arch'.a *.o
3. mv libbgl-*a /path/to/mapping/library

The location where the boost libraries are installed can be specified by changing the BOOST_XXX variables in 'Rules.make'. When building in release mode optimizations are enabled but further tuning can be done by setting the CXXFLAGS environment variable. For example, optimizing for the Pentium 4 can be done using the following commands before running make:

```

bash: export CXXFLAGS="-march=pentium4"
tcsh: setenv CXXFLAGS "-march=pentium4"

```

To compile the mapping library type: make all

To compile the testbench type: make DEBUG=y testbench

***** Using the Library *****

The mapping library will most commonly be used as a component in other applications. The algorithm testbench is one front-end for the mapping library which could be used as a simple command-line tool with minor modifications. The AsAP Mapping Tool on the other hand is a full-featured graphical interface for the mapping library. Either one of these front-ends can serve as an example for interfacing with the library's simple API.

To begin, all programs which link to the mapping library need to include 'algorithm.hpp' and create at least one object of type 'graph_t' and 'configuration_t'. An object of type 'graph_t' is used to describe the application data-flow and an object of type 'configuration_t' contains all the algorithm parameters. These objects are passed to 'algorithm_main' which then does the mapping. The 'input_vertex' and 'output_vertex' fields of the configuration object must be set before running the algorithm. If anything appears inconsistent within the graph or the configuration an error message will be displayed and 'algorithm_main' will return false. Below is a very simple example of using the mapping library.

```
#include <algorithm.hpp>

void prepare_graph (graph_t &graph, configuration_t &config) {

    vertex_t input = add_vertex(graph);
    vertex_t middle = add_vertex(graph);
    vertex_t output = add_vertex(graph);

    add_edge(input, middle, graph);
    add_edge(middle, output, graph);

    config.input_vert = input;
    config.output_vert = output;
}

int main () {

    graph_t graph;
    configuration_t config;

    prepare_graph(graph, config);

    config.num_iterations = 1;
    config.random_seed = 123;

    algorithm_main(graph, config);
}
```

The mapping library makes extensive use of the boost graph library, and the first step is to prepare the desired data-flow graph. The majority of this preparation is based on the API used for creating generic boost graph objects. There are a few additional properties that can be set to make conversions between different in-memory graph formats easier. These are the

vertex source and edge connection properties for storing the information about the original graph. Below are the functions used to create nodes and edges as well as work with node, edge, and graph properties. For more advanced graph manipulations refer to the boost graph library documentation.

=> New vertex:

```
vertex_t // new vertex descriptor
add_vertex(
    graph_t graph // graph object
);
```

=> New edge:

```
std::pair<edge_t, bool> // new edge descriptor, always true
add_edge(
    vertex_t source, // source vertex descriptor
    vertex_t target, // target vertex descriptor
    graph_t graph // graph object
);
```

=> Vertex source property:

```
vertex_t vertex; // previously defined vertex descriptor
object_t object; // previously defined generic object
source_map_t source_map = get(
    vertex_source, // property map tag
    graph // graph object
);
source_map[vertex] = (void *)&object; // set value
void *data = source_map[vertex]; // get value
```

=> Vertex category property:

Values:

```
CATEGORY_PROCESSOR - processor node type
CATEGORY_ONEWAY_ROUTER - one-way router type
CATEGORY_TWOWAY_ROUTER - two-way router type
CATEGORY_THREWAY_ROUTER - three-way router type
CATEGORY_FOURWAY_ROUTER - four-way router type
```

```
vertex_t vertex; // previously defined vertex descriptor
category_map_t category_map = get(
    vertex_category, // property map tag
    graph // graph object
);
category_map[vertex] = CATEGORY_PROCESSOR; // set value
if (category_map[vertex] == CATEGORY_PROCESSOR) { } // get value
```

=> Vertex coordinate property:

```
vertex_t vertex; // previously defined vertex descriptor
int x, y; // previously defined coordinate dimensions
coordinate_map_t coordinate_map = get(
```

```

        vertex_coordinate, // property map tag
        graph              // graph object
    );
    coordinate_map[vertex] = coordinate_t(x, y); // set value
    coordinate_t coordinate = coordinate_map[vertex]; // get value

```

=> Edge connection property:

```

    edge_t edge; // previously defined edge descriptor
    int source, target; // previously defined port numbers
    connection_map_t connection_map = get(
        edge_connection, // property map tag
        graph            // graph object
    );
    connection_map[edge] = connection_t(source, target); // set value
    connection_t connection = connection_map[edge]; // get value

```

=> Graph dimensions property:

```

    int x, y; // previously defined coordinate dimensions
    coordinate_t &dimensions = get_property(
        graph, // graph object
        graph_dimensions // property map tag
    );
    dimensions = coordinate_t(x, y); // set value
    coordinate_t coordinate = dimensions; // get value

```

The next preparation step involves configuring the algorithm parameters. Various fields of the 'configuration_t' object can be modified to change the behavior of the algorithm. For further details refer to the contents of the 'algorithm.hpp' file. Listed below are the configuration fields with a short description and a simple example for setting them. Default values can be found in 'algorithm.hpp'.

=> input_vert/output_vert: input and output vertex descriptors

```

    vertex_t vertex; // previously defined vertex descriptor
    config.input_vert = vertex;

```

=> input_type/output_type: input and output vertex placement positions

Values:

```

    POSITION_NONE - ignore placement related costs
    POSITION_LEFT - cost based on distance from left edge
    POSITION_RIGHT - cost based on distance from right edge
    POSITION_TOP - cost based on distance from top edge
    POSITION_BOTTOM - cost based on distance from bottom edge

```

```

    config.input_type = POSITION_LEFT;

```

=> array_size: desired array size

```

    int x, y; // previously defined coordinate dimensions
    config.array_size = coordinate_t(x, y);

```

=> quick_place: quicker placement phase flag

```
config.quick_place = true;
```

=> use_routing: perform routing phase flag

```
config.use_routing = false;
```

=> add_spacing: additional space before routing flag

```
config.add_spacing = true;
```

=> expand_type: initial placement expansion direction

Values:

DIMENSION_VERTICAL - place vertical first then horizontal

DIMENSION_HORIZONTAL - place horizontal first then vertical

```
config.expand_type = DIMENSION_HORIZONTAL;
```

=> num_iterations: number of placement iterations (Max 32)

```
config.num_iterations = 1;
```

=> max_routes: number of routable paths through a node (Max 4)

```
config.max_routes = 4;
```

=> random_seed: initial random seed

```
config.random_seed = 123;
```

=> space_threshold: edge/node ratio required to insert space

```
config.space_threshold = 100;
```

=> cost_weight[index]: array of cost weights used during placement

Indexes:

COST_EXCLUDED_MATCH - cost for using an excluded coordinate

COST_CHANNEL_LENGTH - cost for using non-nearest neighbor communication

COST_INOUT_DISTANCE - cost for the input/output edge distance

COST_ARRAY_OVERSIZE - cost for exceeding the desired array dimensions

```
config.cost_weight[COST_CHANNEL_LENGTH] = 10;
```

=> excluded_coords: vector of coordinates to exclude

```
int x, y; // previously defined coordinate dimensions
```

```
coordinate_t coordinate(x, y);
```

```
config.excluded_coords.push_back(coordinate);
```

=> fixed_coords: pairs of vertices with their fixed coordinates


```

vertex_t vertex; // previously defined vertex descriptor
int x, y;        // previously defined coordinate dimensions
coordinate_t coordinate(x, y);
vertex_coord_pair_t fixed_pair = std::make_pair(vertex, coordinate);
config.fixed_coords.push_back(fixed_pair);

```

When the preparation is complete, the algorithm can be executed. This can be done by calling 'algorithm_main'. Below is a more detailed description of how the function can be called.

```

bool // returns true unless some failure occurred
algorithm_main(
    graph_t graph,          // graph object
    configuration_t config // configuration object
);

```

When the algorithm is being executed, the status of the algorithm is printed to `std::cout` and any error messages are printed to `std::cerr`. The global `std::string` variable named 'mapping_error_str' will contain the last error message reported if 'algorithm_main' returns false. During the placement phase, it prints the iteration number as well as the temperature progress. During the routing phase it prints the route number being solved along with the total number of routes. A simple report is generated when both phases have been completed. This includes the array size, the number of non-nearest neighbor connections, and other useful information. This output information can be hidden or redirected by assigning `std::cout` and `std::cerr` to new values before executing 'algorithm_main'.

***** Algorithm Testbench *****

The testbench has been designed to easily test the performance and quality of the mapping algorithm as well as serve as an example for using the library. Though not required, the testbench is usually compiled in debug mode which provides internal algorithm information. The testbench is typically executed from the 'testing' directory using make. Simply running 'make' in this directory displays a list of available options.

Before running the testbench first create an 'input.dot' file. This input file is a graphviz DOT file but could also be a symlink to a file in the graphs directory. The input file must have nodes with the labels 'in' and 'out' for the testbench to work. Look at files in the graphs directory for examples. The next step is to run 'make execute' which will execute the testbench and generate the needed debug files. This is followed by 'make results' which processes the debug files and places the results in the results directory. To view the results run 'make analyze' which displays the processed results using an appropriate viewer for each file type. Since each system is different, the programs used by the analyze target may need to be changed to work correctly on your system.

Below is a quick example of how to execute the testbench:

```
1. ln -s graphs/fourier.dot input.dot
```

2. make execute results analyze

Debug files are created in a new directory called 'debug' when the testbench is executed in debug mode. This directory is created in the working directory where the testbench was executed. It contains a series of three file types for debugging both the placement and routing phases. The *.dat files are in CSV format and contain tabulated cost, temperature, and iteration data. The *.dot files are in graphviz DOT format and have position attributes set as well as input/output labels and type coloring. The y-coordinates for the resulting graphviz DOT files have been inverted to correctly render using neato. The *.map files are in plain text format and contain rows of ASCII characters depicting the processor array the types of nodes with respect to their routing states. By default only the initial and final states for placement and routing phases are processed. One additional file not previously mentioned is the 'output.dot' file created by the testbench regardless of debug or release mode. This file contains the final layout including routing processors.

The following is a break-down of the dumped debug files:

```

placement.dat          - placement temperature and cost data
placement-initial.dot  - graph layout before the placement phase
placement-final.dot    - graph layout after the placement phase
placement-??_???.dot  - graph layout at internal placement steps
routing-initial.map    - routing grid-map before the routing phase
routing-final.map      - routing grid-map after the routing phase
routing-??_???.map    - routing grid-map at internal routing steps
output.dot             - final layout after placement and routing

```

The following is a break-down of the processed result files:

```

placement-cost.eps    - graph of current vs overall minimal placement cost
placement-initial.gif - initial graph placement image
placement-final.gif   - final graph placement image
placement-??_???.gif - internal graph placement step images
routing-initial.map   - initial routing grid-map
routing-final.map     - final routing grid-map
routing-???.map       - combined grid-maps from internal routing steps
output.gif            - final layout image after placement and routing

```

The testbench makefile contains two hidden options which can either be set from the command-line or set at the top of the makefile. The SEED variable can be used to change the initial random seed for the mapping algorithm. The VIEWSTEPS variable if set to 'yes' will process intermediate states in addition to the initial and final states. Due to the large number of intermediate placement steps only 10 steps, evenly distributed from each placement iteration, are processed.

In addition to running the testbench in single execution mode, you can use the 'batch.sh' script to run many consecutive executions. This script, by default, iterates over a range of random seeds and saves the resulting layouts as *.gif files and the mapping algorithm output as *.log files. These files are placed in the batch directory as they are created. The batch script will display some simple statistics about each execution while it is running.

To make all the processing work, the testing directory contains a number of helper scripts and other miscellaneous files. The various tools used by these scripts are listed in the tools section. Below is a brief explanation of each file:

```

Makefile           - contains various targets to help with debugging
batch/*            - results from batch execution are placed here
debug/*            - debug information is placed here
graphs/*.dot       - a collection of various test graph files
results/*          - processed results are placed here
scripts/batch.sh   - shell script for running the testbench in batch
scripts/cost.plot  - gnuplot script for rendering the cost curve
scripts/dot2gif.sh - graphviz helper for rendering the *.dot files
scripts/gendot.py  - python script to generate random *.dot files
scripts/graphs.pl  - perl script for rendering placement graphs
scripts/routes.pl  - perl script for combining routing grid-maps

```

***** User Cost Function *****

The mapping library has the ability to use a custom user cost function. This custom user cost function allows the user to include an additional cost based on the working array and the target array. Typical usage for the user cost function is to calculate a cost by comparing a vertex property against its target array location. One example is to compare the load of a vertex to the frequency of the target location. There are three parts to the user cost function. The first part is the code for calculating the cost addition. The second part is the userdata vertex properties, which contain data values for each task that is being assigned. The final part is the userdata gridmap values, which contains data values for each processor in the target array.

To enable the user cost function change 'Rules.make' so that the variable 'USERCOST' is set to 'y'. This will set the build flags needed to include the user cost function code. To change the user cost function modify the code inside the function 'userdata_cost' in the file 'usercost.cpp' so that it calculates the desired cost addition. The remainder of this section will explain how to set the userdata values that are used by 'userdata_cost' and the functions used for reading back the userdata values.

The userdata vertex properties are stored in hash tables (std::map) using strings as keys and integers as values. This allows you to lookup values using strings instead of array index values. By using hash tables you can store an unlimited number of values. The userdata gridmap values are stored as an array of these hash values, using processor locations and strings for looking up stored values.

The following API is used to assign values to the userdata vertex properties. This is done in a similar fashion to that of setting the coordinate or source vertex properties.

```

vertex_t vertex;          // previously defined vertex descriptor
std::string hash_key;    // previously defined hash string key
int hash_value;          // previously defined hash integer value
userdata_map_t userdata_map = get(

```

```

    vertex_userdata,    // property map tag
    graph              // graph object
);
userdata_t &userdata = userdata_map[vertex]; // get vertex hash table
userdata[hash_key] = hash_value;           // set hash value

```

When using the mapping tool, these values are assigned by the tool using the following syntax in either ASAP Assembly files or ASAP XML files:

=> ASAP assembly

```

begin x,y
  #userdata(name) value
  // remaining instructions
end

```

=> ASAP XML

```

<module name="">
  <array size="">
    <processor loc="x,y">
      <code file="">
        <userdata name="xxx" value="xxx"/>
      </code>
    </processor>
  </array>
</module>

```

The following function is used to assign userdata gridmap values for the target array. This can be done manually but that method is not described here as the preferred method is using an userdata file and the accompanying parser. The results are stored in the 'userdata' and 'userdata_size' fields of the configuration object.

```

bool // returns true unless some failure occurred
parse_datafile (
  std::string datafile, // path to the userdata file
  configuration_t &config // configuration object to store data
);

```

The 'parse_datafile' function expects the input file to follow the format described below. Any line which begins with a pound (#) character or contains only whitespace will be ignored. The first processed line (not ignored) contains the size of the target array and the hash key names for each column. The lines following the header contain X,Y locations in the array that the values in the adjacent columns will be assigned to.

=> Userdata file format

```

SX,SY Header1 Header2
X1,Y1 Value1 Value2
X2,Y2 ValueA ValueB

```

When using the mapping tool, this datafile can be set using the 'Mapping

Execute' dialog under the 'Advanced' tab or using the '-f' option in batch mode.

The following functions can be used to read back the stored userdata values inside the user cost function. For an example of using these functions look at the commented code inside the user cost function. NOTE: The key values when stored by the parser and mapping tool are converted to lowercase to avoid case sensitivities but not converted to lowercase when values are retrieved to save cpu time, so passed keys should be lowercase.

=> Get vertex property value

```
int // retrieved userdata value
graph_value (
    graph_t &graph,           // graph containing the vertex
    vertex_t vertex,         // vertex containing the value
    std::string map_key,     // hash key to lookup value
    bool &success            // set to false if not found
);
```

=> Get target gridmap value

```
int // retrieved userdata value
grid_value (
    configuration_t &config, // object containing values
    coordinate_t coord,     // location containing the value
    std::string map_key,    // hash key to lookup value
    bool &success           // set to false if not found
);
```

When writing a custom user cost function, the runtime of the function must be taken into consideration. The placement cost function is called anytime a task is reassigned to a new location within the array (this happens often).

***** Tools *****

The following tools are used for compiling or debugging the mapping library:

=> Boost

Description: generic programming libraries using C++ templates
 Website: <http://www.boost.org/>
 Version: 1.33+
 Usage:

This mapping library uses various parts of this collection of libraries, the most prevalent being the boost graph library (BGL). In addition the POSIX time, formatting, and other libraries are used.

=> GNUPlot

Description: plotting tool for viewing X-Y data files
 Website: <http://www.gnuplot.info/>
 Version: 4.0+
 Usage:

This is used to render graphs of the cost function using data generated by

running the testbench in debug mode. It is very configurable and can save to a number of file formats, eps being the one used here.

=> GraphViz

Description: a collection of tools and libraries for drawing graphs

Website: <http://www.graphviz.org/>

Version: 2.4+

Usage:

These tools are used to render images of the *.dot files generated when running the testbench in debug mode. The *.dot files are rendered to *.gif files using the neato program from this package. This package has an insane amount of options, graph layout algorithms, and export capabilities, including eps.

Appendix B

Readme - AsAP Mapping Tool

Contained within this appendix is the contents of the `Readme` file associated with the AsAP mapping tool. This file can be found inside the AsAP mapping tool source code archive. This `Readme` file was written to provide the end user with instructions on compiling the AsAP mapping tool and basic operating instructions for batch mode execution. A general overview of the internal data structures is also provided along with a brief description of the contents within each source code file.

=== Documentation for the AsAP Mapping Tool ===

Author: Eric Work
E-mail: ewwork@ucdavis.edu
Modified: February 2007

***** Overview *****

The mapping tool is designed primarily as an interface between XML modules and the mapping library. The mapping tool has both a graphical mode and a batch mode. The graphical mode makes it easy to construct and analyze complex applications using an intuitive interface. Batch mode can be used to iterate over an assortment of parameters in an automated fashion to improve mapping quality. The key advantage to using XML modules is that they can be converted to and from programmable AsAP code which can be simulated. These XML modules can also be used by other tools to avoid parsing AsAP code. For more details on using the mapping tool refer to the help file or from the command-line type `'asapmap -h'`.

The mapping tool goes through three phases when mapping applications. The first phase populates the module list and connects processors together. The second phase performs the mapping and analyzes the results. The third phase translates the involved module files and combines all the processors into one large XML file.

The first phase is done by either dragging items to the canvas and linking them with the mouse, or by loading a project file. Module files, which are loaded from project files or dragged onto the canvas, describe the connections between local processors. These connections are based on matching pairs of opposing ports. Processors are linked globally by looking at module port numbers and connecting them according to the links described by the project file or links drawn with the mouse. For batch mode, the only option is loading modules through a project file.

The second phase looks at the output ports for each processor across all modules (inputs are mirrored) to see which processors are connected. A vertex is added for each processor and an edge is added for each connected output port. The algorithm is then executed which computes the coordinates for each vertex. After coordinates have been assigned, new routers are instantiated and inserted into the module list like any other module. The original processor's location and port numbers are compared to calculated numbers to produce translation sets. These translation sets contain the new location and an array of port index numbers that transform the old port index to the new port index. Each time the distance between two processors is non-nearest neighbor a routing channel is created. When in graphical mode, the final array configuration is also displayed after mapping.

The third phase goes through all the modules and routers and loads their XML files one at a time. For each module loaded the parameters are updated, the processor locations are changed, and the direction masks are changed according to the port index map. After being updated the processor elements along with the code inside them are copied and attached to the array element of the output document. Finally after all processors have been copied, the routing channels are added and a new XML module is created.

The following is a break-down of the source file contents:

```
config.h    - contains file locations and various constraints
interface.c - callbacks for events triggered by the graphical interface
interface.h - prototypes for functions inside interface.c
main.c      - program entry point and command-line parsing
mapping.cpp - functions to prep the algorithm and analyze the results
mapping.h   - prototypes for functions inside mapping.cpp
module.c    - functions used for loading and saving modules
module.h    - prototypes for functions inside module.c
project.c   - functions used for loading and saving projects
project.h   - prototypes for functions inside project.c
translate.c - functions used to change XML documents based on results
translate.h - prototypes for functions inside translate.c
types.h     - core data-types used throughout the mapping tool
utility.c   - various utility functions including port manipulations
utility.h   - prototypes for functions inside utility.c
visual.c    - functions for handling visual aspects of the canvas
visual.h    - prototypes for functions inside visual.c
```

The following is a break-down of the resource file contents:

```
asapmap.glade - glade XML interface resource file
```



```

asapmap.gladep - supporting glade project file
asapmap.xml    - program help file in DocBook format
asapmap.xpm    - XPM format application icon file
figures/*      - figures used by the help file
mkhtml.sh     - helper script to make HTML help files

```

***** Compiling the Program *****

To compile the mapping tool you will need to have the boost libraries, and the gnome development platform installed. The boost libraries are used to interface with the mapping library. The gnome platform is used to create the graphical user interface. In particular you will need the following libraries from the gnome platform: gtk2, libgnomeui2, libgnomecanvas2, libglade2, and libxml2. The gnome platform is standard on most recent Linux distributions but may require installing development packages.

You need to compile the mapping library before compiling the mapping tool. For compiling the mapping library refer to the 'Readme' file included in the mapping library source directory. After the library has been compiled, its location can be specified by changing the LIBAMAP_XXX variables in 'Rules.make'. The BOOST_XXX variables in 'Rules.make' should be changed to match the values used for compiling the mapping library. The 'pkg-config' utility is used to locate the gnome development libraries.

The location where the application data files reside can be changed using the DATADIR and MODULEDIR variables inside 'Rules.make'. The DATADIR variable points to the directory where the glade files and helps files reside. This is the 'resources' directory in the program source directory. The MODULEDIR variable points to the base directory where *.mod files are stored. Relative paths used by project files are based on this variable.

To compile the mapping tool type: make all

If the mapping library has been compiled in debug mode, it will generate debug files in exactly the same manor as the mapping algorithm testbench. In addition when the mapping frontend is compiled in debug mode an 'input.dot' file is generated which allows you to view the graph used as the input to the library. When running in debug mode you may find it helpful to copy some of the scripts from the 'testing' directory from the mapping library source directory.

***** Batch Mode Testing *****

Batch mode is designed to optimize applications over many trials using various parameters. To help with this, a collection of scripts and example projects can be found in the 'testing' directory. To get started, simply type 'make' in the 'testing' directory which will list some basic operations. The only requirement when executing the program from the makefile, is that an 'input.proj' file exist in the same directory as the makefile, or where the program will be executed. The requirements for this project file are the same as those required by graphical mode (primarily setting the input/output).

To create this input project file use graphical mode or symlink to one of the files in the 'projects' directory. Next run 'make execute' to generate the output module file called 'output.mod'. The output module can be viewed by either importing the module in graphical mode, or by running 'make analyze'. When viewing modules from the command-line, module view mode is used for displaying the array contents. There is one hidden option inside the makefile, that is the random seed value. This can be changed by either modifying the makefile and changing the value of SEED, or by appending 'SEED=val' when calling 'make execute'.

Below is a quick example of how to execute batch mode:

1. ln -s projects/wireless.proj input.proj
2. make execute analyze SEED=123

The 'batch.sh' script can be used to run a number of consecutive trials which varies the random seed over a given range. This script by default tries the range from 0 to 10, but this can be changed using the first two command-line arguments. The 'ASAPMAP_ARGS' variable inside the script can be modified to set the static mapping parameters. Resulting modules will be saved in the batch directory as they are created, as well as any log files. The batch script will display some simple statistics about each execution while it is running.

The testing directory contains a number of helper scripts and other miscellaneous files. Below is a brief explanation of each file:

Makefile	- contains various batch mode execution targets
batch/*	- results from consecutive trials are placed here
batch-ex/*	- results from batch mode exclusion are placed here
projects/*.proj	- a collection of various example project files
projects/*-data.proj	- a collection of project files using annotations
scripts/batch-ex1.sh	- shell script for sequential excluded coordinates
scripts/batch-exN.sh	- shell script for lists of excluded coordinates
scripts/batch.sh	- shell script for running consecutive trials
scripts/ex1plot.py	- python script for plotting single exclusion data
scripts/excludechg.py	- python script to change excluded coordinates
scripts/findbest.pl	- perl script to find the lowest cost module
scripts/genproject.py	- python script for creating random project files
scripts/genrandex.pl	- perl script to generate excluded coordinate files
scripts/getperform.py	- python script to calculate throughput and power
scripts/getprofile.pl	- perl script to extract runtime information
scripts/getsummary.pl	- perl script to extract mean and base values
scripts/profile.plot	- gnuplot script for plotting runtime information
scripts/summary.plot	- gnuplot script for plotting mean and base values
scripts/usercost.data	- user cost function array data file

***** Changing the Architecture *****

One of the primary operations of the mapping tool is to move processors while maintaining connections. This is done by changing port indexes. At the very core of the mapping tool is the module structure it's array of processor structures and each processor's many port structures. These data types are all

defined in 'types.h', as well as many other types used by the interface. Any changes to the architecture must follow the assumptions that all ports are point-to-point connections, the project has exactly one input and one output, and there are no self-loops or parallel-edges, in order to be mapped.

As expected, processor ports are very important to the core operation of the entire mapping process. Each processor has a set number of input and output ports defined by NUM_PORTS. This value comes from the number of items in the port direction mapping enumeration. Each enumeration entry corresponds to a character in the mask attribute. When loading an XML file these mask characters are converted to indexes to determine which ports are being connected together. When translating module files after mapping, these indexes are converted back to characters to update the mask attributes. Port index related functions are found in 'utility.c', which are what specifies the architecture being mapped. These functions include finding the port for an adjacent processor, finding the best port between two processor locations, and others. These indexes are also used to draw the connecting arrows between processors in the array. For the ASAP architecture the rule is that cardinal ports are nearest neighbor and the direction ports are non-nearest neighbor.

Port indexes determine the communication between processors, but routers determine the mapping flexibility of the architecture. An important step of the mapping process is the insertion of routing processors. These routing processors are loaded from the module files in 'MODULEDIR/routers'. The one-way router is loaded from 'oneway.mod' and the two-way router is loaded from 'twoway.mod'. To change the code inside these routing processors, for example to add buffering, simply change or replace the module files at these locations. The one-way routing module must have exactly one processor, one input, and one output. The two-way routing module must have exactly one processor, two inputs, and two outputs.

***** Libraries *****

The following libraries are used in various parts of the mapping tool.

=> GTK+

Description: library for creating graphical user interfaces
Reference: <http://developer.gnome.org/doc/API/2.0/gtk/index.html>
Version 2.x
Usage:

This library is used for creating the graphical user interface using various widgets from the library to handle user input and output. These widgets include windows, menus, buttons, lists, labels, etc. A related sub-library, glib, is used for directory manipulation, string manipulation, and managing linked-lists.

=> LibGnomeUI

Description: library of functions typically required for applications
Reference: <http://developer.gnome.org/doc/API/2.0/libgnomeui/index.html>
Version: 2.x
Usage:

This library is used for the module icon list, the about box, and setting the window title among other things. This library is built on top of

the GTK+ library and its widgets.

=> LibGnomeCanvas

Description: an object-oriented, event driven canvas widget

Reference: <http://developer.gnome.org/doc/API/2.0/libgnomecanvas/index.html>

Version: 2.x

Usage:

This library is the core of the interactive canvas that displays the modules. It provides basic drawing elements that can be attached to event handlers when clicked, or otherwise manipulated with the mouse or keyboard. This library is built on top of various gnome libraries.

=> LibGlade

Description: loader for glade XML user interface resource files

Reference: <http://developer.gnome.org/doc/API/2.0/libglade/index.html>

Version: 2.x

Usage:

This library is used to load interface files designed by the glade tool. All complex windows in the program are described inside these XML files and are created automatically based on how they were configured in the glade file. Glade files can change the program's appearance without recompiling the code.

=> LibXML

Description: library for working with XML files

Reference: <http://xmlsoft.org/html/libxml-lib.html>

Version: 2.x

Usage:

This library is used to load, store, and manipulate module and project XML files. In this program the document object model (DOM) is used for reading and writing XML files. XPath, a libxml component, is used for easily traversing the DOM tree and finding certain elements.

***** Tools *****

The following tools are used to create resources used by the mapping tool.

=> Glade

Description: WYSIWYG tool for designing GTK-based interfaces

Website: <http://glade.gnome.org/>

Version: 2.x

Usage:

This tool was used to create the user interface for various parts of the program. The output is an XML file containing an arrangement of GTK widgets and properties which is used to dynamically create the interface during program execution. One advantage is that the interface can be changed without re-compiling the code.

=> Inkscape

Description: vector-based drawing program

Website: <http://www.inkscape.org/>

Usage:

This program was used to create the module icons as well as the application

icon. This program works directly with SVG files and can export to PNG format.

=> DocBook

Description: a flexible XML/SGML language for writing documentation

Website: <http://www.docbook.org/>

Usage:

This is the format used for creating the help file, which is compatible with the gnome help viewer (yelp). DocBook can be also converted to HTML, LaTeX, and a number of other formats using OpenJade and other tools. The docbook2html tool is used to create HTML help files.

Glossary

AsAP :

Acronym for *Asynchronous Array of Simple Processors*. A 2D-mesh parallel array architecture designed for power efficiency while executing computationally intensive applications.

AsAP Version 1 :

This is the first implementation of the **AsAP** architecture which has 36 processing elements arranged in a 6x6 array with one input in the upper-left corner and one output which can be any one of the right edge processors. This version of the architecture uses nearest neighbor communication exclusively.

AsAP Version 2 :

This is the second implementation of the **AsAP** architecture which has an array of size 13x13 with a few of the lower processors replaced by hardware-based accelerators. For this work the array is assumed to be homogeneous with a size of 16x16. This version of the architecture introduces a routing overlay network and also allows the input processor to be any one of the left edge processors.

Base Number of Long-Distance Interconnects :

Only valid for tests involving **Excluded Processors**. This is the **Minimum Number of Long-Distance Interconnects** when no processors are excluded.

Base Number of Routing Processors :

Only valid for tests involving **Excluded Processors**. This is the **Minimum Number of Routing Processors** when no processors are excluded.

Base Rectangular Array Area :

Only valid for tests involving **Excluded Processors**. This is the **Minimum Rectangular Array Area** when no processors are excluded.

Configuration Cost :

The result from the `ConfigCost` function, from the placement phase of the mapping algorithm, which calculates the cost associated with a particular simulated annealing configuration. This value is used to determine if one configuration is better than another when deciding whether or not to accept a perturbation.

Configuration Parameters :

A collection of values, packed into one data structure, that controls what components of the mapping algorithm get executed and how the application is optimized.

Critical Processor :

This is a processor within the target array where the maximum frequency is greater than or equal to the maximum frequency of all other processors and the leakage current is less than or equal to the leakage current of all other processors.

Critical Task :

This is a task within the application where the load average is greater than or equal to the load average of all other tasks and the activity level is greater than or equal to the activity level of all other tasks.

Current Configuration Cost :

Valid only during the placement phase of the mapping algorithm. The `Configuration Cost` for the configuration currently being perturbed at intermediate points of the placement phase.

Dataflow Graph :

A series of edges and vertices that describe the structure of an application. Vertices represent tasks within the application and edges represent dependencies between tasks.

Enclosed Array Area :

The `Rectangular Array Area` for a mapping ignoring processors near the perimeter which are not enclosed by the mapping. Any empty location with less than three neighboring tasks that is not completely surrounded by tasks is ignored.

Excluded Processor :

Sometimes called an `Excluded Location`. A processor at a predefined location within the target array which is not allowed to have a task assigned to it.

Long-Distance Interconnects :

Sometimes called a **Routing Channel**. A connection between two tasks which are not nearest neighbors that couldn't be routed during the routing phase due to conflicts (or if the routing phase was disabled).

Mean Number of Long-Distance Interconnects :

Only valid for tests involving **Excluded Processors**. The average of the **Minimum Number of Long-Distance Interconnects** for each set of **Excluded Processors**.

Mean Number of Routing Processors :

Only valid for tests involving **Excluded Processors**. The average of the **Minimum Number of Routing Processors** for each set of **Excluded Processors**.

Mean Rectangular Array Area :

Only valid for tests involving **Excluded Processors**. The average of the **Minimum Rectangular Array Area** for each set of **Excluded Processors**.

Minimum Configuration Cost :

Valid only during the placement phase of the mapping algorithm. The lowest **Configuration Cost** that has been observed since the beginning of the placement phase.

Minimum Number of Long-Distance Interconnects :

This is the lowest number of **Long-Distance Interconnects** observed after running a number of trials on an application using the same configuration parameters and **Excluded Processors**.

Minimum Number of Routing Processors :

This is the lowest number of **Routing Processors** observed after running a number of trials on an application using the same configuration parameters and **Excluded Processors**.

Minimum Rectangular Array Area :

This is the lowest **Rectangular Array Area** observed after running a number of trials on an application using the same configuration parameters and **Excluded Processors**.

Optimal Rectangular Array Area :

This is the smallest **Rectangular Array Area** that will fit every node in the graph, excluding routing processors and ignoring data dependencies. This value is calculated by Equations 5.1, 5.2, and 5.3 on page 81.

Optimization Cost :

A numerical value, calculated by Equation 5.4 on page 84, that combines the various quality metrics, communication, array area, and utilization. This value is used to quickly compare the quality between different mappings.

Parameter, AddSpacing :

This parameter will enable or disable the spacing flag. When enabled empty space will be inserted at the beginning of the routing phase if the spacing threshold is exceeded.

Parameter, ExpandType :

This parameter controls which type of expansion is used by the initial placement function. When using vertical expansion processors are placed top to bottom first then left to right. When using horizontal expansion processors are placed left to right first then top to bottom.

Parameter, InputEdge :

This parameter determines which edge to use when assigning the array input task to a processor. Possible values for this parameter are Left, Right, Top, Bottom, or None.

Parameter, MaxRoutes :

This parameter determines the maximum number of routes that are allowed to pass through a single **Routing Processor**. For **AsAP** this value can be no greater than two.

Parameter, NumIters :

This parameter determines the number of temperature schedules to execute during the placement phase.

Parameter, OutputEdge :

This parameter determines which edge to use when assigning the array output task to a processor. Possible values for this parameter are Left, Right, Top, Bottom, or None.

Parameter, QuickPlace :

This parameter will enable or disable the estimation flag. When enabled the placement phase will terminate early when there appears to be little chance of improvement.

Parameter, SpaceThreshold :

This parameter determines whether or not space needs to be inserted. The threshold is the ratio of the number of edges to the number of vertices expressed as a percentage. When the result is above this threshold space will be inserted if enabled.

Parameter, UseRouting :

This parameter will enable or disable the routing flag. When disabled router insertion paths will not be calculated during the placement phase and the routing phase will be skipped.

Rectangular Array Area :

The product of the array width and the array height for a given mapping.

Routing Processors :

Sometimes called a **Router**. A task or vertex added to the dataflow graph whose sole purpose is to route data between adjacent vertices.

Trial :

One complete execution of the mapping algorithm which produces one possible mapping for an application. Trials are usually executed in blocks using the same **Configuration Parameters** but different random seeds.

Bibliography

- [1] Boost C++ libraries. <http://www.boost.org>.
- [2] GNOME: The free desktop project. <http://www.gnome.org>.
- [3] Gnomelove Drag-N-Drop tutorial. <http://live.gnome.org/GnomeLove/DragNDropTutorial>.
- [4] Anant Agarwal. Raw computation. *Scientific American*, pages 44–47, Aug 1999.
- [5] Ishfaq Ahmad, Yu-Kwong Kwok, Min-You Wu, and Wei Shu. CASCH: A tool for computer-aided scheduling. *IEEE Concurrency*, 8(4):21–33, Oct 2000.
- [6] Ryan W. Apperson. A dual-clock FIFO for the reliable transfer of high-throughput data between unrelated clock domains. Master’s thesis, University of California, Davis, Davis, CA, USA, Sep 2004.
- [7] Bevan M. Baas. A parallel programmable energy-efficient architecture for computationally-intensive DSP systems. In *37th Asilomar Conference on Signals, Systems and Computers*, Nov 2003.
- [8] Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, Thomas Gross, H. T. Kung, Monica Lam, Brian Moore, Craig Peterson, John Pieper, Linda Rankin, P. S. Tseng, Jim Sutton, John Urbanski, and John Webb. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing*, volume 1, pages 330–339, Orlando, FL, USA, Nov 1988.
- [9] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, and William Yoder. Scaling to the end of silicon with EDGE architectures. *Computer Magazine*, 37(7):44–55, Jul 2004.
- [10] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, Stanford, CA, USA, 1984.
- [11] Tilera Corporation. The tile processor architecture: Embedded multicore for networking and digital multimedia. In *IEEE HotChips Symposium on High-Performance Chips 2007*, Aug 2007.
- [12] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.
- [13] Sabih H. Gerez. *Algorithms for VLSI Design Automation*. John Wiley & Sons, Inc., New York, NY, USA, 1999.
- [14] Michael I. Gordon. A stream-aware compiler for communication-exposed architectures. Master’s thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, Aug 2002.
- [15] Micheal I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *Architectural Support for Programming Languages and Operating Systems*, volume 10, pages 291–303, San Jose, CA, USA, Oct 2002.

- [16] Jingcao Hu and Radu Marculescu. Energy-aware mapping for tile-based NoC architectures under performance constraints. In *Asia South Pacific Design Automation Conference*, pages 233–239, Kitakyushu, Japan, Jan 2003.
- [17] Ujval J. Kapasi, Peter Mattson, William J. Dally, John D. Owens, and Brian Towles. Stream scheduling. In *Proceedings of the 3rd Workshop on Media and Streaming Processors*, pages 101–106, Austin, TX, USA, Dec 2001.
- [18] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, Mar 2001.
- [19] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.
- [20] Francois Labonte, Peter Mattson, Ian Buck, Christos Kozyrakis, and Mark Horowitz. The stream virtual machine. In *13th International Conference on Parallel Architectures and Compilation Techniques*, pages 267–277, Antibes Juan-les-Pins, France, Sep 2004.
- [21] C. Y. Lee. An algorithm for path connections and its applications. volume EC-10, pages 346–365, Sep 1961.
- [22] Guy G. Lemieux and Stephen D. Brown. A detailed routing algorithm for allocating wire segments in field-programmable gate arrays. In *ACM Physical Design Workshop*, pages 215–226, Lake Arrowhead, CA, USA, 1993.
- [23] Virginia M. Lo, Sanjay Rajopadhye, Samik Gupta, David Keldsen, Moataz A. Mohamed, Bill Nitzberg, Jan Arne Telle, and Xiaoxiong Zhong. OREGAMI: Tools for mapping parallel computations to parallel architectures. *International Journal of Parallel Programming*, 20(3):237–270, 1991.
- [24] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart Memories: A modular reconfigurable architecture. In *27th Annual International Symposium on Computer Architecture*, pages 161–171, Vancouver, British Columbia, Canada, Jun 2000.
- [25] Peter Mattson, William J. Dally, Scott Rixner, Ujval J. Kapasi, and John D. Owens. Communication scheduling. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 82–92, 2000.
- [26] Michael J. Meeuwsen, Omar Sattari, and Bevan M. Baas. A full-rate software implementation of an IEEE 802.11a compliant digital baseband transmitter. In *IEEE Workshop on Signal Processing Systems*, volume 19, pages 297–301, Oct 2004.
- [27] David R. O’Hallaron. The assign parallel program generator. In *Distributed Memory Computing Conference*, pages 178–185, Portland, OR, USA, Apr 1991.
- [28] A. V. Oppenheim and R. W. Schaffer. *Discrete-Time Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1989.
- [29] Frank Rubin. The Lee path connection algorithm. *IEEE Transactions on Computers*, C-23(9):907–914, Sep 1974.
- [30] Madhu Saravana, Sibi Govindan, Doug Burger, Steve Keckler, and the TRIPS Team. TRIPS: A distributed explicit data graph execution (EDGE) microprocessor. In *IEEE HotChips Symposium on High-Performance Chips 2007*, Aug 2007.
- [31] K. Shahookar and P. Mazumder. VLSI cell placement techniques. *ACM Computing Surveys*, 23(2):143–220, Jun 1991.

- [32] Aaron Smith, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, Kathryn S. McKinley, and Jim Burrill. Compiling for EDGE architectures. In *International Symposium on Code Generation and Optimization*, pages 185–195, New York, NY, USA, Mar 2006.
- [33] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrati, Ben Greenwald, Henry Hoffman, Paul Johnson, Walter Lee, Arvind Saraf, Nathan Shnidman, Volker Strumpfen, Saman Amarasinghe, and Anant Agarwal. A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network. In *IEEE International Solid-State Circuits Conference*, pages 170–171, San Francisco, CA, USA, Feb 2003.
- [34] Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, Apr 1967.
- [35] Min-You Wu and Daniel D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, Jul 1990.
- [36] Tao Yang and Apostolos Gerasoulis. PYRROS: Static task scheduling and code generation for message passing multiprocessors. In *International Conference on Supercomputing*, pages 428–437, New York, NY, USA, Jul 1992.
- [37] Zhiyi Yu, Michael Meeuwsen, Ryan Apperson, Omar Sattari, Michael Lai, Jeremy Webb, Eric Work, Tinoosh Mohsenin, Mandeep Singh, and Bevan Baas. An asynchronous array of simple processors for DSP applications. In *IEEE International Solid-State Circuits Conference*, pages 428–429, San Francisco, CA, USA, Feb 2006.

