3

An Instruction Scheduling Algorithm for Communication–Constrained Microprocessors

by

Christopher James Buehler

B.S.E.E., B.S.C.S. (1996)
University of Maryland, College Park

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1998

[illegible]

Author . . . [illegible] . . . .
Department of Electrical Engineering and Computer Science
August 7, 1998

Certified by [illegible] . . . .
William J. Dally
Professor
Thesis Supervisor

Accepted by . . . . [illegible] . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Eng.

An Instruction Scheduling Algorithm for Communication–Constrained Microprocessors

by

Christopher James Buehler

Submitted to the Department of Electrical Engineering and Computer Science
on August 7, 1998, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

This thesis describes a new randomized instruction scheduling algorithm designed for communication–constrained VLIW–style machines. The algorithm was implemented in a retargetable compiler system for testing on a variety a different machine configurations. The algorithm performed acceptably well for machines with full communication, but did not perform up to expectations in the communication–constrained case. Parameter studies were conducted to ascertain the reason for inconsistent results.

Thesis Supervisor: William J. Dally
Title: Professor

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As VLSI circuit density increases, it becomes possible for microprocessor designers to place more and more logic on a single chip. Studies of instruction level parallelism suggest that this logic may be best spent on exploiting fine–grained parallelism with numerous, pipelined functional units [4, 3]. However, while it is fairly trivial to scale the sheer number of functional units on a chip, other considerations limit the effectiveness of this approach. As many researchers point out, communication resources to support many functional units, such as multi–ported register files and large interconnection networks, do not scale so gracefully [16, 6, 5]. Furthermore, these communication resources occupy significant amounts of chip area, heavily influencing the overall cost of the chip. Thus, to accommodate large numbers of functional units, hardware designers must use non–ideal approaches, such as partitioned register files and limited interconnections between functional units, to limit communication resources.

Such communication–constrained machines boast huge amounts of potential parallelism, but their limited communication resources present a problem to compiler writers. Typical machines of this nature (e.g., VLIWs) shift the burden of instruction scheduling to the compiler. For these highly–parallel machines, efficient static instruction scheduling is crucial to realize maximum performance. However, many traditional static scheduling algorithms fail when faced with communication–constrained machines.

1.1 Traditional Instruction Scheduling

Instruction scheduling is an instance of the general resource constrained scheduling (RCS) problem. RCS involves sequencing a set of tasks that use limited resources. The resulting sequence must satisfy both task precedence constraints and limited resource constraints [2]. In instruction scheduling, instructions are tasks, data dependencies are precedence constraints, and hardware resource are machine resources.

RCS is a well–known NP–complete problem, motivating the development of many heuristics for instruction scheduling. One of the most commonly used VLIW scheduling heuristics is list scheduling [8, 7, 6, 11, 18]. List scheduling is a locally greedy algorithm that maintains an prioritized "ready list" of instructions whose precedence constraints have been satisfied. On each execution cycle, the algorithm schedules instructions from the list until functional unit resources are exhausted or no instructions remain.

List scheduling explicitly observes the limited functional unit resources of the target machine, but assumes that the machine has infinite communication resources. This assumption presents a problem when implementing list scheduling on communication–constrained machines. For example, its locally greedy decisions can consume key communication resources, causing instructions to become "stranded" with no way to access needed data. In light of these problems, algorithms are needed that operate more globally and consider both functional unit and communication resources in the scheduling process. It is proposed in this thesis that randomized instruction scheduling algorithms might fulfill these needs.

1.2 Randomized Instruction Scheduling

The instruction scheduling problem can also be considered a large combinatorial optimization problem. The idea is to systematically search for a schedule that optimizes some cost function, such as the length of the schedule. Many combinatorial optimization algorithms are random in nature. Popular ones include hill–climbing, random sampling, genetic algorithms, and simulated annealing.

Combinatorial optimization algorithms offer some potential advantages over traditional deterministic scheduling algorithms. First, they consider a vastly larger number of schedules, so they should be more likely to find an optimal schedule. Second, they operate on a global scale and do not get hung up on locally bad decisions. Third, they can be tailored to optimize for any conceivable cost function instead of just schedule length. And finally, they can consider any and all types of limited machine resources, including both functional unit and communication constraints. The primary disadvantage is that they can take longer to run, up to three orders of magnitude longer than list scheduling.

In this thesis, an implementation of the simulated annealing algorithm is investigated as a potential randomized instruction scheduling algorithm. The results indicate that this implementation may not be the best choice for a randomized instruction scheduling algorithm. While the algorithm performs consistently well on communication–rich machines, it often fails to find good schedules for its intended targets, communication–constrained machines.

This thesis presents the results of systematic studies designed to find good parameters for the simulated annealing algorithm. The algorithm is extensively tested on a small sampling of programs and communication–constrained machines for which it is expected to perform well. These studies identify some parameter trends that influence the algorithm's performance, but no parameters gave consistently good results for all programs on all machines. In particular, machines with more severe communication constraints elicited poorer schedules from the algorithm.

1.3 Background

Many modern instruction scheduling algorithms for VLIW ("horizontal") machines find their roots in early microcode compaction algorithms. Davidson et al. [7] compare four such algorithms: first–come–first–served, critical path, branch–and–bound, and list scheduling. They find that first–come–first–served and list scheduling often perform optimally and that branch–and–bound is impractical for large microprograms. Tokoro, Tamura, and Takizuka [19] describe a more sophisticated microcode compaction algorithm in which microinstructions are treated as 2–D templates arranged on a grid composed of machine resources vs. cycles. The scheduling process is reduced to tessellation of the grid with variable–sized 2–D microinstruction templates. They provide rules for both local and global optimization of template placement.

Researchers recognized early on that that *global* scheduling algorithms are necessary for maximum compaction. Isoda, Kobayashi, and Ishida [9] describe a global scheduling technique based on the generalized data dependency graph (GDDG). The GDDG represents both data dependencies and control flow dependencies of a microprogram. Local GDDG transformation rules are applied in a systematic manner to compact the GDDG into an efficient microprogram. Fisher [8] also acknowledges the importance of global microcode compaction in his trace scheduling technique. In trace scheduling, microcode is compacted along traces rather than within basic blocks. Traces are probable execution paths through a program that generally contain many more instructions than a single basic block, allowing more compaction options.

Modern VLIW instruction scheduling efforts have borrowed some microcode compaction ideas while generating many novel approaches. Colwell et al. [6] describe the use of trace scheduling in a compiler for a commercial VLIW machine. Lam [11] develops a VLIW loop scheduling technique called software pipelining, also described earlier by Rau [15]. In software pipelining, copies of loop iterations are overlapped at constant intervals to provide optimal loop throughput. Nicolau [13] describes percolation scheduling, which utilizes a small core set of local transformations to parallelize programs. Moon and Ebcioğlu [12] describe a global VLIW scheduling method based on global versions of the basic percolation scheduling transformations.

Other researchers have considered the effects of constrained hardware on the VLIW scheduling problem. Rau, Glaeser, and Picard [16] discuss the complexity of scheduling for a practical horizontal machine with many functional units, separate "scratch–pad" register files, and limited interconnect. In light of the difficulties, they conclude that the best solution is to change the hardware rather than invent better scheduling algorithms. The result is their "polycyclic" architecture, an easily schedulable VLIW architecture. Capitanio, Dutt, and Nicolau [5] also discuss scheduling algorithms for machines with distributed register files. Their approach utilizes simulated annealing to partition code across hardware resources and conventional scheduling algorithms to schedule the resulting partitioned code. Smith, Horowitz, and Lam [17] describe a architectural technique called "boosting" that exposes speculative execution hardware to the compiler. Boosting allows a static instruction scheduler to exploit unique code transformations made possible by speculative execution.

1.4 Thesis Overview

This thesis is organized into six chapters. Chapter 1 contains the introduction, a survey of related research, and this overview.

Chapter 2 gives a high–level overview of the scheduler test system. The source input language *μasm* is described as well as the class of machines for which the scheduler is intended.

Chapter 3 introduces the main data structure of the scheduler system, the program graph, and outlines the algorithms used to construct it.

Chapter 4 outlines the generic simulated annealing search algorithm and how it is applied in this case for instruction scheduling.

Chapter 5 presents the results of parameter studies with the simulated annealing scheduling algorithm. It also provides some analysis of the data and some explanations for its observed performance.

Chapter 6 contains the conclusion and suggestions for some areas of further work.

Chapter 2

Scheduler Test System

The scheduler test system was developed to evaluate instruction scheduling algorithms on a variety of microprocessors. As shown in Figure 2-1, the system is organized into three phases: **parse**, **analysis**, and **schedule**.

The **parse** phase accepts a user–generated program as input. This program is written in a high–level source language, *μasm*, which is described in Section 2.1 of this chapter. Barring any errors in the source file, the **parse** phase outputs a sequence of machine–independent assembly instructions. The mnemonics and formats of these assembly instructions are listed in Appendix B.

The **analysis** phase takes the sequence of assembly instructions from the **parse** phase as its input. The sequence is analyzed using simple dataflow techniques to infer data dependencies and to expose parallelism in the code. These analyses are used to construct the sequence's *program graph*, a data structure that can represent data dependencies and control flow for simple programs. The analyses and algorithms used to construct the program graph are described in detail in Chapter 3.

The **schedule** phase has two inputs: a *machine description*, written by the user, and a program graph, produced by the **analysis** phase. The machine description specifies the processor for which the scheduler generates code. The scheduler can target a certain class of processors, which is described in Section 2.2 of this chapter. During the **schedule** phase, the instructions represented by the program graph are placed into a schedule that satisfies all the data dependencies and respects the limited

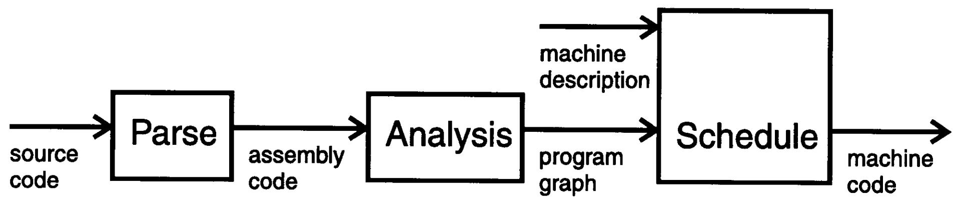


Figure 2-1: Scheduler test system block diagram.

resources of the target machine. The **schedule** phase outputs a scheduled sequence of wide instruction words, the final output of the scheduler test system.

The **schedule** phase can utilize many different scheduling algorithms. The simulated annealing instruction scheduling algorithm, the focus of this thesis, is described in Chapter 4.

2.1 Source Language

The scheduler test system uses a simple language called *μasm* (micro–assembler) to describe its input programs. The *μasm* language is a high–level, strongly–typed language designed to support "streaming computations" on a VLIW style machine. It borrows many syntactic features from the C language including variable declarations, expression syntax, and infix operators. The following sections detail specialized language features that differ from those of C. The complete grammar specification of *μasm* can be found in Appendix A.

2.1.1 Types

Variables in *μasm* can have one of five base types: `int`, `half2`, `byte4`, `float`, or `cc`. These base types can be modified with the type qualifiers `unsigned` and `double`.

The base types `int` and `float` are 32–bit signed integer and floating point types. The base types `half2` and `byte4` are 32–bit quantities containing two signed 16–bit integers and 4 signed 8–bit integers, respectively. The `cc` type is a 1–bit condition code.

The type qualifier `unsigned` can be applied to any integer base type to convert it to an unsigned type. The type qualifier `double` can be applied to any arithmetic type to form a double width (64–bit) type.

2.1.2 I/O Streams

Streaming computations typically operate in compact loops and process large vectors of data called *streams.* Streams must be accessed sequentially, and they are designated as either read–only or write–only. *μasm* supports the stream processing concept with the special functions `istream` and `ostream`, used as follows:

variable = `istream`(*stream#*, *value-type*),

`ostream`(*stream#*, *value-type*) = *value.*

In the above, *variable* is a program variable, *value* is a value produced by an expression in the program, *stream #* is a number identifying a stream, and *value-type* is the type of the value to be read from or written to the stream.

2.1.3 Control Flow

In an effort to simplify compilation, *μasm* does not support the standard looping and conditional language constructs of C. Instead, *μasm* features control flow syntax which maps directly onto the generic class of VLIW hardware for which it is targeted.

Loops in *μasm* are controlled by the `loop` keyword as follows:

`loop` *loop-variable* = *start* , *finish* { *loop-body* },

where *loop-variable* is the loop counter, and *start* and *finish* are integers delineating the range of values (inclusive) for the loop counter.

All conditional expressions in *μasm* are handled by the `?:` conditional ternary operator, an operation naturally supported by the underlying hardware. The language has no if–then capability, requiring all control paths through the program to be executed. The conditional operator is used as follows:

value = *condition* ? *value1* : *value2*.

If *condition* is true, *value1* is assigned to *value*, otherwise *value2* is assigned to *value*. The *condition* variable must be of type `cc`.

2.1.4 Implicit Data Movement

Assignment expressions in *μasm* sometimes have a slightly different interpretation than those in C. When an expression that *creates* a value appears on the right–hand side of an assignment expression, the parser generates normal code for the assignment. However, if the right–hand side of an assignment expression merely *references* a value (e.g., a simple variable name), the parser translates the assignment into a data movement operation. For example, the assignment expression

```
a = b + c;
```

is left unchanged by the parser, as the expression `b + c` creates an unnamed intermediate value that is placed in the data location referenced by `a`. On the other hand, the expression

```
ostream(0,int) = d;
```

is implicitly converted to the expression

```
ostream(0,int) = pass(d);
```

in which the `pass` function creates a value on the right–hand side of the assignment. The `pass` function is an intrinsic *μasm* function that simply passes its input to its output. The `pass` function translates directly to the `pass` assembly instruction, which is used to move data between register files. The `pass` instruction also has special significance during instruction scheduling, as discussed in Chapter 4.

2.1.5 Example Program

An example *μasm* program is shown in Figure 2-2. The program processes two 100–element input streams and constructs a 100–element output stream. Each element

```
int elem0, elem1;
cc gr;

loop count = 0, 99                     // loop 100 times
{
  elem0 = istream(0,int);              // read element from stream 0
  elem1 = istream(1,int);              // read element from stream 1
  gr = elem0 > elem1;                  // which is greater?
  ostream(0,int) = gr ? elem0 : elem1; // output the greater
}
```

Figure 2-2: Example *μasm* program.

of the output stream is selected to be the greater of the two elements in the same positions of the two input streams.

2.2 Machine Description

The scheduler test system is designed to produce code for a strictly defined class of processors. Processors within this class are composed of only three types of components: functional units, register files, and busses. Functional units perform the computation of the processor, register files store intermediate results, and busses route data from functional units to register files. Processors are assumed to be clocked, and all data is one 32–bit "word" wide.

Each processor component has input and output ports with which they are connected to other components. Only certain connections are allowed: functional unit outputs must connect to bus inputs, bus outputs must connect to register file inputs, and register file outputs must connect to functional unit inputs. The general flow of data through such a processor is illustrated in Figure 2-3.

A processor may contain many different instances of each component type. The various parameters that distinguish components are described in Sections 2.2.1, 2.2.2, and 2.2.3.

While such a restrictive processor structure may seem artificially limiting, a wide

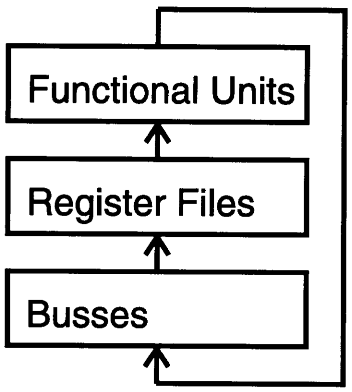


Figure 2-3: General structure of processor.

variety of sufficiently "realistic" processors can be modeled within these limitations. Examples are presented in Section 2.2.4.

2.2.1 Functional Units

Functional units operate on a set of input data words to produce a set of output data words. The numbers of input words and output words are determined by the number of input ports and output ports on the functional unit.

Functional unit operations correspond to the assembly instruction mnemonics listed in Appendix B. A functional unit may support anywhere from a single assembly instruction to the complete set.

A functional unit completes all of its operations in the same fixed amount of time, called the *latency*. Latency is measured in clock cycles, the basic unit of time used throughout the scheduler system. For example, if a functional unit with a 2 cycle latency reads inputs on cycle 8, then it produces outputs on cycle 10.

Functional units may be fully pipelined, or not pipelined at all. A fully pipelined unit can read a new set of input data words on every cycle, while a non–pipelined unit can only read inputs after all prior operations have completed.

In the machine description, a functional unit is completely specified by the number of input ports, the number of output ports, the latency of operation, the degree of pipelining, and a list of supported operations.

2.2.2 Register Files

Register files store intermediate results and serve as delay elements during computation. All registers are one data word wide. On each clock cycle, a register file can write multiple data words into its registers, and read multiple data words out of its registers. The numbers of input and output ports determine how many words can be written or read in a single cycle.

In the machine description, a register file is completely specified by the number of input ports, the number of output ports, and the number of registers contained within it.

2.2.3 Busses

Busses transmit data from the outputs of functional units to the inputs of register files. They are one data word wide, and provide instantaneous (0 cycle) transmission time. In this microprocessor model, bus latency is wrapped up in the latency of the functional units. Aside from the number of distinct busses, no additional parameters are necessary to describe busses in the machine description.

2.2.4 Example Machines

In this section, four example machine descriptions are presented. Each description is given in two parts: a list of component parameterizations and a diagram showing connectivity between components. For the sake of simplicity, it assumed that the possible set of functional unit operations is ADD, SUB, MUL, DIV, and SHFT. The basic characteristics of the four machines are summarized in Table 2.1.

The first machine is a simple scalar processor (Figure 2-4). It has one functional unit which supports all possible operations and a single large register file. The functional unit latency is chosen to be the latency of the longest instruction, DIV.

The second machine is a traditional VLIW machine with four functional units (Figure 2-5) [20]. This machine distributes operations across all four units, which have variable latencies. It has one large register file through which the functional

	# Functional Units	# Register Files	# Busses	Communication Connectedness
Scalar	1	1	2	FULL
Traditional VLIW	4	1	6	FULL
Distributed VLIW	4	8	6	FULL
Multiply–Add	4	8	5	CONSTRAINED

Table 2.1: Summary of example machine descriptions.

units can exchange data.

The third machine is VLIW machine with distributed register files and full interconnect (Figure 2-6). Functional units store data locally in small register files and route data through the bus network when results are needed by other units.

The fourth machine is a *communication–constrained* machine with an adder and a multiplier connected in a "multiply-add" configuration (Figure 2-7). Unlike the previous three machines, communication–constrained machines are not *fully–connected.* A fully–connected machine is a machine in which there is a *direct data path* from every functional unit output to every functional unit input. A direct data path starts at a functional unit output, connects to a bus, passes through a register file, and ends at a functional unit input. In this machine, data from the multiplier must pass through the adder before it can arrive at any other functional unit. Thus, there is no direct data path from the output of the multiplier to the input of any unit except the adder.

(#) Functional Units	**# ins**	**# outs**	**latency**	**pipe?**	**ops**
(1) PROCESSOR	2	2	10	NO	ADD, SUB, MUL, DIV, SHFT
(#) Register Files	**# ins**	**# outs**	**# regs**		
(1) REGFILE	2	2	32		
(#) Busses					
(2) BUS					

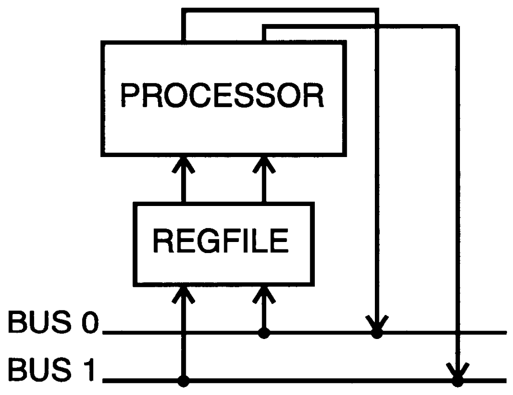


Figure 2-4: Simple scalar processor.

(#) Functional Units	# ins	# outs	latency	pipe?	ops
(1) ADDER	2	1	2	YES	ADD, SUB
(1) MULTIPLIER	2	2	3	YES	MUL
(1) DIVIDER	2	2	10	NO	DIV
(1) SHIFTER	2	1	1	YES	SHFT
(#) Register Files	**# ins**	**# outs**	**# regs**		
(1) REGFILE	6	8	32		
(#) Busses					
(6) BUS					

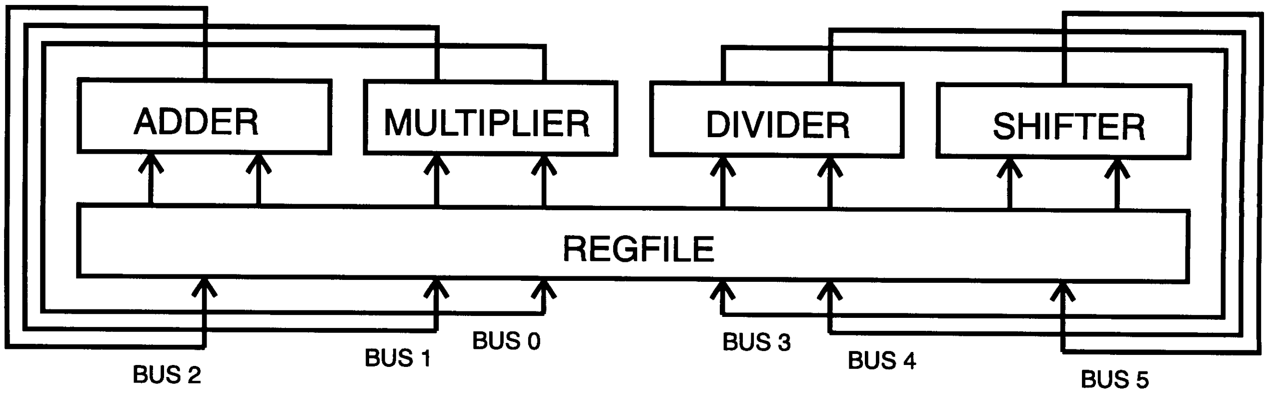


Figure 2-5: Traditional VLIW processor.

(#) Functional Units	**# ins**	**# outs**	**latency**	**pipe?**	**ops**
(1) ADDER	2	1	2	YES	ADD, SUB
(1) MULTIPLIER	2	2	3	YES	MUL
(1) DIVIDER	2	2	10	NO	DIV
(1) SHIFTER	2	1	1	YES	SHFT
(#) Register Files	**# ins**	**# outs**	**# regs**		
(8) REGFILE	1	1	4		
(#) Busses					
(6) BUS					

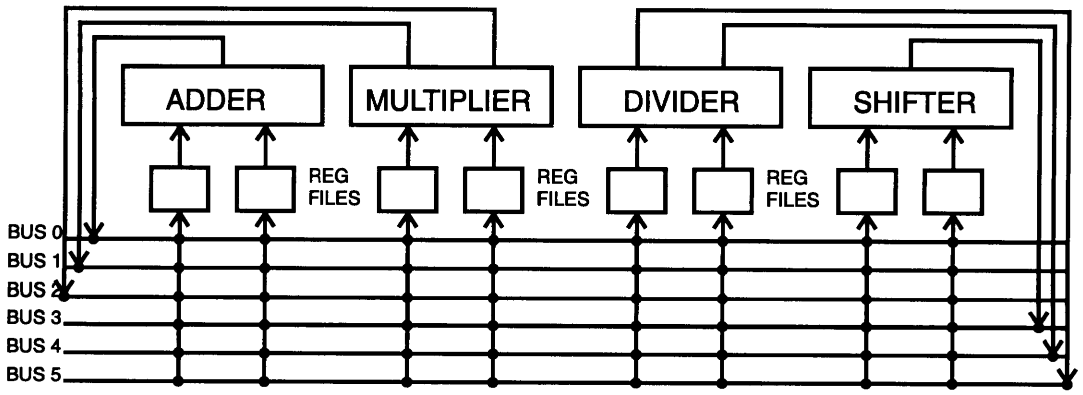


Figure 2-6: Distributed register file VLIW processor.

(#) Functional Units	**# ins**	**# outs**	**latency**	**pipe?**	**ops**
(1) ADDER	2	1	1	YES	ADD, SUB
(1) MULTIPLIER	2	1	1	YES	MUL
(1) DIVIDER	2	2	5	NO	DIV
(1) SHIFTER	2	1	1	YES	SHFT
(#) Register Files	**# ins**	**# outs**	**# regs**		
(8) REGFILE	1	1	4		
(#) Busses					
(5) BUS					

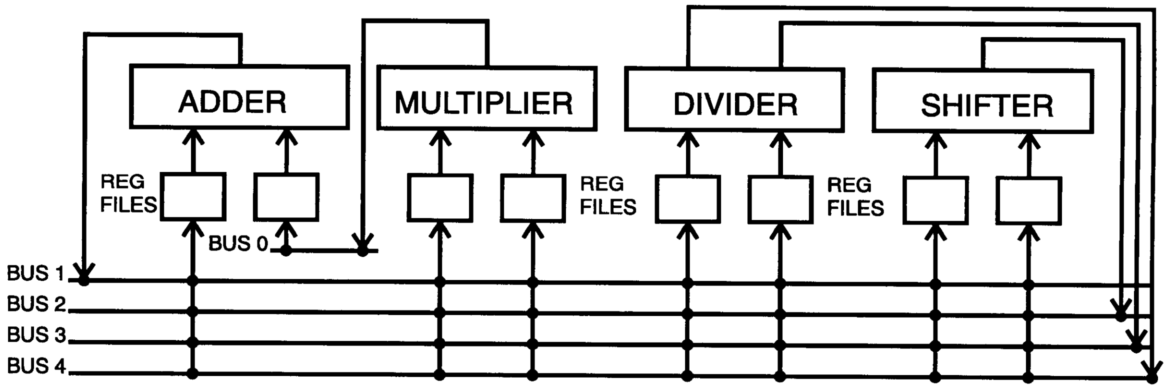


Figure 2-7: Communication–constrained (multiply–add) VLIW processor.

2.3 Summary

This chapter describes the basic structure of the scheduler test system. The scheduler test system produces instruction schedules for a class of processors. It takes two inputs from the user: a program to schedule, and a machine on which to schedule it. Schedule generation is divided into three phases: **parse**, **analysis**, and **schedule**. The **parse** phase converts a program into assembly instructions, the **analysis** phase processes the assembly instructions to produce a program graph, and the **schedule** phase uses the program graph to produce a schedule for a particular machine.

Input programs are written in a simple C-like language called μasm. μasm is a stream–oriented language that borrows some syntax from C. It also has support for special features of the underlying hardware, such as zero–overhead loops and conditional select operations.

Machines are described in terms of basic components that are connected together. There are three types of components: functional units, register files, and busses. Functional units compute results that are stored in register files, and busses route data between functional units and register files. Although restrictive, these simple components are sufficient to describe a wide variety of machines.

Chapter 3

Program Graph Representation

It is common to use a graph representation, such as a directed acyclic graph (DAG), to represent programs during compilation [10, 1]. During the **analysis** phase, the scheduler test system produces an internal graph representation of a program called a *program graph.* A program graph is effectively a DAG with some additions for representing the simple control flow of μasm.

Several factors motivated the design of the program graph as an internal program representation. First, an acceptable representation must expose much of the parallelism in a program. The scheduler targets highly parallel machines, and effective instruction scheduling must exploit all available parallelism.

Second, a representation must allow for simple code motion across basic blocks. Previous researchers have demonstrated that scheduling across basic blocks can be highly effective for VLIW style machines [8, 13]. In this case, since μasm has no conditionally executed code, the representation need only handle the special case of code motion into and out of loops.

Finally, a representation must be easily modifiable for use in the simulated annealing algorithm. As described fully in Chapter 4, the simulated annealing instruction scheduling algorithm dynamically modifies the program graph to search for efficient schedules.

The basic program graph, described in Section 3.1, represents the structure of a program and is independent of the machine on which the program is scheduled. When used in the simulated annealing instruction scheduling algorithm, the program graph is labeled with annotations that record scheduling information. These annotations are specific to the target machine class and are described in Section 3.2.

3.1 Basic Program Graph

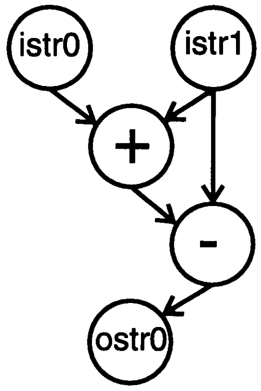
The basic program graph is best introduced by way of example. Figures 3-1a and 3-1b show a simple *μasm* program and the assembly instruction sequence produced by the **parse** phase of the scheduler test system. Because the program has no loops, the program graph for this program is simply a DAG, depicted in Figure 3-1c. The nodes in the DAG represent assembly instructions in the program, and the edges designate data dependencies between operations.

```
int a,b;
a = istream(0,int);
b = istream(1,int);
a = a + b;
ostream(0,int) = a - b;
```

(a)

```
istream R0, #0
istream R1, #1
iadd32  R0, R0, R1
isub32  R2, R0, R1
ostream R2, #0
```

(b)



(c)

Figure 3-1: Example loop-free *μasm* program (a), its assembly listing (b), and its program graph (c).

3.1.1 Code Motion

DAGs impose a partial order on the instructions (nodes) in the program (program graph). An ordering of the nodes that respects the partial order is called a *valid order* of the nodes, and instructions are allowed to "move" relative to one another as long as a valid order is maintained. Generally, there are many different valid orders for instructions in a program, as shown in Figure 3-2. However, there is always at least one valid order, the *program order*, which is the order in which the instructions appear in the original assembly program.

In Chapter 4 it is shown how the scheduler utilizes code motion within the program graph constraints to form instruction schedules.

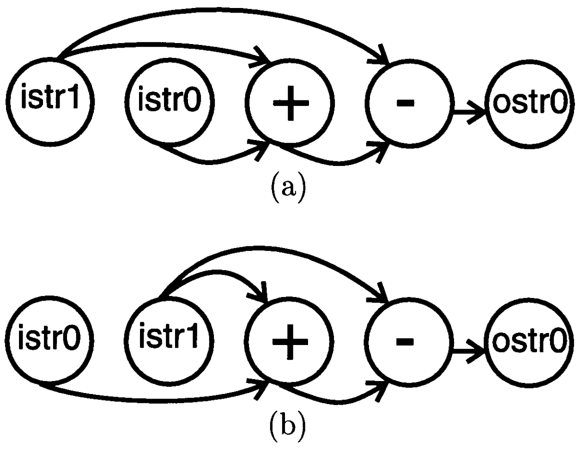


Figure 3-2: Two different valid orderings of the example DAG.

3.1.2 Program Graph Construction

Constructing a DAG for programs with no loops is straightforward. First, nodes are created for each instruction in the program, and then directed edges are added where data dependencies exist. Table–based algorithms are commonly used for adding these directed edges [18]. A simple table–based algorithm for adding edges to an existing list of nodes is given in Figure 3-3. The table records the nodes that have created the most recent values for variables in the program.

The simple DAG construction algorithm can be modified to produce program graphs for programs with loops. The program in Figure 3-4 has one loop, and the program graph construction process is illustrated in Figure 3-5. First, nodes are created for each instruction in the program, including loop instructions. Second, the nodes are scanned in program order using a table to add forward–directed data dependency edges. Third, the nodes within the loop body are scanned a second

```
build-dag(L)
  for each node N in list L do
    I = instruction associated with node N
    for each source operand S of instruction I do
      M = TABLE[S]
      add edge from node M to node N
    for each destination operand D of instruction I do
      TABLE[D] = N
```

Figure 3-3: Table–based DAG construction algorithm.

time with the *same* table to add backward–directed data dependency edges (back edges). Program graphs use dependency cycles to represent looping control flow. Finally, special loop dependency edges are added to help enforce code motion rules for instructions around loops. These special loop dependency edges and the code motion rules are explained in Section 3.1.3.

```
int a,b;
a = istream(0,int);
loop count = 0,99
{
  b = istream(1,int);
  a = a + b;
  ostream(0,int) = a - b;
}
```

(a)

```
istream R0, #0
loop    #100
istream R1, #1
iadd32  R0, R0, R1
isub32  R2, R0, R1
ostream R2, #0
endloop
```

(b)

Figure 3-4: Example *μasm* program with loops (a) and its assembly listing (b).

The construction process outlined above can be generalized to programs with arbitrary numbers of nested loops. In general, each loop body within a program must be scanned twice. Intuitively, the first scan determines the initial values for variables within the loop body, and the second scan introduces back edges for variables redefined during loop iteration. An algorithm for constructing program graphs (without loop dependency edges) is presented in Figure 3-6.

Clearly, program graphs are not DAGs; cycles appear in the program graph where

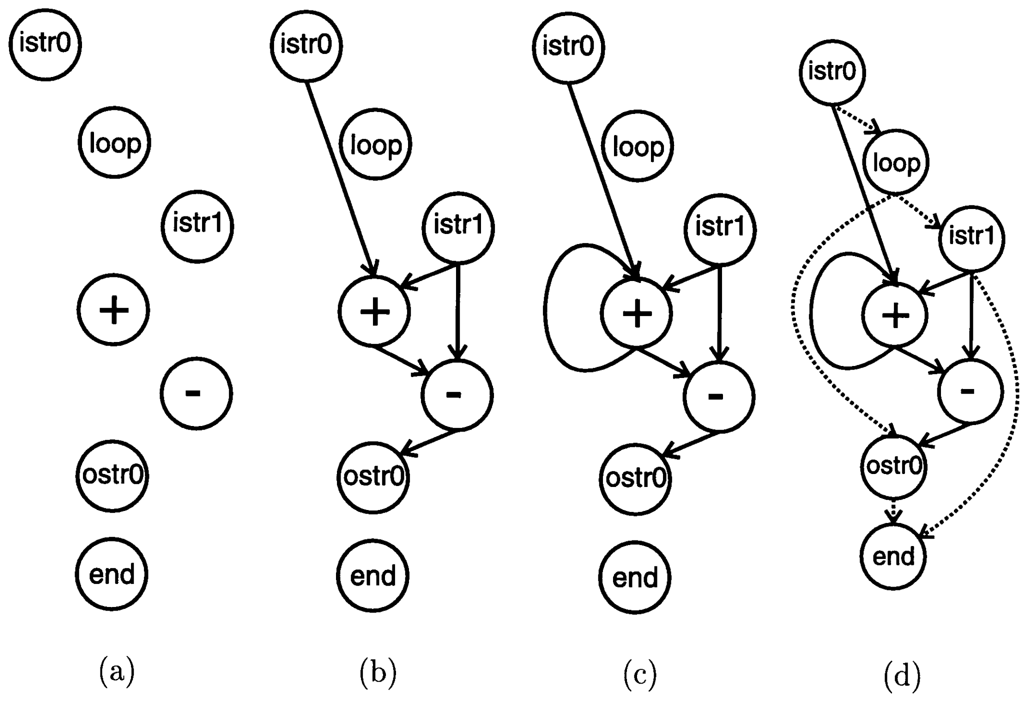


Figure 3-5: Program graph construction process: nodes (a), forward edges (b), back edges (c), loop dependency edges (d).

data dependencies exist between loop iterations. However, a program graph can be treated much like a DAG if back edges are never allowed to become forward edges in any ordering of the nodes. When restricted in this manner, back edges effectively become special forward edges that are simply marked as backward. In all further discussions, back edges are considered so restricted.

3.1.3 Loop Analysis

Program graphs are further distinguished from DAGs by special loop nodes which mark the boundaries of loop bodies. These nodes govern how instructions may move into or out of loop bodies.

An instruction can only be considered inside or outside of a loop with respect to some valid ordering of the program graph nodes. If, in some ordering, a node in the program graph follows a loop start node and precedes the corresponding loop end node, then the instruction represented by that node is considered to be inside

```
build-program-graph(L)
  for each node N in list L do
    I = instruction associated with node N
    if I is not a loop end instruction
      for each source operand S of instruction I do
        M = TABLE[S]
        add edge from node M to node N
      for each destination operand D of instruction I do
        TABLE[D] = N
    else
      L2 = list of nodes in loop body of I, excluding I
      build-dag(L2)
```

Figure 3-6: Program graph construction algorithms.

that loop. Otherwise, it is considered outside the loop. A node's *natural loop* is the innermost loop that it occupies when the nodes are arranged in program order.

Compilers commonly move code out of loop bodies as a code optimization [1]. Fewer instructions inside a loop body generally result in faster execution of the loop. In the case of wide instruction word machines, code motion into loop bodies may also make sense [8]. Independent code outside of loop bodies can safely occupy unused instruction slots within a loop, making the overall program more compact.

However, not all code can safely be moved into or out of a loop body without changing the outcome of the program. The program graph utilizes a combination of static and dynamic analyses to determine safe code motions.

Static Loop Analysis

Static loop analysis determines two properties of instructions with respect to all loops in a program: *loop inclusion* and *loop exclusion.* If an instruction is *included* in a loop, then that instruction can never move out of that loop. If an instruction is *excluded* from a loop, then that instruction can never move into that loop. If it is neither, then that instruction is free to move into or out of that loop.

A program graph represents static loop inclusion and exclusion with pairs of loop dependency edges. Loop inclusion edges behave exactly like data dependency edges, forcing an instruction to always follow the loop start instruction *and* to always precede the loop end instruction. Loop exclusion edges are interpreted slightly differently. They require an instruction to always follow a loop end instruction *or* to always precede a loop start instruction. Figure 3-7 demonstrates loop dependency edges.

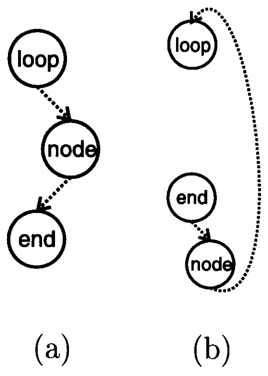


Figure 3-7: Loop inclusion (a) and loop exclusion (b) dependency edges.

Static loop analysis uses the following simple rules to determine loop inclusion and loop exclusion for nodes in a program graph:

1. If a node has side effects, then it is *included* in its natural loop and *excluded* from all other loops contained within its natural loop.

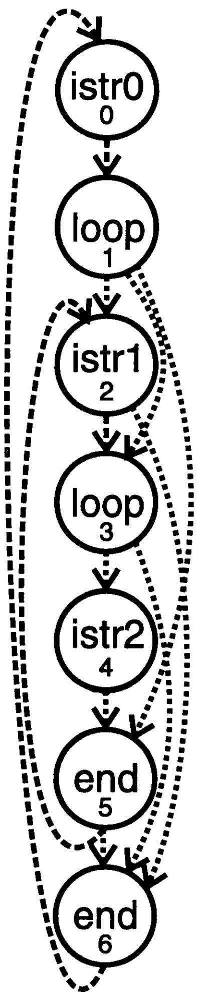
2. If a node references (reads or writes) a back edge created by a loop, then it is *included* in that loop.

The first rule ensures that instructions that cause side effects in the machine, such as loop start, loop end, istream, or ostream instructions, are executed exactly the number of times intended by the programmer. Figure 3-8 depicts a simple situation in which this rule is used to insert loop inclusion and loop exclusion edges into a program graph. The program has multiple istream instructions that are contained within two nest loops. As a result of static loop analysis, the first istream instruction (node 0) is excluded from the outermost loop (and, consequently, all loops contained within it). The second istream instruction (node 2) is included in the outermost loop and excluded from the innermost loop, while the third istream instruction (node 4) is simply included in the innermost loop.

```
int a;

a = istream(0,int);
loop count = 0,99
{
  a = istream(1,int);
  loop count2 = 0,99
  {
    a = istream(2,int);
  }
}
```

```
0 istream R0, #0
1 loop    #100
2 istream R0, #1
3 loop    #100
4 istream R0, #2
5 end
6 end
```



(a) (b) (c)

Figure 3-8: Static loop analysis (rule 1 only) example program (a), labeled assembly listing (b), and labeled program graph (c).

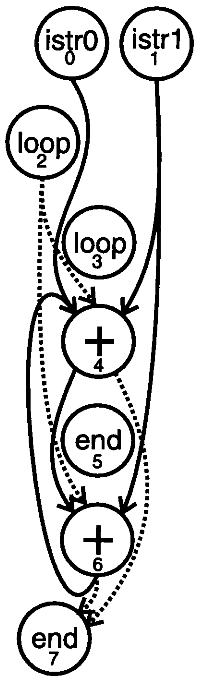
The second rule forces instructions that read or write variables updated inside a loop to also remain inside that loop. Figure 3-9 shows a simple situation in which this rule is enforced. The program contains two `iadd32` instructions, which are connected by a back edge created by the outermost loop. Thus, both nodes are included in this loop. Note that the first add instruction (node 4) is not included in its natural loop (the innermost loop). Inspection of the program reveals that moving node 4 from its natural loop does not change the outcome of the program.

These two rules are not sufficient to prevent all unsafe code motions with regard to loops. It is possible to statically restrict all illegal code motions, but at the expense

```
int a,b,c;

a = istream(0,int);
b = istream(1,int);
loop count = 0,99
{
  loop count2 = 0,99
  {
    c = a + b;
  }
  a = c + b;
}
```

```
0 istream R0, #0
1 istream R1, #1
2 loop    #100
3 loop    #100
4 iadd32  R2, R0, R1
5 end
6 iad32   R0, R2, R1
7 end
```



(a) (b) (c)

Figure 3-9: Static loop analysis (rule 2 only) example program (a), labeled assembly listing (b), and labeled program graph (c).

of some legal ones. However, dynamic loop analysis offers a less restrictive way to disallow illegal code motions, but at a runtime penalty.

Dynamic Loop Analysis

Some code motion decisions can be better made dynamically. For example, consider the program and associated program graph in Figures 3-10a and 3-10b. As a result of static loop analysis, nodes 3 and 7 are included in the outer loop but are free to move into the inner loop. Inspection of the program graph reveals that either node 3 or node 7 can safely be moved into the inner loop, but not both. Although the inner loop is actually independent from the outer loop, moving both nodes into the inner loop causes the outer loop computation to be repeated too many times. Such problems can occur whenever a complete dependency cycle is moved from one loop to another.

Dynamic loop analysis seeks to prevent complete cycles in the program graph

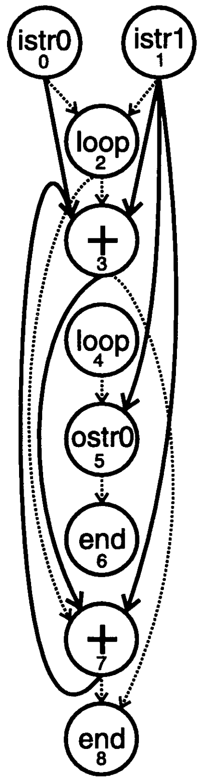
```
int a,b,c;

a = istream(0,int);
b = istream(1,int);
loop count1 = 0,99
{
  c = a + b;
  loop count2 = 0, 99
  {
  ostream(0,int) = b;
  }
  a = c + b;
}
```

(a)

```
0 istream R0, #0
1 istream R1, #1
2 loop    #100
3 iadd32  R2, R0, R1
4 loop    #100
5 ostream R1, #0
6 end
7 iadd32  R0, R2, R1
8 end
```

(b)



(c)

Figure 3-10: Dynamic loop analysis example program (a), labeled assembly listing (b), and labeled program graph (c).

from changing loops as a result of code motion. Checks are dynamically performed before each potential change to the program graph ordering. Violations of the cycle constraint are disallowed.

Central to dynamic loop analysis is the notion of the *innermost shared loop* of a set of nodes. The innermost shared loop of a set of nodes is the innermost loop in the program that contains all the nodes in the set. There is always one such loop for any subset of program graph nodes; it is assumed that the entire program itself is a special "outermost" loop, and all nodes share at least this one loop.

When moving a node on a computation cycle, dynamic loop analysis ensures that the innermost shared loop for all nodes on the cycle is the same as that when the nodes are arranged in program order. Otherwise, the move is not allowed.

3.2 Annotated Program Graph

Often, a DAG (or some other data structure) is used to guide the code generation process during compilation [1]. In addition, for complex machines, a separate scoreboard structure may be used to centrally record resource usage. However, to facilitate dynamic modification of the schedule, it is often useful to embed scheduling information in the graph structure itself. Embedding such information in a basic program graph results in an *annotated program graph.*

Scheduling information is recorded as annotations to the nodes and edges of the basic program graph. These annotations are directly related to the type of hardware on which the program is to be scheduled. For the class of machines described in Section 2.2, *node annotations* record information about functional unit usage, and *edge annotations* record information about communication between functional units.

3.2.1 Node Annotations

Annotated program graph nodes contain two annotations: **unit** and **cycle**. The annotations represent the instruction's functional unit and initial execution cycle.

Node annotations lend concreteness to the notion of ordering in the program graph. By considering the **unit** and **cycle** annotations to be two independent dimensions, the program graph can be laid out on a grid in "space–time" (see Figure 3-11). This grid is a useful way to visualize program graphs during the scheduling process.

3.2.2 Edge Annotations

Edges in an annotated program graph represent the flow of data from one functional unit to another. They contain annotations that describe a direct data path through the machine. Listed in the order encountered in the machine, these annotations are **unit-out-port**, **bus**, **reg-in-port**, **register**, **reg-out-port**, and **unit-in-port**. Figure 3-12 illustrates the relationship between edge annotations and the actual path of data through the machine.

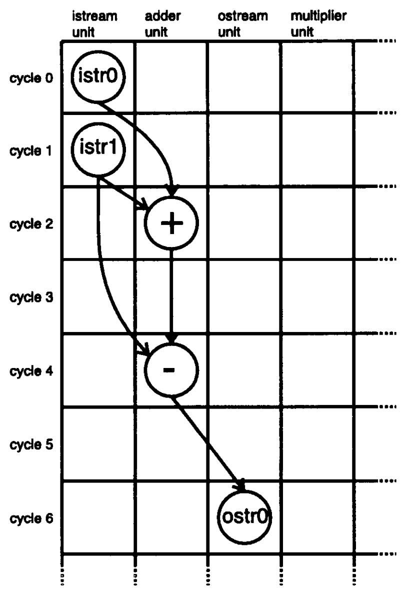


Figure 3-11: Program graph laid out on grid.

Assigning values to the annotations of an edge that connects two annotated nodes is called *routing data.* Two annotated nodes determine a source and destination for a data word. Many paths may exist between the source and destination, so routing data is generally done by systematically searching all possibilities for the first valid path.

Valid paths may not exist if the machine does not have the physical connections, or if the machine resources are already used for other routing. If no valid paths exist for routing data, then the edge is considered *broken.* Broken edges have unassigned annotations.

3.2.3 Annotation Consistency

The data routing procedure raises the topic of annotation consistency. Annotations must be assigned such that they are consistent with one another. For example, an edge cannot be assigned resources that are already in use by a different edge or resources that do not exist in the machine.

Similarly, two nodes generally can not be assigned the same **cycle** and **unit** an-

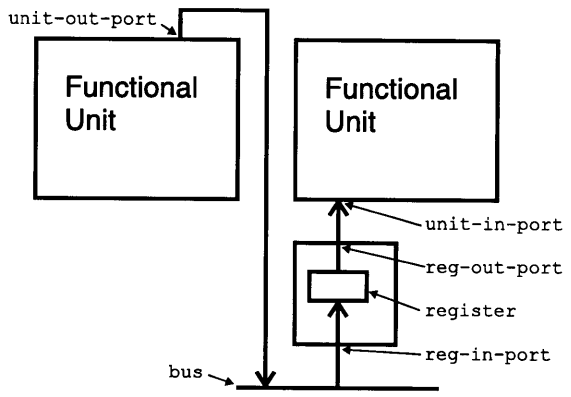


Figure 3-12: Edge annotations related to machine structure.

notations. An exception to this rule occurs when the two nodes are *compatible.* Two nodes are considered compatible if they compute identical outputs. For example, common subexpressions in programs generate compatible program graph nodes. Such nodes would be allowed to share functional unit resources, effectively eliminating the common subexpression.

Additionally, nodes can not be assigned annotations that cause an invalid ordering of the program graph nodes. By convention, only edge annotations are allowed to be unassigned (broken). This restriction implies that data dependency constraints are always satisfied in properly annotated program graphs.

3.3 Summary

This chapter introduces the program graph, a data structure for representing data and simple control flow for programs. The scheduler test system uses the program graph to represent programs for three reasons: (1) it exposes much program parallelism, (2) it allows code motion into and out of loops, and (3) it is easily modifiable.

A program graph consists of nodes and edges. As in a DAG representation, nodes correspond to instructions in the program, and edges correspond to data dependencies between instructions. In addition, special loop nodes and edges represent program control flow.

Program graphs are constructed with a simple table–based algorithm, similar to a table–based DAG construction algorithm. Loop edges are created by a static loop analysis post–processing step. Dynamic loop analysis supplements the static analysis to ensure that modifications to the program graph to not result in incorrect program execution.

An annotated program graph is a program graph that has been augmented for use in a scheduling algorithm. Two types of annotations are used: node annotations and edge annotations. Node annotations record on which cycle and unit an instruction is scheduled, and edge annotations encode data flow paths through the machine.

Chapter 4

Scheduling Algorithm

This chapter describes a new instruction scheduling algorithm based on the simulated annealing algorithm. This algorithm is intended for use on communication–constrained VLIW machines.

4.1 Simulated Annealing

Simulated annealing is a randomized search algorithm used for combinatorial optimization. As its name suggests, the algorithm is modeled on the physical processes behind cooling crystalline materials. The physical structure of slowly cooling (i.e., annealing) material approaches a state of minimum energy despite small random fluctuations in its energy level during the cooling process. Simulated annealing mimics this process to achieve function minimization by allowing a function's value to fluctuate locally while slowly "cooling down" to a globally minimal value.

The pseudocode for an implementation of the simulated annealing algorithm is given in Figure 4-1. This implementation of the algorithm takes T, the current temperature, and α, the temperature reduction factor, as parameters. These parameters, determined empirically, guide the cooling process of the algorithm, as described later in this section.

The simulated annealing algorithm uses three data–dependent functions: **initialize**, **energy**, and **reconfigure**. The **initialize** function provides an initial data point

```
D = initialize()
E = energy(D)
repeat until 'cool'
  repeat until reach 'thermal equilibrium'
    newD = reconfigure(D)
    newE = energy(D)
    if newE < E
      P = 1.0
    else
      P = exp(-(newE - E)/T)
    if(random number in [0,1) < P)
      D = newD
      E = newE
  T = alpha*T
```

Figure 4-1: The simulated annealing algorithm.

from which the algorithm starts its search. The **energy** function assigns an energy level to a particular data point. The simulated annealing algorithm attempts to find the data point that minimizes the **energy** function. The **reconfigure** function randomly transforms a data point into a new data point. The algorithm uses the **reconfigure** function to randomly search the space of possible data points. These three functions, and their definitions for instruction scheduling, are detailed further in Section 4.2.

4.1.1 Algorithm Overview

The simulated annealing algorithm begins by calculating an initial data point and initial energy using **initialize** and **energy**, respectively. Then, it generates a sequence of data points starting with the initial point by calling **reconfigure**. If the energy of a new data point is less than the energy of the current data point, the new data point is accepted unconditionally. If the energy of a new data point is greater than the energy of the current data point, the new data point is conditionally accepted with some probability that is governed by the following equation:

$$p(accept) = e^{-\frac{\Delta E}{T}}, \tag{4.1}$$

where T is the current "temperature" of the algorithm, and ΔE is the magnitude of the energy change between the current data point and the new one. If a new data point is accepted, it becomes the basis for future iterations; otherwise the old data point is retained.

This iterative process is repeated at the same temperature level until "thermal equilibrium" has been reached. Thermal equilibrium occurs when continual energy decreases in the data become offset by random energy increases. Thermal equilibrium can be detected in many ways, ranging from a simple count of data reconfigurations to a complex trend detection scheme. In this thesis, exponential and window averages are commonly used to detect when the energy level at a certain temperature has reached steady–state.

Upon reaching thermal equilibrium, the temperature must be lowered for further optimization. Lower temperatures allow fewer random energy increases, reducing the average energy level. In this implementation, the temperature parameter T is reduced by a constant multiplicative factor α, typically between 0.85 and 0.99.

Temperature decreases continue until the temperature has become sufficiently "cool," usually around temperature zero. Near this temperature, the probability of accepting an energy increase approaches zero, and the algorithm no longer accepts random increases in the energy level. The algorithm terminates when it appears that no further energy decreases can be found.

It is interesting to note that the inner loop of the algorithm is similar to a simple "hill–climbing" search algorithm. In the hill–climbing algorithm, new data points are accepted only if they are better than previous data points. The simulated annealing algorithm relaxes this requirement by accepting less–fit data points with an exponentially decreasing probability. This relaxation permits the algorithms to avoid getting trapped in local minima. As the temperature decreases, the behavior of the simulated annealing algorithm approaches that of the hill–climbing search.

4.2 Simulated Annealing and Instruction Scheduling

Application of the simulated annealing algorithm to any problem requires definition of the three data–dependent functions **initialize**, **energy**, and **reconfigure** as well as selection of the initial parameters T and α. The function definitions and initial parameters for the problem of optimal instruction scheduling are provided in the following sections.

4.2.1 Preliminary Definitions

A *data point* for the simulated annealing instruction scheduler is a *schedule.* A schedule is a consistent assignment of annotations to each node and edge in an annotated program graph. Schedules may be valid or invalid. A *valid schedule* is a schedule in which the annotation assignment satisfies all dependencies implied by the program graph, respects the functional unit resource restrictions of the target hardware, and allows all data to be routed (i.e., there are no broken edges). The definition of annotation consistency in Section 3.2.3 implies that a schedule can only be invalid if its program graph contains broken edges.

4.2.2 Initial Parameters

The initial parameters T and α govern the cooling process of the simulated annealing algorithm. A proper rate of cooling is crucial to the success of the algorithm, so good choices for these parameters are important.

The initial temperature T is a notoriously data–dependent parameter [14]. Consequently, it is often selected automatically via an initial data–probing process. The data–probing algorithm used in this thesis is shown in Figure 4-2. It is controlled by an auxiliary parameter P, the initial acceptance probability. The parameter P is intended to approximate the probability with which an average energy increase will be initially accepted by the simulated annealing algorithm. Typically, P is set very close to one to allow sufficient probability of energy increases early in the simulated annealing process.

The data probing algorithm reconfigures the initial data point a number of times and accumulates the average change in energy ΔE_{avg}. Inverting Equation (4.1) yields the corresponding initial temperature:

$$T_{initial} = \frac{-\Delta E_{avg}}{\ln P}. \tag{4.2}$$

```
probe-initial-temperature(D,P)
  E = energy(D)
  total = 0
  repeat 100 times
    D2 = reconfigure(D)
    E2 = energy(D2)
    deltaE = abs(E - E2)
    total = total + deltaE
  avgDeltaE = total / 100
  T = -avgDeltaE / ln(P)
  return T
```

Figure 4-2: Initial temperature calculation via data–probing.

The initial parameter α is generally less data–dependent than T. In this thesis, values for α are determined empirically by trial–and–error. The results of these experiments are discussed later in Chapter 5.

4.2.3 Initialize

The **initialize** function generates an initial data point for the simulated annealing algorithm. In the domain of optimal instruction scheduling, the **initialize** function takes a program graph as input and produces an annotation assignment for that program graph (i.e., it creates a schedule).

```
cycle = 0
for each node N in program graph P do
    N->cycle = cycle
    N->unit  = random unit
    cycle = cycle + N->unit->latency + 1
for each edge E in program graph P do
  if data can be routed for edge E
    assign edge annotations to E
  else
    mark E broken
```

Figure 4-3: Maximally–bad initialization algorithm.

The goal of the **initialize** function is to *quickly* produce a schedule. The schedules need not be near–optimal or even valid. One obvious approach is to use a fast, sub–optimal scheduling algorithm, such as a list scheduler, to generate the initial schedule. This approach is easy if the alternate scheduling algorithm is available, but may have the unwanted effect of biasing the simulated annealing algorithm toward schedules close to the initial one. Initializing the simulated annealing algorithm with a data point deep inside a local minimum can cause the algorithm to become stuck near that data point if the initial temperature is not high enough.

Another approach is to construct a "maximally bad" (within reasonable limits) schedule. Such a schedule lies outside all local minima and allows the simulated annealing algorithm to discover randomly which minima to investigate. Maximally bad schedules can be quickly generated using the algorithm shown in Figure 4-3. This algorithm traverses a program graph in program order and assigns a unique start cycle and a random unit to each node in the program graph. A second traversal assigns edge annotations, if possible.

4.2.4 Energy

The **energy** function evaluates the optimality of a schedule. It takes a schedule as input and outputs a positive real number. Smaller energy values are assigned to more desirable schedules. Energy evaluations can be based on any number of schedule properties including critical path length, schedule density, data throughput, or hardware resource usage. Penalties can be assigned to undesirable schedule features such as broken edges or unused functional units. Some example energy functions are described in the following paragraphs.

Largest–start–time

The **largest–start–time** energy function is shown in Figure 4-4. The algorithm simply computes the largest start cycle of all operations in the program graph. Optimizing this energy function results in schedules that use a minimum number of VLIW instructions, often resulting in fast execution. However, this function is not well suited to the simulated annealing algorithm, as it is very flat and exhibits infrequent, abrupt changes in magnitude. In general, flat functions provide no sense of "progress" to the simulated annealing algorithm, resulting in a largely undirected, random search.

```
lst = 0
for each node N in program graph P
  if N->cycle > lst
    lst = N->cycle
return lst
```

Figure 4-4: **Largest–start–time** energy function.

Sum-of-start-times

The **sum–of–start–times** energy function appears in Figure 4-5. Slightly more sophisticated than **largest–start–time**, this algorithm attempts to measure schedule length while remaining sensitive to small changes in the schedule. Since all nodes contribute to the energy calculation (rather than just one as in **largest–start–time**), the function output reflects even small changes in the input schedule, making it more suitable for use in the simulated annealing algorithm.

```
m = 0
for each node N in program graph P
  m = m + N->cycle
return m
```

Figure 4-5: **Sum–of–start–times** energy function.

Sum-of-start-times (with penalty)

Figure 4-6 shows the **sum–of–start–times** energy function with a penalty applied for broken program graph edges. Assessing penalties for undesirable schedule features causes the simulated annealing algorithm to reject those schedules with high probability. In this case, the simulated annealing algorithm would not likely accept schedules with broken edges (i.e., invalid schedules).

```
m = 0
for each node N in program graph P
  m = m + N->cycle
brokenedgecount = 0
for each edge E in program graph P
  if E is broken
    brokenedgecount = brokenedgecount + 1
return m * (1 + brokenedgecount*brokenedgepenalty)
```

Figure 4-6: **Sum–of–start–times** (with penalty) energy function.

4.2.5 Reconfigure

The **reconfigure** function generates a new schedule by slightly transforming an existing schedule. There are many possible schedule transformations, the choice of which affect the performance of the simulated annealing algorithm.

In this thesis, good reconfigure functions for simulated annealing possess two required properties:

reversibility The simulated annealing algorithm should be able to undo any reconfigurations that it applies during the course of optimization.

completeness The simulated annealing algorithm should be able to generate any data point from any other data point with a finite number of reconfigurations.

The reconfiguration functions used in this thesis are based on a small set of primitive schedule transformations that together satisfy the above conditions. Those primitives and the reconfiguration algorithms based on them are described in detail in the next sections.

4.3 Schedule Transformation Primitives

All reconfiguration functions used in this thesis are implemented as a composition of three primitive schedule transformation functions: **move–node**, **add–pass–node**, and **remove–pass–node**. Conceptually, these functions act only on nodes in an annotated program graph. In practice, they explicitly modify the annotations of a single node in the program graph, and in doing so may implicitly modify the annotations of any number of edges. Annotation consistency is always maintained.

4.3.1 Move–node

The **move–node** function moves (i.e., reannotates) a node from a source cycle and unit to a destination cycle and unit, if the move is possible. The program graph is left unchanged if the move is not possible. A move is considered possible if it does not violate any data or loop dependencies and if the destination is not already occupied by an incompatible operation. The **move–node** function attempts to reroute all data along affected program graph edges. If data rerouting is not possible, the affected edges become broken. Pseudocode for and an illustration of **move–node** appear in Figure 4-7.

4.3.2 Add–pass–node

The **add–pass–node** function adds a new data movement node along with a new data edge to a source node in a program graph. The new node is initially assigned node annotations identical to the source node, as they are considered compatible. Pseudocode for and an illustration of **add–pass–node** appear in Figure 4-8.

4.3.3 Remove–pass–node

The **remove–pass–node** function removes a data movement node along with its corresponding data edge from the program graph. Pass nodes are only removable if they occupy the same cycle and unit as the node whose output they pass. Pseudocode for and an illustration of **remove–pass–node** appear in Figure 4-9.

```
bool move-node(node, cycle, unit)
  node->cycle = cycle
  node->unit = unit
  if any dependencies violated
    restore old annotations
    return failure
  for each node N located at (cycle, unit)
    if node not compatible with N
      restore old annotations
      return failure
  for each edge E in program graph
    if E affected by move
      add E to set S
  search for edge annotation assignment for set S
  if search successful
    assign new annotations to edges in set S
  else
    mark edges in set S broken
  return success
```

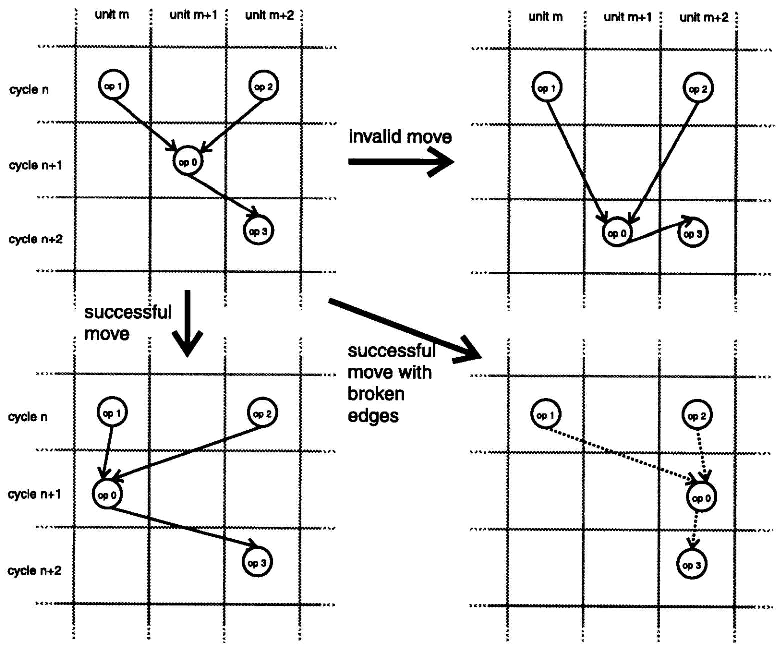


Figure 4-7: The **move–node** schedule transformation primitive.

```
bool add-pass-node(node)
  if pass node already exists here
    return failure
  create new pass node P with input data edge E
  P->cycle = node->cycle
  P->unit = node->unit
  move old output edge from node to P
  attach new edge E to node
  return success
```

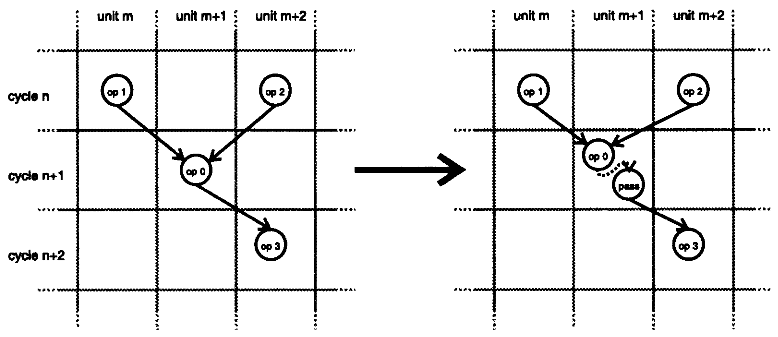


Figure 4-8: The **add–pass–node** schedule transformation primitive.

```
bool remove-pass-node(passnode)
  if passnode is not removable
    return failure
  N = source node of passnode
  move output edge of passnode to N
  remove input edge to passnode
  destroy passnode
  return success
```

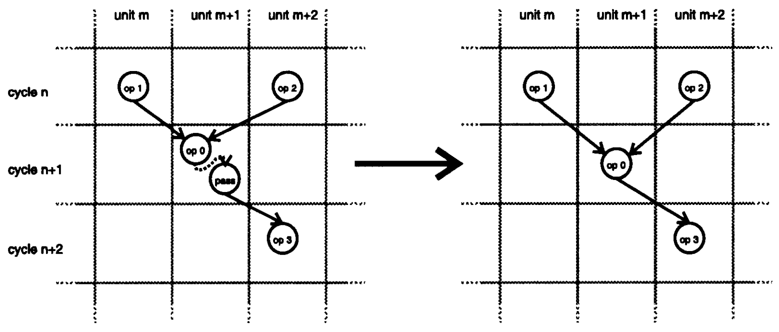


Figure 4-9: The **remove–pass–operation** schedule transformation primitive.

4.4 Schedule Reconfiguration Functions

The schedule transformation primitives described in the previous section can be composed in a variety of ways to generate more complex schedule reconfiguration functions. Three such functions are described in the following sections.

4.4.1 Move–only

The **move–only** reconfiguration function moves one randomly selected node in a program graph to a randomly selected cycle and unit. **Move-only** consists of just one successful application of the **move–node** transformation primitive, as shown in the pseudocode of Figure 4-10.

The **move–only** reconfiguration function satisfies the two requirements of a simulated annealing reconfiguration function only in special cases. The first requirement, reversibility, is clearly always satisfied. The second requirement, completeness, is satisfied only for spaces of schedules with *isomorphic* program graphs. Two program graphs P1 and P2 are considered isomorphic if for every node and edge in P1, there exist corresponding nodes and edges in P2. Further, the corresponding nodes and edges must be connected in an identical fashion. This limited form of completeness can be shown with the following argument.

Consider two schedules S1 and S2 (for the same original program) with isomorphic program graphs P1 and P2. Completeness requires that there exist a sequence of reconfigurations that transform S1 into S2 or, equivalently, P1 into P2. One such sequence can be constructed in two stages. In the first stage, schedule S1 is translated in time by moving each node in P1 from its original cycle C to cycle $C + C_{finalS2}$, where $C_{finalS2}$ is the last cycle used in schedule S2. These moves are applied in reverse program order. In the second stage, each node of the translated program graph P1 is moved to the cycle and unit of its corresponding node in P2. These moves are applied in program order.

Move–only is a useful reconfiguration function for scheduling fully–connected machine configurations. These machines never require additional data movement nodes to generate valid schedules, so the program graph topology need not change during the course of scheduling.

```
move-only(P)
  schedule random node N from program graph P
  repeat
    select random unit U
    select random cycle C
  until move-node(N, C, U) succeeds
```

Figure 4-10: Pseudocode for move–only schedule reconfiguration function.

4.4.2 Aggregate–move–only

While the **move–only** function satisfies (nearly) the two requirements of a good reconfiguration function, it does have a possible drawback. For large schedules, moving a single node is a relatively small change. However, it seems reasonable to assume that the simulated annealing algorithm might accelerate its search if larger changes were made possible by the reconfigure function. The **aggregate–move–only** function is an attempt to provide such variability in the size of the reconfiguration. The pseudocode is shown in Figure 4-11.

Aggregate–move–only applies the **move–only** function a random number of times. The maximum number of applications is controlled by the parameter M, which is a fraction of the total number of nodes in the program graph. For example, at $M = 2$ the maximum number of **move–only** applications is twice the number of program graph nodes. At $M = 0$, **aggregate–move–only** reduces to **move–only**. Defined in this way, **aggregate–move–only** can produce changes to the schedule that vary in magnitude proportional to the schedule size. **Aggregate–move–only** can also produce changes to the schedule that would be unlikely to occur using **move–only**, as it allows chains of **move–node** operations, with potentially large intermediate energy increases, to be accepted unconditionally.

Aggregate–move–only performs identically to **move–only** with respect to the simulated annealing requirements for good reconfigure functions.

```
aggregate-move-only(P,M)
  Y = number of nodes in P
  select random integer X from range [1, M*Y + 1]
  repeat X times
    move-only(P)
```

Figure 4-11: Pseudocode for aggregate–move–only schedule reconfiguration function.

4.4.3 Aggregate–move–and–pass

Enforcing the completeness requirement for non–isomorphic program graphs requires the use of the other two transformation primitives, **add–pass–node** and **remove–pass–node**. These primitives change the topology of a program graph by adding data movement nodes between two existing nodes.

The **aggregate–move–and–pass** function, shown in Figure 4-12, randomly applies one of the two pass–node primitives or the **aggregate–move–only** function. It is controlled by three parameters: the aggregate move parameter M, the probability R of applying a pass node transformation, and the probability S of adding a pass node given that a pass node transformation is applied.

The **aggregate–move–and–pass** function is clearly reversible, and it satisfies a stronger completeness requirement. It is complete for all schedules that have isomorphic program graphs after removal of all pass nodes, as shown in the following argument.

Consider two schedules S1 and S2 (for the same original program) with program graphs P1 and P2 that are isomorphic after removing all pass nodes. A sequence of reconfigurations to transform P1 into P2 can be constructed in five stages. In the first stage, all pass nodes are removed from P1, possibly resulting in broken edges. In the second stage, schedule S1 is translated in time just as in the argument for **move–only**. In the third stage, each node of the translated program graph P1 is moved to the cycle and unit of its corresponding node in P2. In the fourth stage, a pass node is added to the proper node in P1 for each pass node in P2. In the final stage, these newly added pass nodes are moved to the cycles and units of their corresponding pass nodes in P2.

```
aggregate-move-and-pass(P,M,R,S)
  if random number in [0,1) >= R
    aggregate-move-only(P,M)
  else
    if random number in [0,1) < S
      select random node N in P
      add-pass-node(N)
    else
      select random pass node N in P
      remove-pass-node(N)
```

Figure 4-12: Pseudocode for aggregate–move–and–pass schedule reconfiguration function.

4.5 Summary

This chapter describes the simulated annealing algorithm in general and its specific application to the problem of optimal instruction scheduling.

The simulated annealing algorithm is presented along with the three problem–dependent functions **initialize**, **energy**, and **reconfigure** that are required to implement it.

Straightforward implementations of **initialize** and **energy** for the problem of optimal instruction scheduling are given. Versions of **reconfigure** based on the three schedule transformation primitives **move–node**, **add–pass–node**, and **remove–pass–node** are proposed. The reversibility and completeness properties of these functions are discussed.

Chapter 5

Experimental Results

In theory, the simulated annealing instruction scheduling algorithm outlined in the previous chapter is able to find optimal instruction schedules given enough time. In practice, success within a reasonable amount of time depends heavily upon good choices for the algorithm's various parameters. Good choices for these parameters, in turn, often depend on the inputs to the algorithm, making the problem of parameter selection a vexing one. This chapter presents the results of parameter studies designed to find acceptable values for these parameters.

5.1 Summary of Results

The experiments in this chapter investigate five parameters: the initial acceptance probability P, the temperature reduction factor α, the aggregate move fraction M, the pass node transformation probability R, and the pass node add probability S. These parameters are varied for a selection of input programs and machine configurations to find values that may apply in more general situations.

The initial acceptance probability P and temperature reduction factor α are examined together in an experiment described in Section 5.3. It is found that, given sufficiently high starting temperature, the solution quality and algorithm runtime are directly influenced by the value of α. Values of $P \geq 0.8$ gave sufficiently high starting temperatures, and values of $\alpha \geq 0.95$ gave best final results.

The aggregate move fraction M is considered in the experiment of Section 5.4. It is found that large aggregate moves do not reduce the number of reconfigurations needed to reach a solution or the overall run time of the algorithm. In fact, large reconfigurations may even have a negative effect. Thus, an aggregate move fraction of $M = 0$ is recommended.

The pass node transformation probability R and the pass node add probability S are investigated in Section 5.5. It is found that low values of S (0.1 – 0.3) and mid–range values of R (0.3 – 0.5) provide the best chance of producing valid schedules with no broken edges. However, the parameter R did exhibit some input–dependent behavior. In comparison with hand schedules, no combination of R and S resulted in optimal schedules that made good use of the machine resources.

5.2 Overview of Experiments

In all experiments, the **sum–of–start–times** (with penalty) energy function and the **aggregate–move–and–pass** reconfigure function are used. The invalid edge penalty is set at 100.0.

Experiments are conducted using two source input programs: `paradd8.i` and `paradd16.i`. Both programs are very similar, although `paradd16.i` is approximately twice as large as `paradd8.i`. These programs are chosen to investigate how the parameter settings influence the performance of the simulated annealing algorithm on increasing program sizes. The source code for these programs appears in Appendix C.

The experiments in Sections 5.3 and 5.4 use two fully–connected machine configurations: `small_single_bus.md` and `large_multi_bus.md`. The first machine has four functional units (adder, multiplier, shifter, and divider) and distributed register files connected with a single bus. The second machine has sixteen functional units (four of each from the first machine) and distributed register files connected with a full crossbar bus network. These machines are chosen to see how the parameter settings affect the performance of the algorithm on machines of varying complexity.

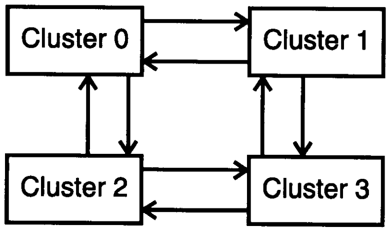


Figure 5-1: Nearest neighbor communication pattern.

The machine description files for these machines appear in Appendix D.

The pass node experiment in Section 5.5 uses two communication–constrained machine configurations: `cluster_with_move.md` and `cluster_without_move.md`.

The first communication–constrained machine has twenty functional units organized into four clusters with five functional units each. Each cluster has an adder, a multiplier, a shifter, a divider, and a data movement (move) unit. Within a cluster, the functional units communicate directly to one another via a crossbar network. Between clusters, units must communicate through move units. Thus, for data to move from one cluster to another, it must first be passed through a move unit, adding a one cycle latency to the operation.

The second communication–constrained machine has sixteen functional units similarly organized into four clusters. Clusters cannot communicate within themselves, but must write their results into other clusters. Thus, data is necessarily transferred from cluster to cluster during the course of computation.

In both communication–constrained machines, clusters are connected in a nearest–neighbor fashion, as depicted in Figure 5-1. Because of the move units, `cluster_with_move.md` is considered more difficult to schedule than `cluster_without_move.md`.

It should be noted that each data point presented in the following sections results from a single run of the algorithm. Due to its randomized nature, the algorithm is expected occasionally to produce anomalous results. Such anomalous results are reflected by outliers and "spikes" in the data. Ideally, each data point should represent an average of many runs of the algorithm with an associated variance, but the algorithm's long runtimes do not permit this much data collection.

5.3 Annealing Experiments

Empirically determining cooling parameters is often done when using the simulated annealing algorithm [14]. In this implementation of the algorithm, the cooling process is controlled by two parameters: the initial acceptance probability P and the temperature reduction factor α. The following experiments attempt to find values for these parameters which yield a minimum energy in a reasonable amount of time. These experiments are carried out only on fully–connected machine configurations, as the parameters needed for communication–constrained machines are yet to be determined. It is hoped that the parameter values found in this experiment carry over to other programs and machine configurations.

The programs `paradd8.i` and `paradd16.i` are tested on machine configurations `small_single_bus.md` and `large_multi_bus.md`. As the temperature probing algorithm is sensitive to the initial state of the algorithm, both list–scheduler and maximally–bad initialization strategies are used, resulting in eight sets of data.

For each set of data, P is varied from 0.05 to 0.99, and α is varied from 0.5 to 0.99. All other parameters (M, R, and S) are set to zero. For each pair of P and α, the minimum energy found and the number of reconfigurations required to find it are recorded.

The results for `paradd8.i` are plotted in Figure 5-2, and those for `paradd16.i` in Figure 5-3. All the raw data from the experiment can be found in Appendix E.

5.3.1 Analysis

The parameter α has perhaps the largest effect on the scheduling outcomes. As shown in the graphs, the number of reconfigurations (and consequently the runtime of the algorithm) exhibits an exponential dependence on the α parameter. In addition, the quality of the scheduling result, as measured in the graphs of minimum energy, is strongly correlated with high α values, which is not unexpected given its effect on runtime. The value of 0.99 gave best results, but at an extreme cost in the number of reconfigurations. A slightly lower value of 0.95 is probably sufficient in most cases.

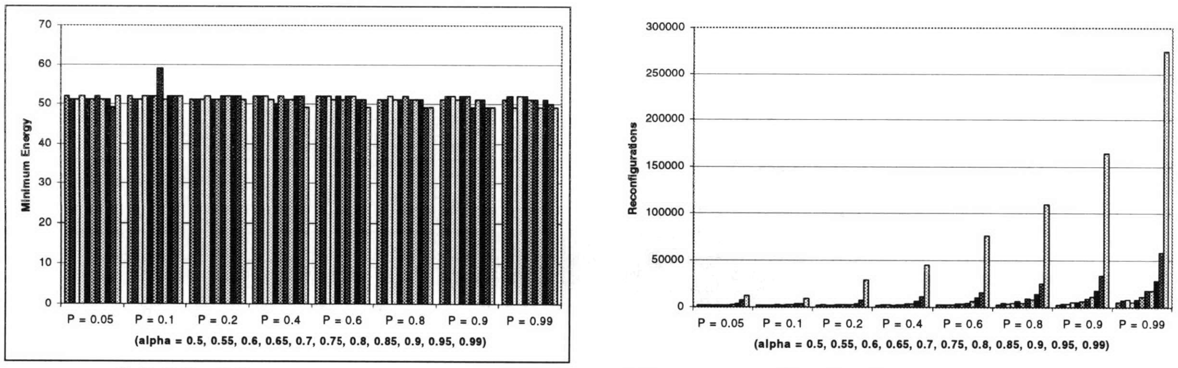
The dependence on parameter P is less dramatic. In the minimum energy graphs that demonstrate some variation in P, it appears that there is some threshold after which P has a positive effect. This threshold corresponds to some sufficient temperature that allows the algorithm enough time to find a good minimum. In most cases, this threshold value occurs at $P = 0.8$ or higher.

The influence of parameters α and P is more clearly illustrated in plots of energy vs. time. Figure 5-4 shows four such plots for the program `paradd16.i` on machine configuration `small_single_bus.md`. In these plots, the "time" axis is labeled with the temperatures at each time, so that the absolute temperature values are evident. In these plots, it seems that P controls the amplitude of the energy oscillation, and α controls the number of reconfigurations (more data points indicate more reconfigurations).

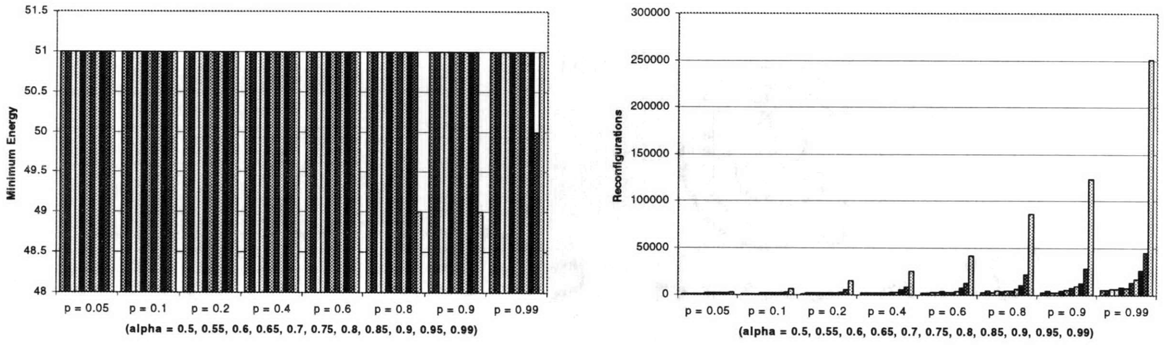
The initialization strategy has little effect on the scheduling outcomes. At some low temperatures, the experiments initialized with the list scheduler seem to get hung up on the initial data point, but this behavior disappears at higher temperatures. This result is in line with expectations; list schedulers perform fine on fully–connected machines like the ones in this experiment.

The difference in machine complexity has the expected result: the smaller machine takes less time to schedule than the more complex one.

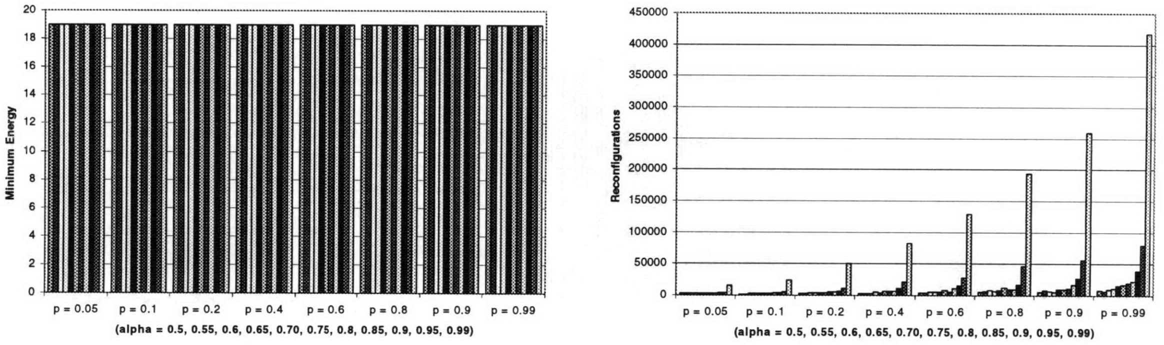
The most surprising result is that the smaller program takes more reconfigurations to schedule than the larger one. This anomaly may be due to the temperature probing procedure used to determine starting temperature. The probing process may have been calculating relatively higher starting temperatures for the smaller program.



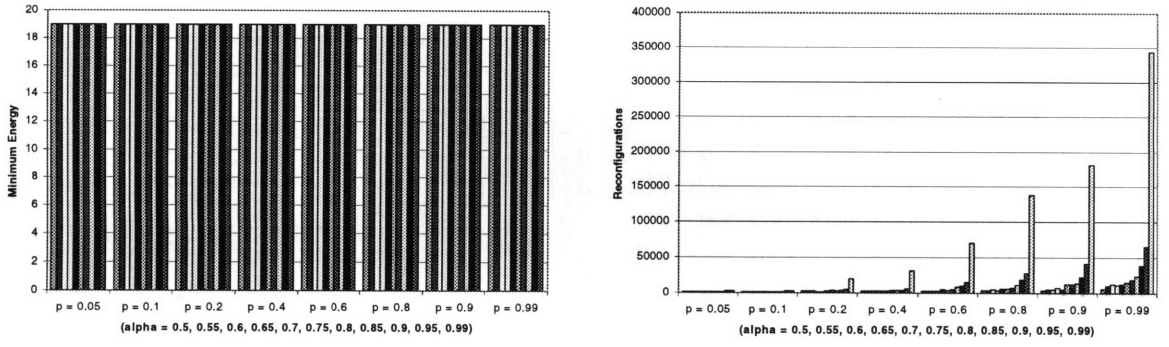
(a) Machine `small_single_bus.md` with maximally–bad initialization.



(b) Machine `small_single_bus.md` with list–scheduler initialization.

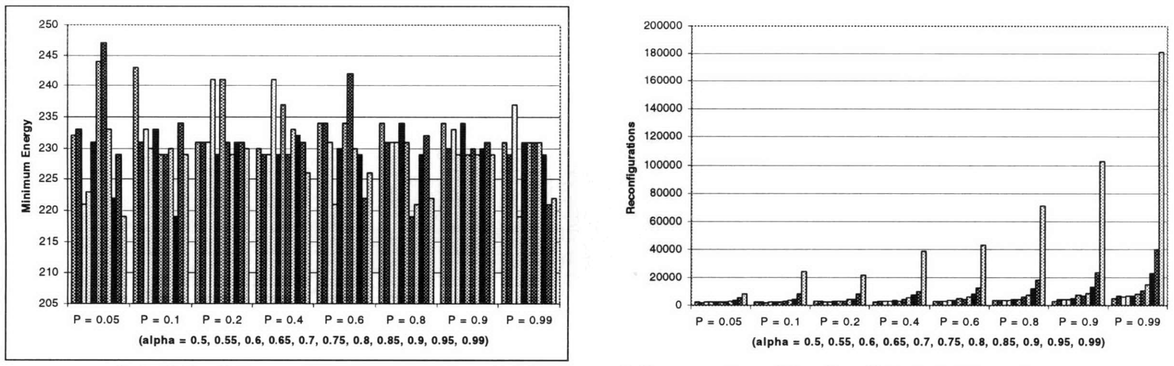


(c) Machine `large_multi_bus.md` with maximally–bad initialization.

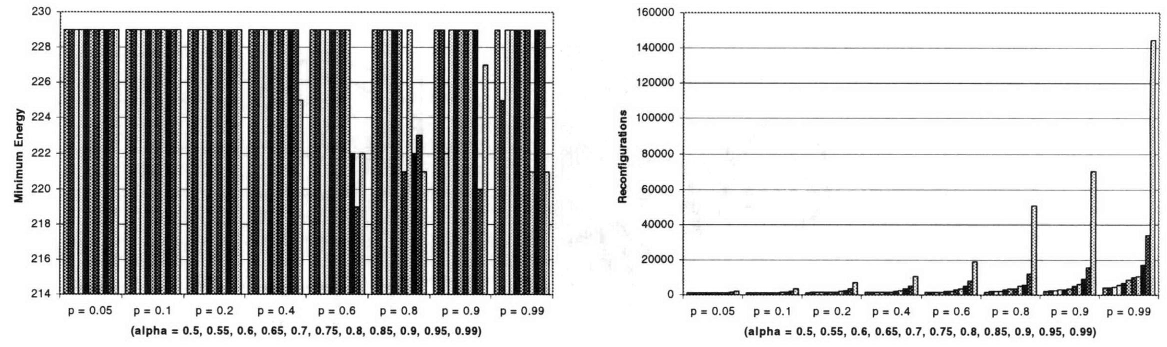


(d) Machine `large_multi_bus.md` with list–scheduler initialization.

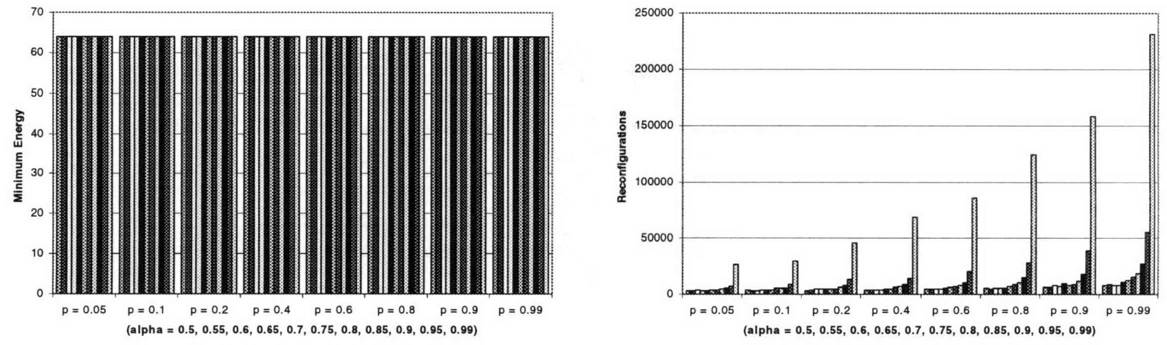
Figure 5-2: Annealing experiments for `paradd8.i`.



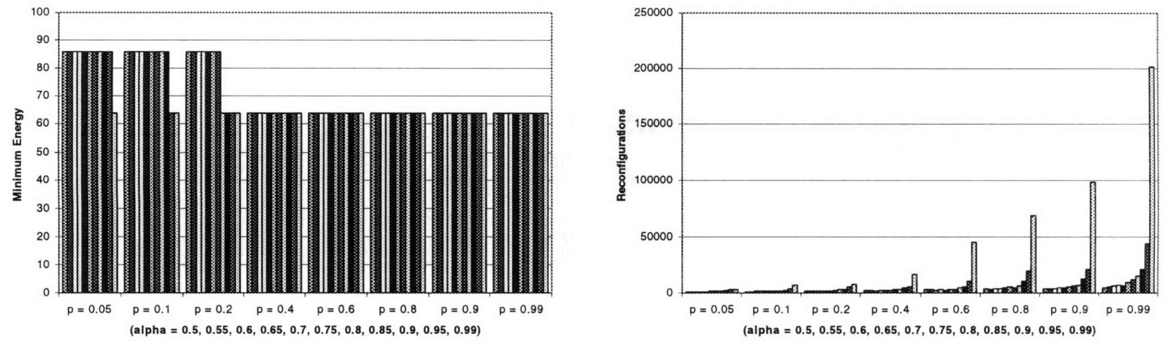
(a) Machine small_single_bus.md with maximally–bad initialization.



(b) Machine small_single_bus.md with list–scheduler initialization.

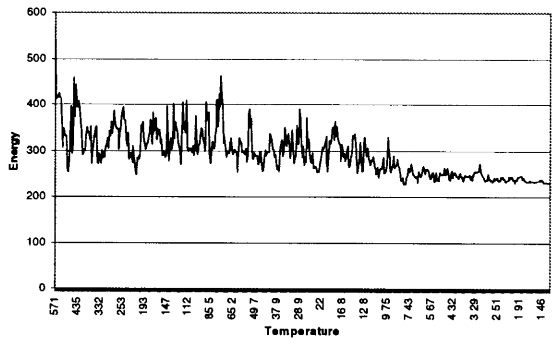


(c) Machine large_multi_bus.md with maximally–bad initialization.

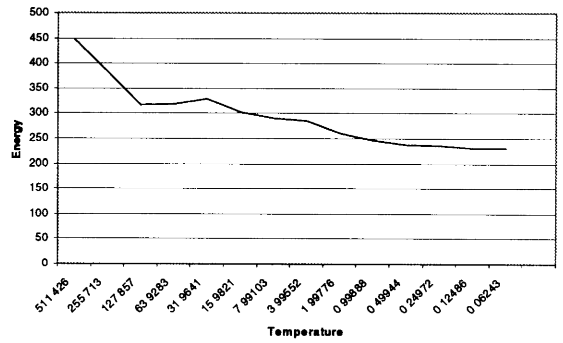


(d) Machine large_multi_bus.md with list–scheduler initialization.

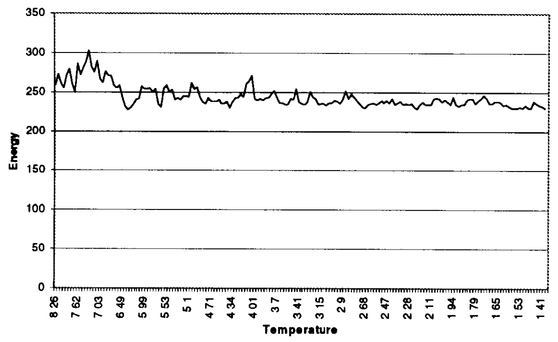
Figure 5-3: Annealing experiments for paradd16.i.



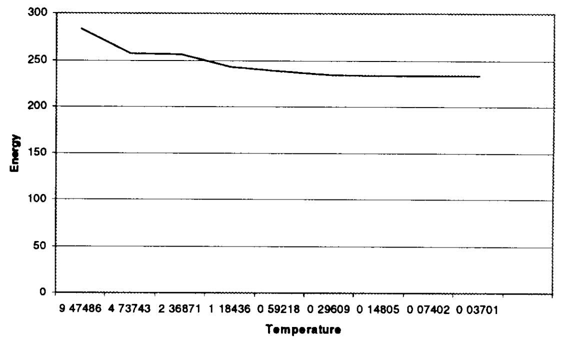
(a) $P = 0.99$, $\alpha = 0.99$



(b) $P = 0.99$, $\alpha = 0.5$



(a) $P = 0.6$, $\alpha = 0.99$



(b) $P = 0.6$, $\alpha = 0.5$

Figure 5-4: Energy vs. time (temperature) for `paradd16.i` on machine `small_single_bus.md`.

5.4 Aggregate Move Experiments

The aggregate–move reconfiguration function is intended to accelerate the simulated annealing search process by allowing larger changes in the data to occur. The size of the aggregate–move is controlled by the aggregate–move fraction M. This experiment attempts to determine a value of M that results in good schedules in a short amount of time.

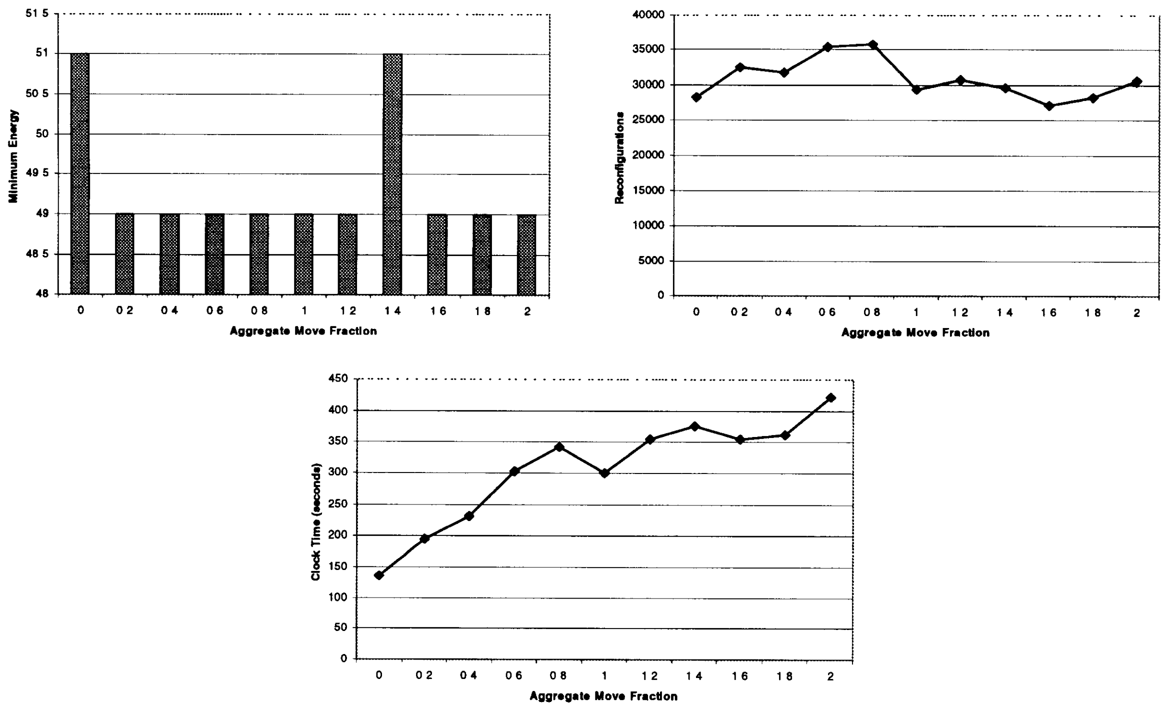
The programs `paradd8.i` and `paradd16.i` are tested on machine configurations `small_single_bus.md` and `large_multi_bus.md`. Only maximally–bad initialization is used, as the results from the Annealing Experiments indicate that list–scheduler initialization does not make much difference for these programs and machine configurations.

For each set of data, M is varied from 0.0 to 2.0. Parameters P and α are set to 0.8 and 0.95, respectively. All other parameters (R and S) are set to zero. For each value of M, the minimum energy found, the number of reconfigurations used to find it, and the clock time are recorded.

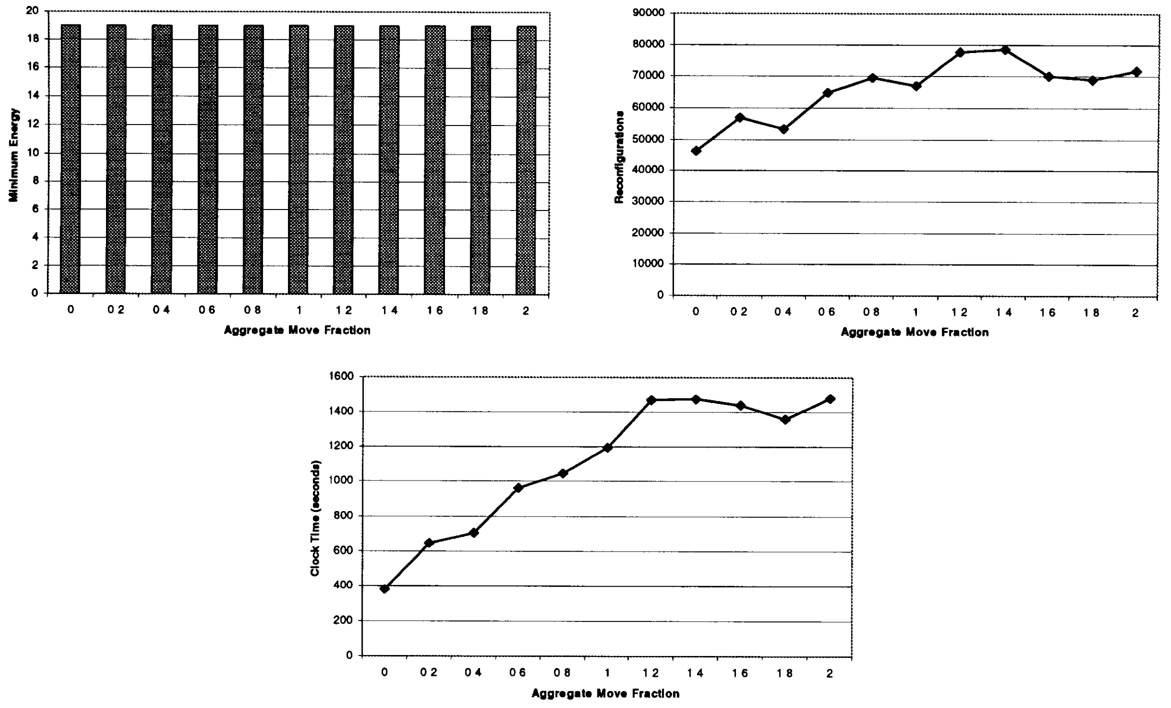
The results for `paradd8.i` are plotted in Figure 5-5, and those for `paradd16.i` in Figure 5-6. All the raw data from the experiment can be found in Appendix E.

5.4.1 Analysis

Variation of the parameter M does not have a significant effect on the minimum energy found by the algorithm. In the only experiment where there is some variation, setting M greater than zero results in worse performance. Increasing M also causes increased runtimes and does not reduce the number of reconfigurations with any regularity, if at all. In general, the aggregate–move reconfiguration function does not achieve its intended goal of accelerating the simulated annealing process. Thus, $M = 0$ (i.e., a single move at a time) seems the only reasonable setting to use.

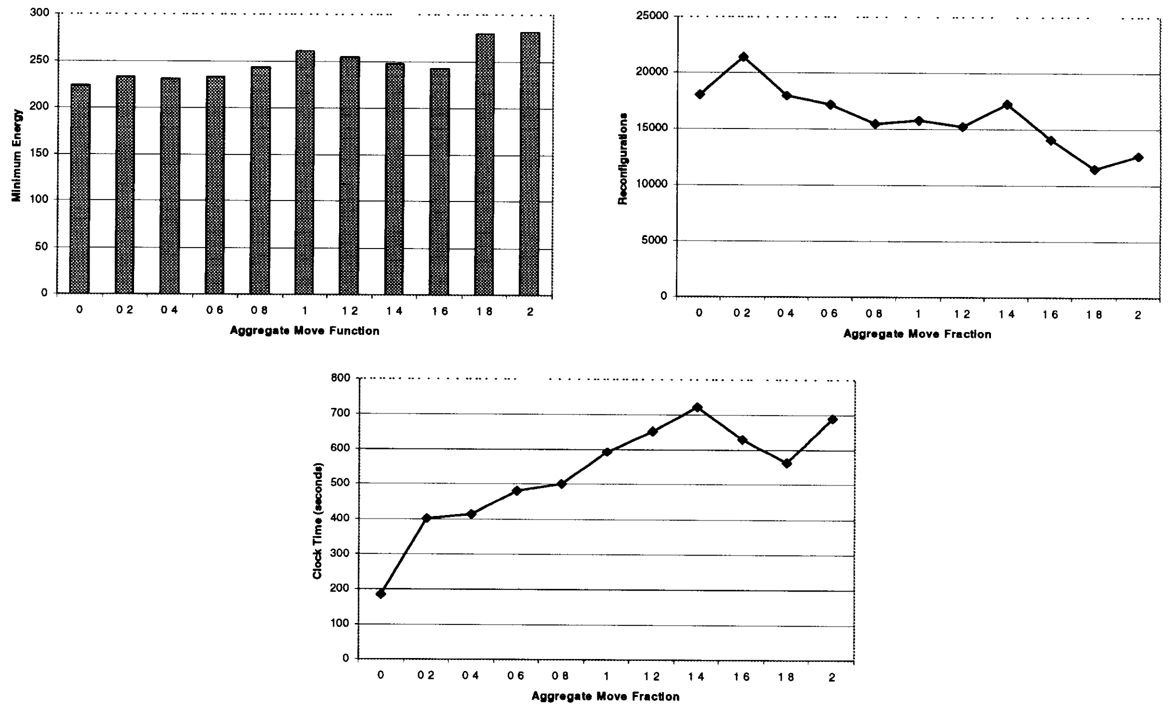


(a) Machine small_single_bus.md.

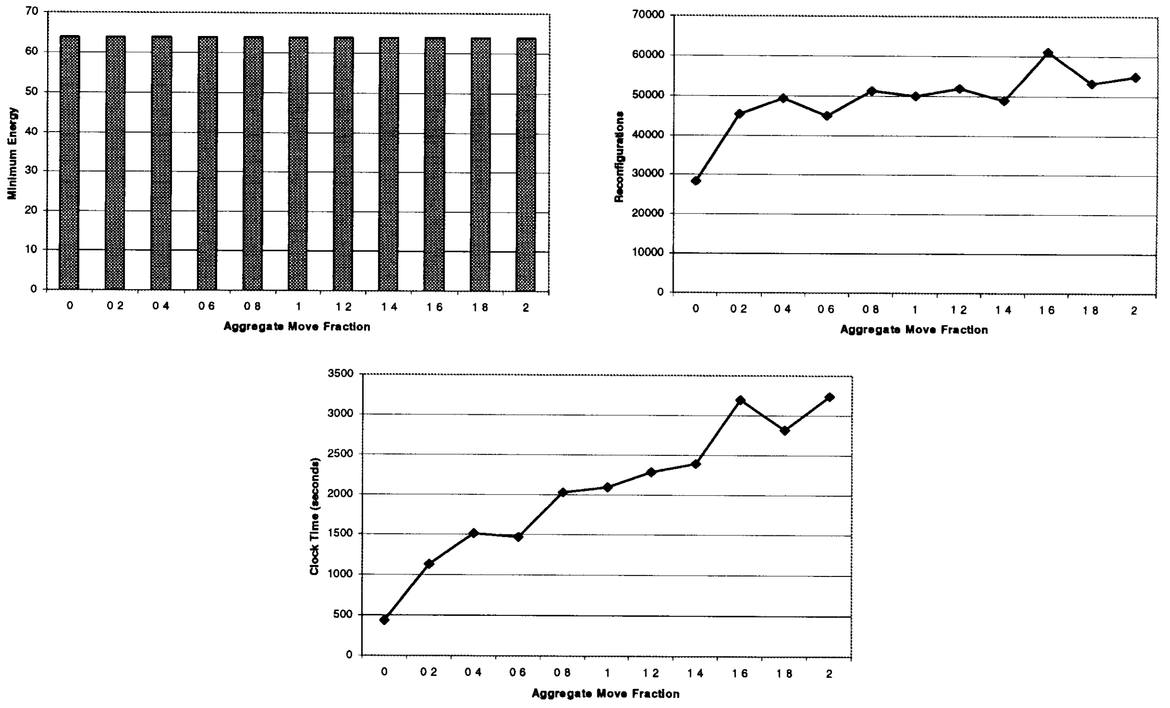


(b) Machine large_multi_bus.md.

Figure 5-5: Aggregate–move experiments for paradd8.i.



(a) Machine `small_single_bus.md`.



(b) Machine `large_multi_bus.md`.

Figure 5-6: Aggregate–move experiments for `paradd16.i`.

5.5 Pass Node Experiments

The add–pass–node and remove–pass–node schedule transformation primitives are key to the success or failure of the simulated annealing instruction scheduling algorithm. In order to create efficient schedules for its intended targets, communication–constrained processors, the algorithm must insert the proper number of pass nodes at the proper locations in the program graph. In doing so, the algorithm must maintain a delicate balance between too many pass nodes and not enough. Insert too many, and the schedule can expand to twice, or even more, its optimal size. Insert too few, and the schedule may become invalid; data is not routed to where it needs to be.

Adding and removing pass nodes is controlled by two parameters, denoted R and S. The parameter R is the probability that the algorithm attempts to add or remove a pass node from the program graph. The parameter S is the probability with which the algorithm adds a pass node given that it has already decided to add or remove one. Thus, the overall probability of adding a pass node is RS, and the overall probability of removing a pass node is $R(1 - S)$. This experiment attempts to find values for R and S which provide the necessary balance to produce efficient schedules.

The programs `paradd8.i` and `paradd16.i` are tested on communication–constrained machine configurations `cluster_with_move.md` and `cluster_without_move.md`. Both maximally–bad and list–scheduler initialization are used.

For each set of data, R and S are varied from 0.1 to 0.9. Parameters P, α, and M are set to 0.8, 0.95, and 0, respectively. For each pair of values, the minimum energy, the actual schedule length, the number of broken edges, and the number of pass nodes is recorded. The clock time is not reported here (see Appendix E), but these experiments took much longer to run than the fully–connected experiments at the same temperature parameters.

The results for `paradd8.i` are plotted in Figure 5-5, and those for `paradd16.i` in Figure 5-6. All the raw data from the experiment can be found in Appendix E.

5.5.1 Analysis

These experiments illustrate the potential problem with using the list scheduler for initialization. The simulated annealing algorithm selects an answer close to the initial data point in all experiments initialized with the list scheduler, as revealed by the absence of broken edges in every experiment (the list scheduler always produces an initial schedule with no broken edges). In some cases, the simulated annealing algorithm is able to improve the list scheduling answer, but such improvements are rare.

The results of the list–scheduler–initialized experiments could indicate that the initial temperature was not set high enough to allow the algorithm to escape from the local minimum created by the list scheduler. This explanation would be valid if the maximally–bad–initialized experiments produce much better answers than the list–scheduler–initialized ones. However, the graphs show that, in almost all cases, the maximally–bad–initialized experiments produce minimum energies that are equivalent or worse than those of the list–scheduler–initialized experiments. Thus, it cannot be determined if the temperature is not set high enough in the list–scheduler–initialized experiments, as the algorithm rarely, if ever, bests the list scheduler's answer.

Lower values of S (0.1–0.3) generally do a better job of eliminating broken edges from the schedule, as evidenced by the graphs of broken edge counts. The graphs also show that, as S increases, the number of pass nodes in the final schedule generally increases along with the minimum energy. After a point, excess pass nodes cause the schedules to become intolerably bad regardless of the number of broken edges. Smaller values of S typically do better on machine `cluster_without_move.md`, which is reasonable as this machine requires fewer pass operations to form efficient hand schedules.

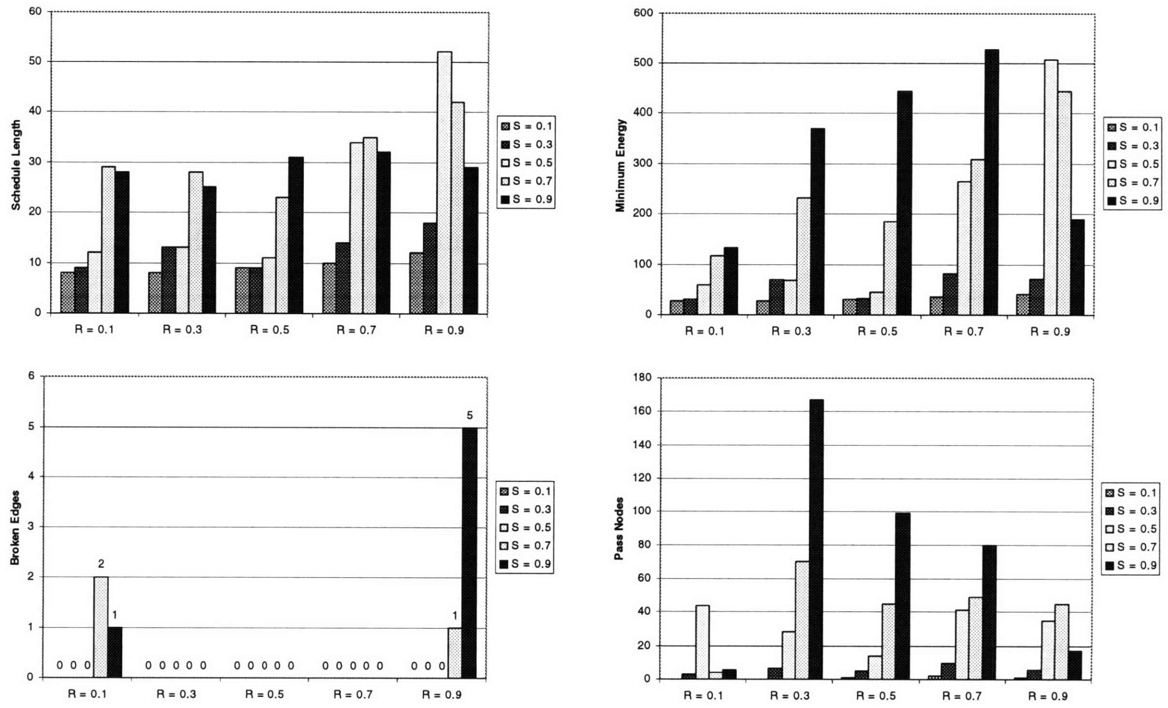
Mid–range values of R (0.3–0.7) result in the fewest broken edges, however its influence on minimum energy and the number of pass nodes is less clear. These measures peak at low values of R for the program `paradd8.i`, but they peak at mid–range values of R for the program `paradd16.i`. These results suggest that R might be more input–dependent than the other parameters.

In general, the algorithm performs better on the `cluster_without_move.md` machine than on the `cluster_with_move.md` machine, as is expected. In some instances, the algorithm finds solutions that are identical to hand–scheduled results for the `cluster_without_move.md` machine. In no case does the algorithm match hand–scheduled results on the `cluster_with_move.md` machine. Most of the automatically generated schedules for this machine utilize only one or two clusters, while efficient hand–scheduled versions make use of all four clusters to reduce schedule length.

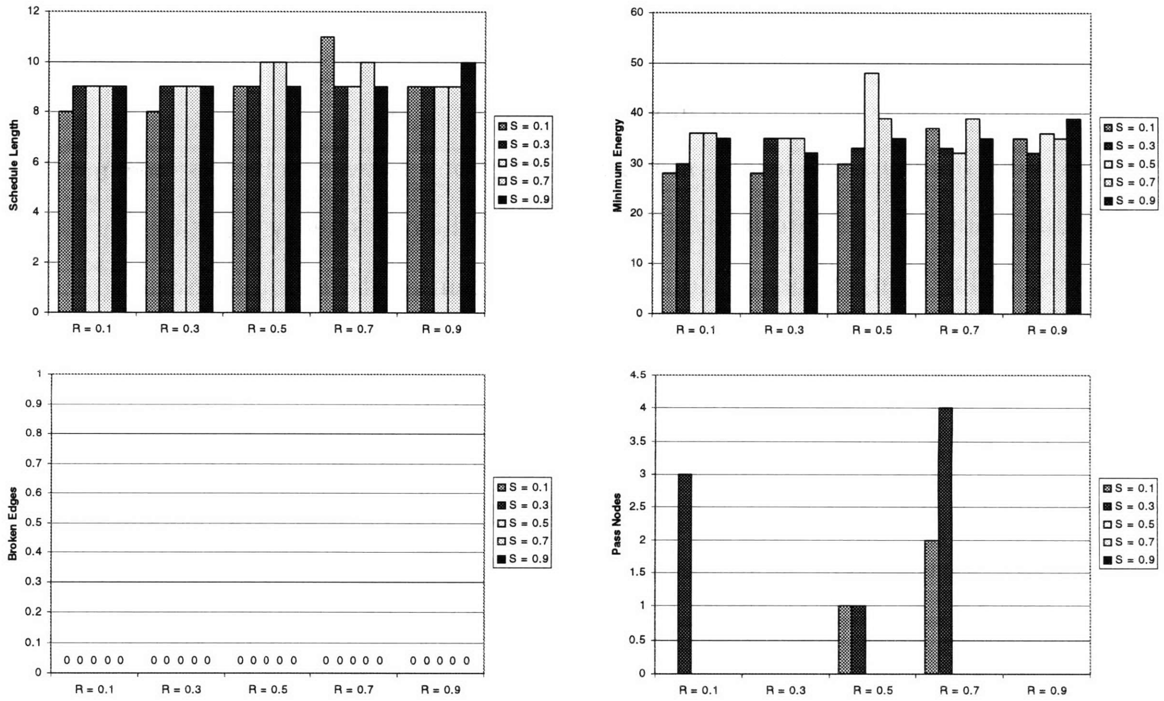
The failure to match hand–scheduled results could be explained by cosidering the ease of transformation from one schedule to another given certain energy and temperature levels. At high temperature levels, moving instructions between clusters, while incurring a large energy penalty, is generally easy to do since high temperatures allow temporary increases in energy level. However, at the high energy levels generally associated with high temperatures, instructions are not compacted optimally, and equivalent energy levels can occur whether instructions are distributed across clusters or not. Thus, at high temperature and energy levels, instructions *can* become distributed across clusters, but have no reason to do so.

At low temperature levels, moving instructions between clusters becomes more difficult. Such moves produce broken edges and large energy penalties, which are rejected at low temperatures. Additionally, low temperatures imply low energy levels, at which instructions are more compacted. When schedules become compact, lowering the energy level further can only be accomplished by distributing instructions across clusters. Thus, at low temperature and energy levels, instructions *cannot* become distributed across, but must do so in order to further optimize the schedule.

In light of the above analysis, truly optimal schedules can only be obtained if the algorithm happens upon the correct cluster distribution at a medium–high temperature and does not (or cannot) change it as the temperature decreases. Such a scenario seems unlikely to happen, as demonstrated by these experiments.

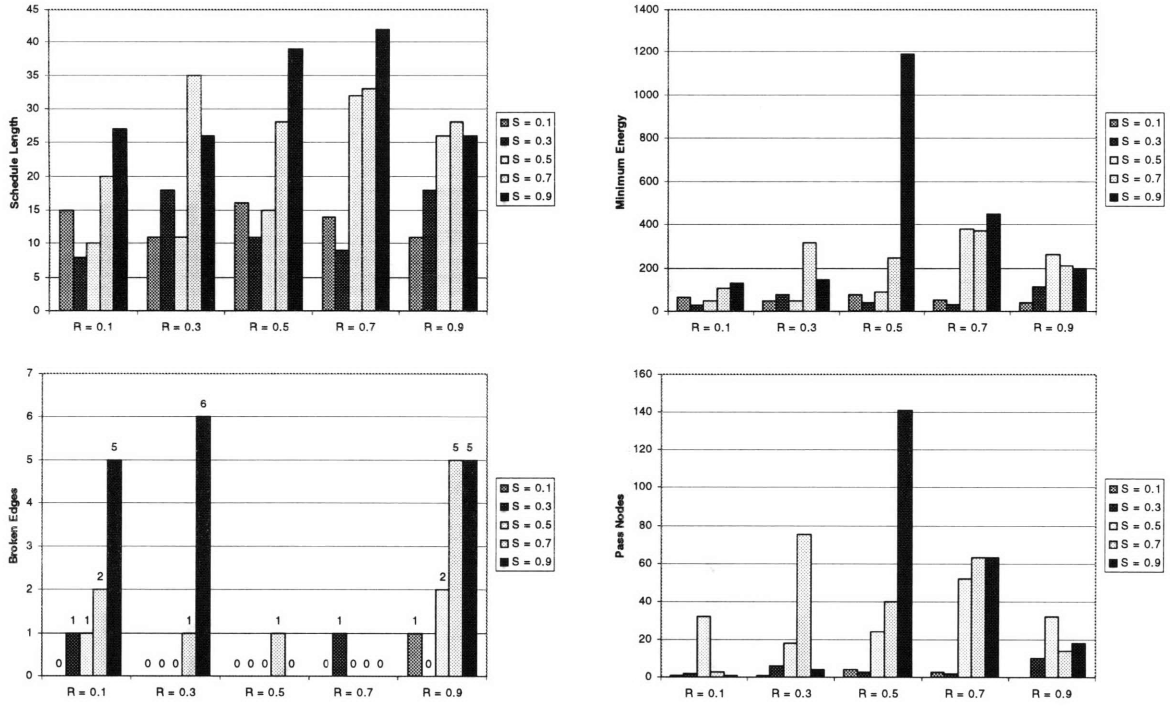


(a) Maximally–bad initialization.

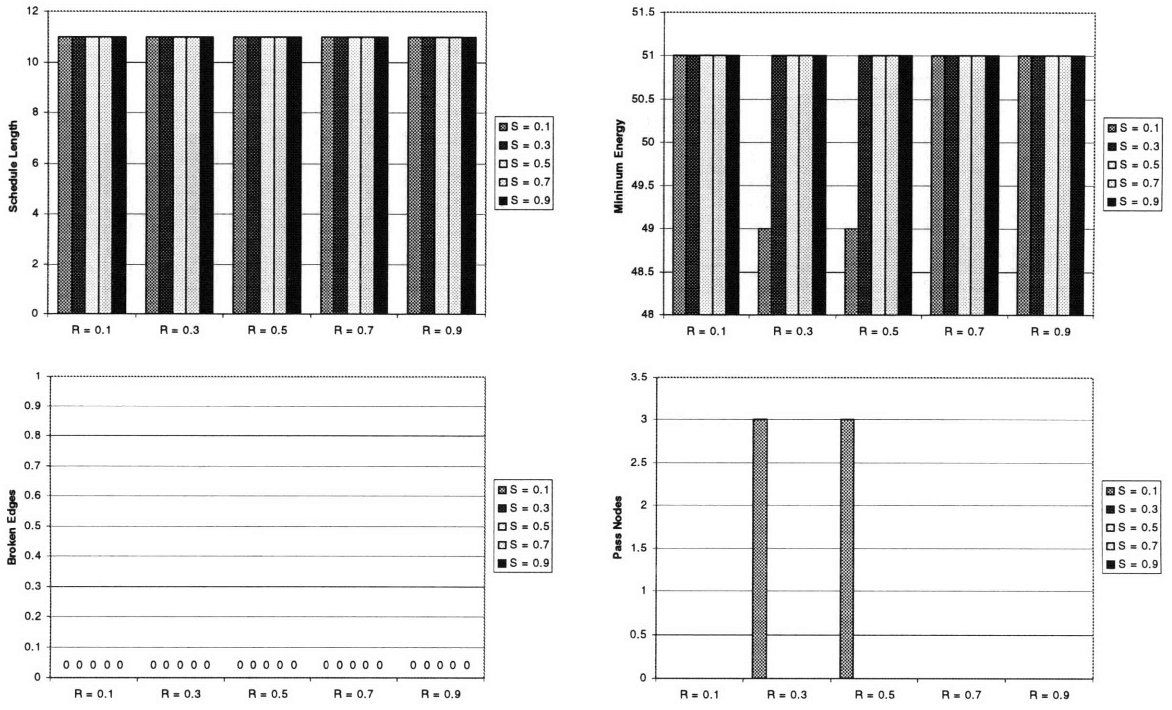


(b) List–scheduler initialization.

Figure 5-7: Pass node experiments for `paradd8.i` on machine `cluster_without_move.md`.

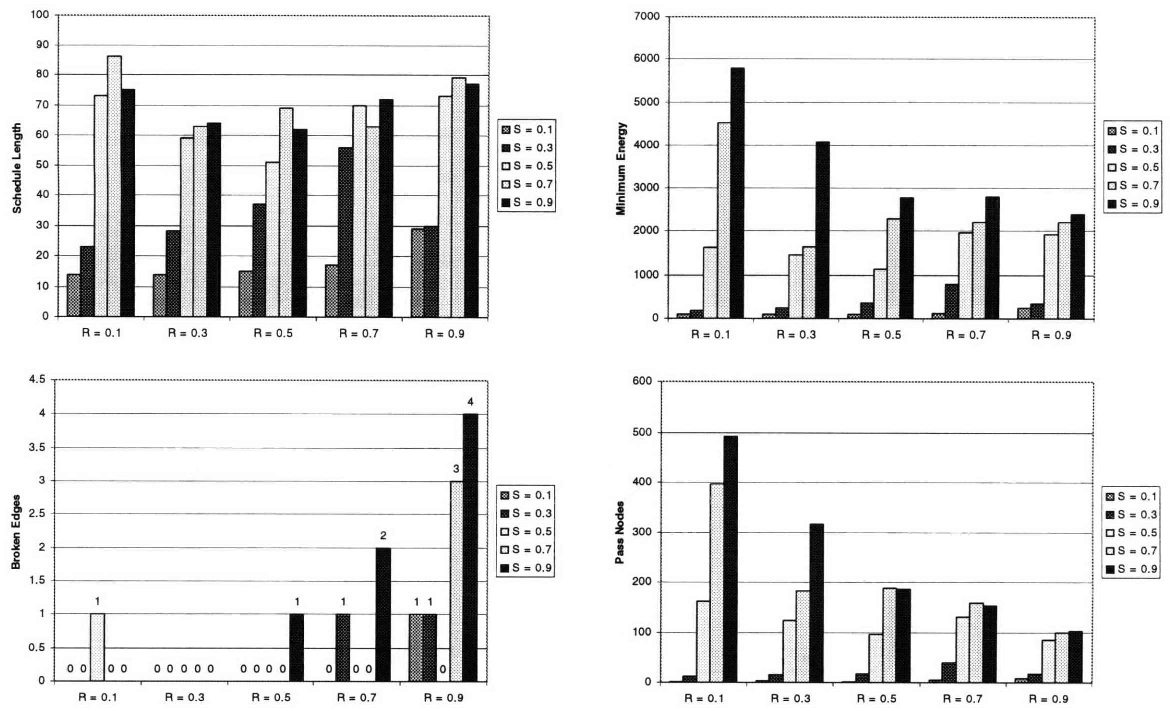


(a) Maximally–bad initialization.

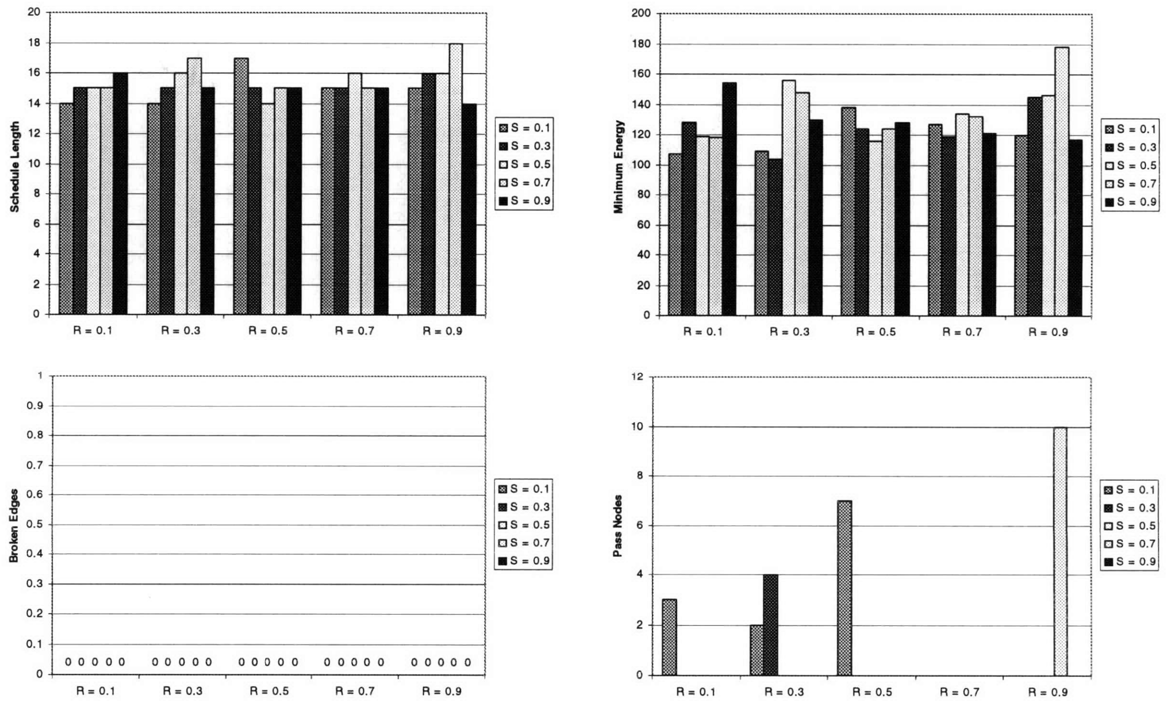


(b) List–scheduler initialization.

Figure 5-8: Pass node experiments for `paradd8.i` on machine `cluster_with_move.md`.

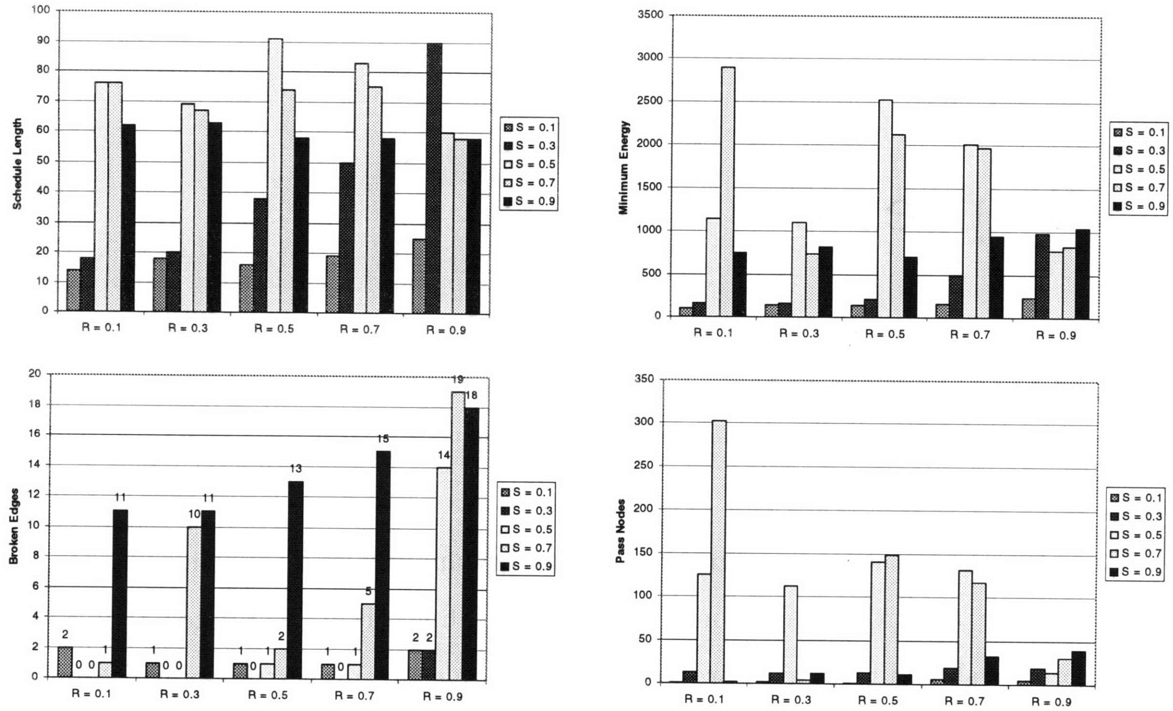


(a) Maximally–bad initialization.

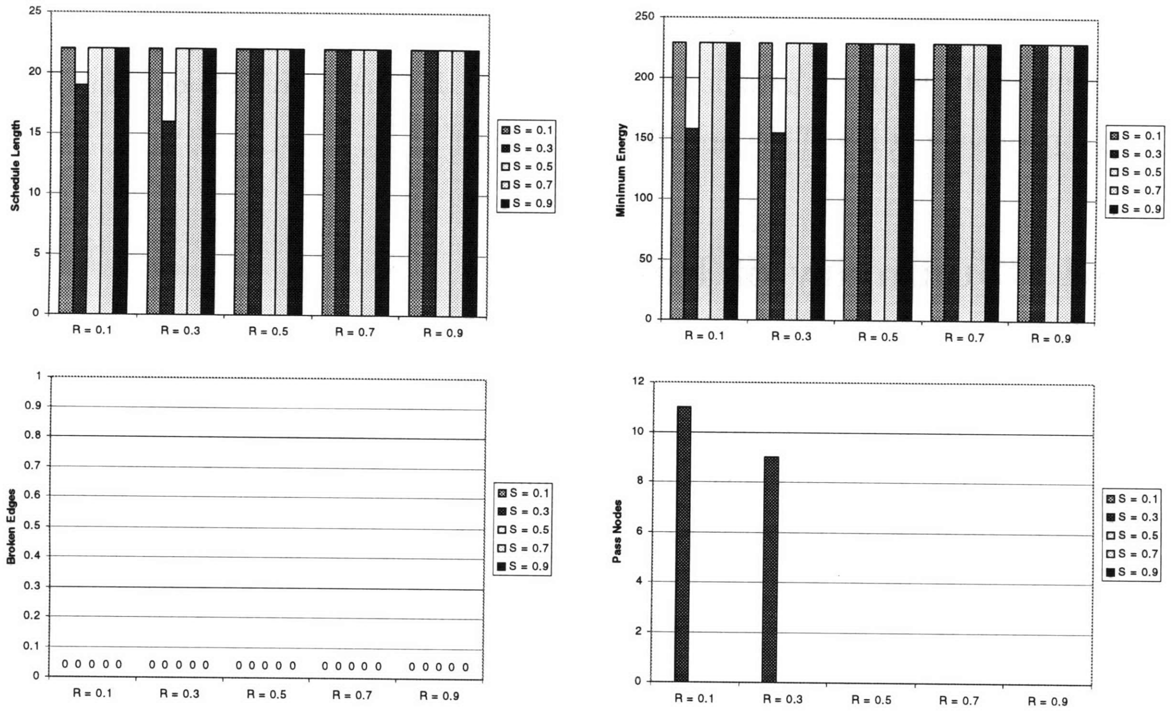


(b) List–scheduler initialization.

Figure 5-9: Pass node experiments for `paradd16.i` on machine `cluster_without_move.md`.



(a) Maximally–bad initialization.



(b) List–scheduler initialization.

Figure 5-10: Pass node experiments for `paradd16.i` on machine `cluster_with_move.md`.

Chapter 6

Conclusion

This thesis presents the design and preliminary analysis of a randomized instruction scheduling algorithm based on simulated annealing. It is postulated that such an algorithm should be able to produce good schedules for processor configurations that are difficult to schedule with traditional scheduling algorithms. This postulate remains unresolved as the algorithm has not been found to perform consistently for any setting of its five main parameters. As a result, this thesis presents only the results of a parameter study of the proposed algorithm.

6.1 Summary of Results

- As expected, the algorithm performs better the longer it is allowed to run. Setting initial acceptance probability $P \geq 0.8$ and temperature reduction factor $\alpha \geq 0.95$ generally allow the algorithm enough time to find optimal schedules for fully–connected machines.

- The algorithm tends to run longer for more complex, larger machine configurations.

- The algorithm tends to run longer for *smaller* programs. This anomaly is probably an artifact of the data probing procedure used to determine an initial temperature for the simulated annealing algorithm.

- The aggregate move parameter M has only negative effects on scheduling efficiency, both in terms of algorithm runtime and schedule quality. Disabling the aggregate move function ($M = 0$) gave best results.

- There are good ranges for the pass node add/remove probability R (0.3–0.7) and the pass node add probability S (0.1–0.3) that result in very few or no broken edges in schedules for communication–constrained machines. These ranges are fairly consistent across programs and machines, but not perfect.

- There are no consistent values of R and S that yield a good pass node "balance." The numbers of pass nodes in the schedules tend to increase with S, but vary widely with R for different programs and machines.

- The algorithm occasionally produced schedules for `cluster_without_move.md` that matched the performance of hand–scheduled code. The algorithm never matched the hand schedules for `cluster_with_move.md`.

6.2 Conclusions

- The algorithm *can* work. The schedules produced for the "easy" communication–constrained machine matched the hand–scheduled versions for good settings of R and S. These schedules often beat the list scheduler, which made poorer schedules for the communication–constrained machines.

- The pass node parameters are very data–dependent. In these experiments, they tended to depend more on the hardware configuration than the input program, but equal dependence can be expected for both. If the hardware is very communication–constrained, then many pass nodes may be needed for scheduling. However, if the program's intrinsic communication pattern mirrors the communication paths in the machine, then fewer pass nodes may be needed. Similarly, even if the machine is only mildly communication–constrained, a program could be devised to require a maximum number of pass nodes.

- The temperature probing algorithm is not entirely data–independent. The anomaly in runtimes for programs of different sizes suggests that the probing process gives temperatures that are relatively higher for the short program than the larger one.

- The algorithm has problems moving computations from one cluster to another when a direct data path is not present. Most of the schedules produced for the "hard" communication–constrained machine are confined to one or two clusters only. (The list scheduler schedules only a single cluster as well). Only once *ever* did the algorithm find the optimal solution using four clusters.

  These problems are probably due to the formulation of the simulated annealing data–dependent functions. Different **energy** and **reconfigure** functions may be able to move computations more efficiently.

- The algorithm is too slow, regardless of the schedule quality. Many of the datapoints for the communication–constrained tests took over four hours to compute, which is far too long to wait for programs that can be efficiently hand–scheduled in minutes. Perhaps such a long runtime is tolerable for extremely complex machines, but such machines are likely impractical.

6.3 Further Work

- Data–probing algorithms can be devised for the pass node parameters. Coming up with an accurate way to estimate the need for pass nodes in a schedule could make the algorithm much more consistent. Of course, the only way of doing this may be to run the algorithm and observe what happens. Dynamically changing pass–node parameters may work in this case, although simulated annealing generally does not use time varying **reconfigure** functions.

- Different reconfiguration primitives can be created for the scheduler. There are many scheduling algorithms based on different sets of transformations. Different transformations may open up a new space of schedules that are unreachable with the primitives used in this thesis. In particular, none of the primitives in this thesis allow code duplication, a common occurrence in other global instruction scheduling algorithms.

- Different energy functions may give better results. The functions used in this thesis focus on absolute schedule length, while more intelligent ones may optimize inner–loop throughput or most–likely trace length. In addition, more sophisticated penalties can be used. For example, a broken edge that would require two pass nodes to reconnect could receive a higher penalty than one that requires only a single pass node. Broken edges that can never be reconnected (e.g., no room for pass node because of precedence constraints) could be assigned an even greater penalty. Additionally, energy penalties could be assigned to inefficient use of resources, perhaps encouraging use of all machine resources even for non–compact schedules.

- A different combinatorial optimization algorithm could be used. Simulated annealing is good for some problems, but not for others. Randomized instruction scheduling still has promise even if simulated annealing is not the answer.

Appendix A

μasm Grammar

```
program:      statements
       ;
statements:   statements statement
            | statement
            ;
statement:    declaration ';'
            | assignment ';'
            | loop
            ;
decl_id:      ID
            | ID '[' INUM ']'
            ;
idlist:       idlist ',' decl_id
            | decl_id
            ;
declaration: TYPE idlist
            | UNSIGNED TYPE idlist
            | DOUBLE TYPE idlist
            | DOUBLE UNSIGNED TYPE idlist
            ;
ridentifier: ID
            | ID '[' INUM ']'
            | ID '[' ID ']'
            ;
lidentifier: ID
            | '[' ID ',' ID ']'
            | ID '[' INUM ']'
            | ID '[' ID ']'
            ;
assignment:   lidentifier '=' expr
            | OSTREAM '(' INUM ',' TYPE ')' '=' expr
            ;
exprlist:     exprlist ',' expr
            | expr
            ;
expr:         ridentifier
            | INUM
            | FNUM
            | '(' expr ')'
            | expr ORL expr
            | expr ANDL expr
            | expr AND expr
            | expr OR expr
            | expr EQ expr
            | expr COMPARE expr
            | expr SHIFT expr
            | expr ADD expr
            | expr MUL expr
            | NOTL expr
            | NOT expr
            | ID '?' expr ':' expr
            | FUNC '(' exprlist ')'
            | TYPE '(' expr ')'
            | UNSIGNED TYPE '(' expr ')'
            | ISTREAM '(' INUM ',' TYPE ')'
            | COMM '(' ridentifier ',' ID ')'
            | '[' expr ',' expr ']'
            ;
loop:         countloop
            ;
countloop:    LOOPP ID '=' INUM ',' INUM '{' statements '}'
            ;
```

Appendix B

Assembly Language Reference

Instruction	Operands	Description
`IADD{32,16,8}`	*dest, src1, src2*	word, half–word, byte add
`UADD{32,16,8}`	*dest, src1, src2*	word, half–word, byte unsigned add
`ISUB{32,16,8}`	*dest, src1, src2*	word, half–word, byte subtract
`USUB{32,16,8}`	*dest, src1, src2*	word, half–word, byte unsigned subtract
`IABS{32,16,8}`	*dest, src*	word, half–word, byte absolute value
`IMUL{32,16,8}`	*[dest1, dest2], [src1, src2]*	word, half–word, byte multiply
`UMUL{32,16,8}`	*[dest1, dest2], [src1, src2]*	word, half–word, byte unsigned multiply
`IDIV{32,16,8}`	*[dest1, dest2], [src1, src2]*	word, half–word, byte divide
`UDIV{32,16,8}`	*[dest1, dest2], [src1, src2]*	word, half–word, byte unsigned divide
`SHIFT{32,16,8}`	*dest, src1, src2*	word, half–word, byte shift
`SHIFTA{32,16,8}`	*dest, src1, src2*	word, half–word, byte arithmetic shift
`ROTATE{32,16,8}`	*dest, src1, src2*	word, half–word, byte rotate
`ANDL{32,16,8}`	*dest, src1, src2*	word, half–word, byte logical AND
`ORL{32,16,8}`	*dest, src1, src2*	word, half–word, byte logical OR
`XORL{32,16,8}`	*dest, src1, src2*	word, half–word, byte logical XOR
`NOTL{32,16,8}`	*dest, src*	word, half–word, byte logical NOT
`AND`	*dest, src1, src2*	bitwise AND
`OR`	*dest, src1, src2*	bitwise OR
`XOR`	*dest, src1, src2*	bitwise XOR

`NOT`	*dest, src*	bitwise NOT
`IEQ{32,16,8}`	*dest, src1, src2*	word, half–word, byte equal
`INEQ{32,16,8}`	*dest, src1, src2*	word, half–word, byte not–equal
`ILT{32,16,8}`	*dest, src1, src2*	word, half–word, byte less–than
`ULT{32,16,8}`	*dest, src1, src2*	word, half, byte unsigned less–than
`ILE{32,16,8}`	*dest, src1, src2*	word, half–word, byte less–equal
`ULE{32,16,8}`	*dest, src1, src2*	word, half, byte unsigned less–equal
`FADD`	*dest, src1, src2*	floating–point add
`FSUB`	*dest, src1, src2*	floating–point subtract
`FABS`	*dest, src*	floating–point absolute value
`FEQ`	*dest, src1, src2*	floating–point equal
`FNEQ`	*dest, src1, src2*	floating–point not–equal
`FLT`	*dest, src1, src2*	floating–point less–than
`FLE`	*dest, src1, src2*	floating–point less–or–equal
`FMUL`	*[dest1, dest2], [src1, src2]*	floating–point multiply
`FNORMS`	*dest, src*	single–prec. floating–pt. norm
`FNORMD`	*dest, [src1, src2]*	double–prec. floating–pt. norm
`FALIGN`	*[dest1, dest2], [src1, src2]*	floating–point mantissa align
`FDIV`	*[dest1, dest2], [src1, src2]*	floating–point divide
`FSQRT`	*dest, src*	floating–point square root
`FTOI`	*dest, src*	convert floating–point to integer
`ITOF`	*dest, src*	convert integer to floating–point
`SHUFFLE`	*dest, src1, src2*	byte shuffle
`ISELECT{32,16,8}`	*dest, cc–src, src1, src2*	word, half–word, byte select
`PASS`	*dest, src*	operand pass
`SETCC`	*cc–dest, src*	set condition code
`LOOP`	*#const*	loop start instruction
`END`		loop end instruction
`ISTREAM`	*dest, #const*	istream read
`OSTREAM`	*src, #const*	ostream write

Appendix C

Test Programs

C.1 paradd8.i

```
// paradd8.i
// add a sequence of numbers using tree of adds
// uses eight istreams

int num0, num1, num2, num3, num4, num5, num6, num7;

num0 = istream(0,int);
num1 = istream(1,int);
num2 = istream(2,int);
num3 = istream(3,int);
num4 = istream(4,int);
num5 = istream(5,int);
num6 = istream(6,int);
num7 = istream(7,int);

num0 = num0 + num1;
num1 = num2 + num3;
num2 = num4 + num5;
num3 = num6 + num7;

num0 = num0 + num1;
num1 = num2 + num3;

num0 = num0 + num1;
```

C.2 `paradd16.i`

```
// paradd16.i
// add a sequence of 16 numbers using tree of adds
// uses eight istreams

int num0, num1, num2, num3, num4, num5, num6, num7;
int sum0, sum1;

num0 = istream(0,int);
num1 = istream(1,int);
num2 = istream(2,int);
num3 = istream(3,int);
num4 = istream(4,int);
num5 = istream(5,int);
num6 = istream(6,int);
num7 = istream(7,int);

num0 = num0 + num1;
num1 = num2 + num3;
num2 = num4 + num5;
num3 = num6 + num7;

num0 = num0 + num1;
num1 = num2 + num3;

sum0 = num0 + num1;

num0 = istream(0,int);
num1 = istream(1,int);
num2 = istream(2,int);
num3 = istream(3,int);
num4 = istream(4,int);
num5 = istream(5,int);
num6 = istream(6,int);
num7 = istream(7,int);

num0 = num0 + num1;
num1 = num2 + num3;
num2 = num4 + num5;
num3 = num6 + num7;

num0 = num0 + num1;
num1 = num2 + num3;
```

```
sum1 = num0 + num1;

sum0 = sum0 + sum1;
```

Appendix D

Test Machine Descriptions

D.1 small_single_bus.md

```
cluster small_single_bus
{
  unit ADDER
  {
    inputs[2];
    outputs[1];
    operations = (FADD, IADD32, IADD16, IADD8, UADD32, UADD16, UADD8,
                  FSUB, ISUB32, ISUB16, ISUB8, USUB32, USUB16, USUB8,
                  FABS, IABS32, IABS16, IABS8, IANDL32, IANDL16, IANDL8,
                  IORL32,  IORL16,  IORL8, IXORL32, IXORL16, IXORL8,
                  INOTL32, INOTL16, INOTL8,
                  FEQ,   IEQ32,  IEQ16,  IEQ8, FNEQ, INEQ32, INEQ16, INEQ8,
                  FLT,   ILT32,  ILT16,  ILT8,  ULT32,  ULT16,  ULT8,
                  FLE,   ILE32,  ILE16,  ILE8,  ULE32,  ULE16,  ULE8,
                  ISELECT32, ISELECT16, ISELECT8, PASS,
                  IAND, IOR, IXOR, INOT, CCWRITE);
    latency = 2;
    pipelined = yes;
    area = 30;
  };

  unit MULTIPLIER
  {
    inputs[2];
    outputs[2];
    operations = (FMUL, IMUL32, IMUL16, IMUL8, UMUL32, UMUL16, UMUL8, PASS);
    latency = 3;
    pipelined = yes;
    area = 300;
  };

  unit SHIFTER
  {
```

```
  inputs[2];
  outputs[2];
  operations = (USHIFT32, USHIFT16, USHIFT8,
                USHIFTF32, USHIFTF16, USHIFTF8,
                USHIFTA32, USHIFTA16, USHIFTA8,
                UROTATE32, UROTATE16, UROTATE8,
                FNORMS, FNORMD, FALIGN, FTOI, ITOF, USHUFFLE, PASS);
  latency = 1;
  pipelined = yes;
  area = 200;
};

unit DIVIDER
{
  inputs[2];
  outputs[2];
  operations = (FDIV, FSQRT, IDIV32, IDIV16, IDIV8, UDIV32, UDIV16, UDIV8);
  latency = 5;
  pipelined = no;
  area = 300;
};

unit MC
{
  inputs[0];
  outputs[0];
  operations = (COUNT, WHILE, STREAM, END);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT0  {
  inputs[0];
  outputs[1];
  operations = (IN0);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT1  {
  inputs[0];
  outputs[1];
  operations = (IN1);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT2  {
  inputs[0];
  outputs[1];
  operations = (IN2);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT3  {
  inputs[0];
  outputs[1];
  operations = (IN3);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT4  {
  inputs[0];
  outputs[1];
  operations = (IN4);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT5  {
  inputs[0];
  outputs[1];
  operations = (IN5);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT6  {
  inputs[0];
  outputs[1];
  operations = (IN6);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT7  {
  inputs[0];
  outputs[1];
  operations = (IN7);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit OUTPUT0  {
  inputs[1];
  outputs[0];
  operations = (OUT0);
```

```
    latency = 0;
    pipelined = yes;
    area = 0;
  };

  unit OUTPUT1  {
    inputs[1];
    outputs[0];
    operations = (OUT1);
    latency = 0;
    pipelined = yes;
    area = 0;
  };

  regfile OUTPUTREG
  {
    inputs[1];
    outputs[1];
    size = 8;
    area = 8;
  };

  regfile DATAREGFILE
  {
    inputs[1];
    outputs[1];
    size = 8;
    area = 64;
  };

  ADDER[1],
  MULTIPLIER[1],
  SHIFTER[1],
  DIVIDER[1],
  INPUT0[1],
  INPUT1[1],
  INPUT2[1],
  INPUT3[1],
  INPUT4[1],
  INPUT5[1],
  INPUT6[1],
  INPUT7[1],
  OUTPUT0[1],
  OUTPUT1[1],
  MC[1],
  BUS[10],
  DATAREGFILE[8],
  OUTPUTREG[2];

// unit -> network connections

  ( ADDER[0:0].out[0], MULTIPLIER[0:0].out[0],
    SHIFTER[0:0].out[0], DIVIDER[0:0].out[0] ),
  ( MULTIPLIER[0:0].out[1], SHIFTER[0:0].out[1],
    DIVIDER[0:0].out[1] )  -> BUS[0:1].in[0];

  INPUT0[0].out[0] -> BUS[2].in[0];
  INPUT1[0].out[0] -> BUS[3].in[0];
  INPUT2[0].out[0] -> BUS[4].in[0];
  INPUT3[0].out[0] -> BUS[5].in[0];
  INPUT4[0].out[0] -> BUS[6].in[0];
  INPUT5[0].out[0] -> BUS[7].in[0];
  INPUT6[0].out[0] -> BUS[8].in[0];
  INPUT7[0].out[0] -> BUS[9].in[0];

//  register file -> unit connections

  DATAREGFILE[0:7].out[0:0] -> ADDER[0:0].in[0:1], MULTIPLIER[0:0].in[0:1],
                               SHIFTER[0:0].in[0:1], DIVIDER[0:0].in[0:1];
  OUTPUTREG[0].out[0] -> OUTPUT0[0].in[0];
  OUTPUTREG[1].out[0] -> OUTPUT1[0].in[0];

//  network -> register file connections

  ( BUS[0:9].out[0] ) -> ( DATAREGFILE[0:7].in[0:0] , OUTPUTREG[0:1].in[0] );
}
```

D.2 large_multi_bus.md

```
cluster large_multi_bus
{
  unit ADDER
  {
    inputs[2];
    outputs[1];
    operations = (FADD, IADD32, IADD16, IADD8, UADD32, UADD16, UADD8,
                  FSUB, ISUB32, ISUB16, ISUB8, USUB32, USUB16, USUB8,
                  FABS, IABS32, IABS16, IABS8, IANDL32, IANDL16, IANDL8,
                  IORL32,  IORL16,  IORL8, IXORL32, IXORL16, IXORL8,
                  INOTL32, INOTL16, INOTL8,
                  FEQ,   IEQ32,  IEQ16,  IEQ8, FNEQ, INEQ32, INEQ16, INEQ8,
                  FLT,   ILT32,  ILT16,  ILT8,  ULT32,  ULT16,  ULT8,
                  FLE,   ILE32,  ILE16,  ILE8,  ULE32,  ULE16,  ULE8,
                  ISELECT32, ISELECT16, ISELECT8, PASS,
                  IAND, IOR, IXOR, INOT, CCWRITE);
    latency = 2;
    pipelined = yes;
    area = 30;
  };

  unit MULTIPLIER
  {
    inputs[2];
    outputs[2];
    operations = (FMUL, IMUL32, IMUL16, IMUL8, UMUL32, UMUL16, UMUL8, PASS);
    latency = 3;
    pipelined = yes;
    area = 300;
  };

  unit SHIFTER
  {
    inputs[2];
    outputs[2];
    operations = (USHIFT32, USHIFT16, USHIFT8,
                  USHIFTF32, USHIFTF16, USHIFTF8,
                  USHIFTA32, USHIFTA16, USHIFTA8,
                  UROTATE32, UROTATE16, UROTATE8,
                  FNORMS, FNORMD, FALIGN, FTOI, ITOF, USHUFFLE, PASS);
    latency = 1;
    pipelined = yes;
    area = 200;
  };

  unit DIVIDER
  {
    inputs[2];
    outputs[2];
    operations = (FDIV, FSQRT, IDIV32, IDIV16, IDIV8, UDIV32, UDIV16, UDIV8);
    latency = 5;
    pipelined = no;
  area = 300;
};

unit MC
{
  inputs[0];
  outputs[0];
  operations = (COUNT, WHILE, STREAM, END);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT0  {
  inputs[0];
  outputs[1];
  operations = (IN0);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT1  {
  inputs[0];
  outputs[1];
  operations = (IN1);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT2  {
  inputs[0];
  outputs[1];
  operations = (IN2);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT3  {
  inputs[0];
  outputs[1];
  operations = (IN3);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT4  {
  inputs[0];
  outputs[1];
  operations = (IN4);
  latency = 0;
```

```
    pipelined = yes;
    area = 0;
  };

  unit INPUT5  {
    inputs[0];
    outputs[1];
    operations = (IN5);
    latency = 0;
    pipelined = yes;
    area = 0;
  };

  unit INPUT6  {
    inputs[0];
    outputs[1];
    operations = (IN6);
    latency = 0;
    pipelined = yes;
    area = 0;
  };

  unit INPUT7  {
    inputs[0];
    outputs[1];
    operations = (IN7);
    latency = 0;
    pipelined = yes;
    area = 0;
  };

  unit OUTPUT0  {
    inputs[1];
    outputs[0];
    operations = (OUT0);
    latency = 0;
    pipelined = yes;
    area = 0;
  };

  unit OUTPUT1  {
    inputs[1];
    outputs[0];
    operations = (OUT1);
    latency = 0;
    pipelined = yes;
    area = 0;
  };

  regfile OUTPUTREG
  {
    inputs[1];
    outputs[1];
    size = 8;
    area = 8;
  };

  regfile DATAREGFILE
  {
    inputs[1];
    outputs[1];
    size = 8;
    area = 64;
  };

  ADDER[4],
  MULTIPLIER[4],
  SHIFTER[4],
  DIVIDER[4],
  INPUT0[1],
  INPUT1[1],
  INPUT2[1],
  INPUT3[1],
  INPUT4[1],
  INPUT5[1],
  INPUT6[1],
  INPUT7[1],
  OUTPUT0[1],
  OUTPUT1[1],
  MC[1],
  BUS[36],
  DATAREGFILE[32],
  OUTPUTREG[2];

// unit -> network connections
  ADDER[0:3].out[0], MULTIPLIER[0:3].out[0:1],
  SHIFTER[0:3].out[0:1], DIVIDER[0:3].out[0:1]  -> BUS[0:27].in[0];

  INPUT0[0].out[0] -> BUS[28].in[0];
  INPUT1[0].out[0] -> BUS[29].in[0];
  INPUT2[0].out[0] -> BUS[30].in[0];
  INPUT3[0].out[0] -> BUS[31].in[0];
  INPUT4[0].out[0] -> BUS[32].in[0];
  INPUT5[0].out[0] -> BUS[33].in[0];
  INPUT6[0].out[0] -> BUS[34].in[0];
  INPUT7[0].out[0] -> BUS[35].in[0];

//  register file -> unit connections
  DATAREGFILE[0:31].out[0:0] -> ADDER[0:3].in[0:1], MULTIPLIER[0:3].in[0:1],
                                SHIFTER[0:3].in[0:1], DIVIDER[0:3].in[0:1];
  OUTPUTREG[0].out[0] -> OUTPUT0[0].in[0];
  OUTPUTREG[1].out[0] -> OUTPUT1[0].in[0];

//  network -> register file connections
  ( BUS[0:35].out[0] ) -> ( DATAREGFILE[0:31].in[0:0] , OUTPUTREG[0:1].in[0] );
}
```

D.3 cluster_with_move.md

```
cluster cluster_with_move
{
  unit ADDER
  {
    inputs[2];
    outputs[1];
    operations = (FADD, IADD32, IADD16, IADD8, UADD32, UADD16, UADD8,
                  FSUB, ISUB32, ISUB16, ISUB8, USUB32, USUB16, USUB8,
                  FABS, IABS32, IABS16, IABS8, IANDL32, IANDL16, IANDL8,
                  IORL32,  IORL16,  IORL8, IXORL32, IXORL16, IXORL8,
                  INOTL32, INOTL16, INOTL8,
                  FEQ,   IEQ32,  IEQ16,  IEQ8, FNEQ, INEQ32, INEQ16, INEQ8,
                  FLT,   ILT32,  ILT16,  ILT8,  ULT32,  ULT16,  ULT8,
                  FLE,   ILE32,  ILE16,  ILE8,  ULE32,  ULE16,  ULE8,
                  ISELECT32, ISELECT16, ISELECT8, PASS,
                  IAND, IOR, IXOR, INOT, CCWRITE);
    latency = 2;
    pipelined = yes;
    area = 30;
  };

  unit MULTIPLIER
  {
    inputs[2];
    outputs[2];
    operations = (FMUL, IMUL32, IMUL16, IMUL8, UMUL32, UMUL16, UMUL8, PASS);
    latency = 3;
    pipelined = yes;
    area = 300;
  };

  unit SHIFTER
  {
    inputs[2];
    outputs[2];
    operations = (USHIFT32, USHIFT16, USHIFT8,
                  USHIFTF32, USHIFTF16, USHIFTF8,
                  USHIFTA32, USHIFTA16, USHIFTA8,
                  UROTATE32, UROTATE16, UROTATE8,
                  FNORMS, FNORMD, FALIGN, FTOI, ITOF, USHUFFLE, PASS);
    latency = 1;
    pipelined = yes;
    area = 200;
  };

  unit DIVIDER
  {
    inputs[2];
    outputs[2];
    operations = (FDIV, FSQRT, IDIV32, IDIV16, IDIV8, UDIV32, UDIV16, UDIV8);
    latency = 5;
    pipelined = no;
    area = 300;
  };

  unit MOVER
  {
inputs[1];
outputs[1];
operations = (PASS);
latency = 0;
pipelined = yes;
area = 100;
  };

  unit MC
  {
    inputs[0];
    outputs[0];
    operations = (COUNT, WHILE, STREAM, END);
    latency = 0;
    pipelined = yes;
    area = 0;
  };

  unit INPUT0  {
    inputs[0];
    outputs[1];
    operations = (IN0);
    latency = 0;
    pipelined = yes;
    area = 0;
  };

  unit INPUT1  {
    inputs[0];
    outputs[1];
    operations = (IN1);
    latency = 0;
    pipelined = yes;
    area = 0;
  };

  unit INPUT2  {
    inputs[0];
    outputs[1];
    operations = (IN2);
    latency = 0;
    pipelined = yes;
    area = 0;
  };

  unit INPUT3  {
    inputs[0];
    outputs[1];
    operations = (IN3);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT4  {
  inputs[0];
  outputs[1];
  operations = (IN4);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT5  {
  inputs[0];
  outputs[1];
  operations = (IN5);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT6  {
  inputs[0];
  outputs[1];
  operations = (IN6);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT7  {
  inputs[0];
  outputs[1];
  operations = (IN7);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit OUTPUT0  {
  inputs[1];
  outputs[0];
  operations = (OUT0);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit OUTPUT1  {
  inputs[1];
  outputs[0];
  operations = (OUT1);
```

```
    latency = 0;
    pipelined = yes;
    area = 0;
  };

  regfile OUTPUTREG
  {
    inputs[1];
    outputs[1];
    size = 8;
    area = 8;
  };

  regfile DATAREGFILE
  {
    inputs[1];
    outputs[1];
    size = 8;
    area = 64;
  };

  ADDER[4],
  MULTIPLIER[4],
  SHIFTER[4],
  DIVIDER[4],
  MOVER[4],
  INPUT0[1],
  INPUT1[1],
  INPUT2[1],
  INPUT3[1],
  INPUT4[1],
  INPUT5[1],
  INPUT6[1],
  INPUT7[1],
  OUTPUT0[1],
  OUTPUT1[1],
  MC[1],
  BUS[44],
  DATAREGFILE[36],
  OUTPUTREG[2];

// 9 busses per cluster, 7 for internal data, 2 for moved data x 4 clusters
// + 6 busses for input units = 42 busses total

// unit -> network connections

  // cluster 0 contains units 0 of each type
  // cluster 0 uses bus 0:6 for internal data, bus 7,38 for moved data
  ADDER[0].out[0], MULTIPLIER[0].out[0],
  SHIFTER[0].out[0], DIVIDER[0].out[0] -> BUS[0:3].in[0];
  MULTIPLIER[0].out[1], SHIFTER[0].out[1], DIVIDER[0].out[1] -> BUS[4:6].in[0];
  MOVER[0].out[0] -> ( BUS[15].in[0], BUS[41].in[0] );

  // cluster 1 contains units 1 of each type
  // cluster 1 uses bus 8:14 for internal data, bus 15,39 for moved data
  ADDER[1].out[0], MULTIPLIER[1].out[0],
  SHIFTER[1].out[0], DIVIDER[1].out[0] -> BUS[8:11].in[0];
  MULTIPLIER[1].out[1], SHIFTER[1].out[1], DIVIDER[1].out[1] -> BUS[12:14].in[0];
  MOVER[1].out[0] -> ( BUS[23].in[0], BUS[38].in[0] );

  // cluster 2 contains units 2 of each type
  // cluster 2 uses bus 16:22 for internal data, bus 23,40 for moved data
  ADDER[2].out[0], MULTIPLIER[2].out[0],
  SHIFTER[2].out[0], DIVIDER[2].out[0] -> BUS[16:19].in[0];
  MULTIPLIER[2].out[1], SHIFTER[2].out[1], DIVIDER[2].out[1] -> BUS[20:22].in[0];
  MOVER[2].out[0] -> ( BUS[31].in[0], BUS[39].in[0] );

  // cluster 3 contains units 3 of each type
  // cluster 3 uses bus 24:30 for internal data, bus 31,41 for moved data
  ADDER[3].out[0], MULTIPLIER[3].out[0],
  SHIFTER[3].out[0], DIVIDER[3].out[0] -> BUS[24:27].in[0];
  MULTIPLIER[3].out[1], SHIFTER[3].out[1], DIVIDER[3].out[1] -> BUS[28:30].in[0];
  MOVER[3].out[0] -> ( BUS[7].in[0], BUS[40].in[0] );

  // input units write to busses 32 - 37
  INPUT0[0].out[0] -> BUS[32].in[0];
  INPUT1[0].out[0] -> BUS[33].in[0];
  INPUT2[0].out[0] -> BUS[34].in[0];
  INPUT3[0].out[0] -> BUS[35].in[0];
  INPUT4[0].out[0] -> BUS[36].in[0];
  INPUT5[0].out[0] -> BUS[37].in[0];
  INPUT6[0].out[0] -> BUS[42].in[0];
  INPUT7[0].out[0] -> BUS[43].in[0];

//  register file -> unit connections

  // cluster 0
  DATAREGFILE[0:8].out[0:0] -> ADDER[0].in[0:1], MULTIPLIER[0].in[0:1],
                               SHIFTER[0].in[0:1], DIVIDER[0].in[0:1], MOVER[0].in[0];

  // cluster 1
  DATAREGFILE[9:17].out[0:0] -> ADDER[1].in[0:1], MULTIPLIER[1].in[0:1],
                                SHIFTER[1].in[0:1], DIVIDER[1].in[0:1], MOVER[1].in[0];

  // cluster 2
  DATAREGFILE[18:26].out[0:0] -> ADDER[2].in[0:1], MULTIPLIER[2].in[0:1],
                                 SHIFTER[2].in[0:1], DIVIDER[2].in[0:1], MOVER[2].in[0];

  // cluster 3
  DATAREGFILE[27:35].out[0:0] -> ADDER[3].in[0:1], MULTIPLIER[3].in[0:1],
                                 SHIFTER[3].in[0:1], DIVIDER[3].in[0:1], MOVER[3].in[0];

  OUTPUTREG[0].out[0] -> OUTPUT0[0].in[0];
  OUTPUTREG[1].out[0] -> OUTPUT1[0].in[0];

//  network -> register file connections

  // cluster 0
  ( BUS[0:7].out[0], BUS[38].out[0] ) -> ( DATAREGFILE[0:8].in[0], OUTPUTREG[0:1].in[0] );

  // cluster 1
  ( BUS[8:15].out[0], BUS[39].out[0] ) -> ( DATAREGFILE[9:17].in[0], OUTPUTREG[0:1].in[0] );

  // cluster 2
  ( BUS[16:23].out[0], BUS[40].out[0] ) -> ( DATAREGFILE[18:26].in[0], OUTPUTREG[0:1].in[0] );

  // cluster 3
  ( BUS[24:31].out[0], BUS[41].out[0] ) -> ( DATAREGFILE[27:35].in[0], OUTPUTREG[0:1].in[0] );

  // global
  (BUS[32:37].out[0], BUS[42:43].out[0]) -> (DATAREGFILE[0:35].in[0:0],OUTPUTREG[0:1].in[0]);
}
```

D.4 cluster_without_move.md

```
cluster cluster_without_move
{
  unit ADDER
  {
    inputs[2];
    outputs[1];
    operations = (FADD, IADD32, IADD16, IADD8, UADD32, UADD16, UADD8,
                  FSUB, ISUB32, ISUB16, ISUB8, USUB32, USUB16, USUB8,
                  FABS, IABS32, IABS16, IABS8, IANDL32, IANDL16, IANDL8,
                  IORL32,  IORL16,  IORL8, IXORL32, IXORL16, IXORL8,
                  INOTL32, INOTL16, INOTL8,
                  FEQ,   IEQ32,  IEQ16,  IEQ8, FNEQ, INEQ32, INEQ16, INEQ8,
                  FLT,   ILT32,  ILT16,  ILT8,  ULT32,  ULT16,  ULT8,
                  FLE,   ILE32,  ILE16,  ILE8,  ULE32,  ULE16,  ULE8,
                  ISELECT32, ISELECT16, ISELECT8, PASS,
                  IAND, IOR, IXOR, INOT, CCWRITE);
    latency = 2;
    pipelined = yes;
    area = 30;
  };

  unit MULTIPLIER
  {
    inputs[2];
    outputs[2];
    operations = (FMUL, IMUL32, IMUL16, IMUL8, UMUL32, UMUL16, UMUL8, PASS);
    latency = 3;
    pipelined = yes;
    area = 300;
  };

  unit SHIFTER
  {
    inputs[2];
    outputs[2];
    operations = (USHIFT32, USHIFT16, USHIFT8,
                  USHIFTF32, USHIFTF16, USHIFTF8,
                  USHIFTA32, USHIFTA16, USHIFTA8,
                  UROTATE32, UROTATE16, UROTATE8,
                  FNORMS, FNORMD, FALIGN, FTOI, ITOF, USHUFFLE, PASS);
    latency = 1;
    pipelined = yes;
    area = 200;
  };

  unit DIVIDER
  {
    inputs[2];
    outputs[2];
    operations = (FDIV, FSQRT, IDIV32, IDIV16, IDIV8, UDIV32, UDIV16, UDIV8);
    latency = 5;
    pipelined = no;
  area = 300;
};

unit MC
{
  inputs[0];
  outputs[0];
  operations = (COUNT, WHILE, STREAM, END);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT0  {
  inputs[0];
  outputs[1];
  operations = (IN0);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT1  {
  inputs[0];
  outputs[1];
  operations = (IN1);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT2  {
  inputs[0];
  outputs[1];
  operations = (IN2);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT3  {
  inputs[0];
  outputs[1];
  operations = (IN3);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT4  {
  inputs[0];
  outputs[1];
  operations = (IN4);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT5  {
  inputs[0];
  outputs[1];
  operations = (IN5);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT6  {
  inputs[0];
  outputs[1];
  operations = (IN6);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit INPUT7  {
  inputs[0];
  outputs[1];
  operations = (IN7);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit OUTPUT0  {
  inputs[1];
  outputs[0];
  operations = (OUT0);
  latency = 0;
  pipelined = yes;
  area = 0;
};

unit OUTPUT1  {
  inputs[1];
  outputs[0];
  operations = (OUT1);
  latency = 0;
  pipelined = yes;
  area = 0;
};

regfile OUTPUTREG
{
  inputs[1];
  outputs[1];
  size = 8;
```

```
    area = 8;
  };

  regfile DATAREGFILE
  {
    inputs[1];
    outputs[1];
    size = 8;
    area = 64;
  };
  ADDER[4],
  MULTIPLIER[4],
  SHIFTER[4],
  DIVIDER[4],
  INPUT0[1],
  INPUT1[1],
  INPUT2[1],
  INPUT3[1],
  INPUT4[1],
  INPUT5[1],
  INPUT6[1],
  INPUT7[1],
  OUTPUT0[1],
  OUTPUT1[1],
  MC[1],
  BUS[36],
  DATAREGFILE[32],
  OUTPUTREG[2];

// 7 busses per cluster, 7 for internal data x 4 clusters
// + 6 busses for input units = 34 busses total

// unit -> network connections
  // cluster 0 contains units 0 of each type
  // cluster 0 writes to bus 0:6, reads from 21:27
  ADDER[0].out[0], MULTIPLIER[0].out[0],
  SHIFTER[0].out[0], DIVIDER[0].out[0] -> BUS[0:3].in[0];
  MULTIPLIER[0].out[1], SHIFTER[0].out[1], DIVIDER[0].out[1] -> BUS[4:6].in[0];

  // cluster 1 contains units 1 of each type
  // cluster 1 writes to bus 7:13, reads from 0:6
  ADDER[1].out[0], MULTIPLIER[1].out[0],
  SHIFTER[1].out[0], DIVIDER[1].out[0] -> BUS[7:10].in[0];
  MULTIPLIER[1].out[1], SHIFTER[1].out[1], DIVIDER[1].out[1] -> BUS[11:13].in[0];

  // cluster 2 contains units 2 of each type
  // cluster 2 writes to bus 14:20, reads from 7:13
  ADDER[2].out[0], MULTIPLIER[2].out[0],
  SHIFTER[2].out[0], DIVIDER[2].out[0] -> BUS[14:17].in[0];
  MULTIPLIER[2].out[1], SHIFTER[2].out[1], DIVIDER[2].out[1] -> BUS[18:20].in[0];

  // cluster 3 contains units 3 of each type
  // cluster 3 writes to bus 21:27, reads from 14:20
  ADDER[3].out[0], MULTIPLIER[3].out[0],
    SHIFTER[3].out[0], DIVIDER[3].out[0] -> BUS[21:24].in[0];
    MULTIPLIER[3].out[1], SHIFTER[3].out[1], DIVIDER[3].out[1] -> BUS[25:27].in[0];

    // input units write to busses 28:33
    INPUT0[0].out[0] -> BUS[28].in[0];
    INPUT1[0].out[0] -> BUS[29].in[0];
    INPUT2[0].out[0] -> BUS[30].in[0];
    INPUT3[0].out[0] -> BUS[31].in[0];
    INPUT4[0].out[0] -> BUS[32].in[0];
    INPUT5[0].out[0] -> BUS[33].in[0];
    INPUT6[0].out[0] -> BUS[34].in[0];
    INPUT7[0].out[0] -> BUS[35].in[0];

//  register file -> unit connections
    // cluster 0
    DATAREGFILE[0:7].out[0:0] -> ADDER[0].in[0:1], MULTIPLIER[0].in[0:1],
                                 SHIFTER[0].in[0:1], DIVIDER[0].in[0:1];

    // cluster 1
    DATAREGFILE[8:15].out[0:0] -> ADDER[1].in[0:1], MULTIPLIER[1].in[0:1],
                                  SHIFTER[1].in[0:1], DIVIDER[1].in[0:1];

    // cluster 2
    DATAREGFILE[16:23].out[0:0] -> ADDER[2].in[0:1], MULTIPLIER[2].in[0:1],
                                   SHIFTER[2].in[0:1], DIVIDER[2].in[0:1];

    // cluster 3
    DATAREGFILE[24:31].out[0:0] -> ADDER[3].in[0:1], MULTIPLIER[3].in[0:1],
                                   SHIFTER[3].in[0:1], DIVIDER[3].in[0:1];

    OUTPUTREG[0].out[0] -> OUTPUT0[0].in[0];
    OUTPUTREG[1].out[0] -> OUTPUT1[0].in[0];

//  network -> register file connections
    // cluster 0
    ( BUS[21:27].out[0], BUS[7:13].out[0] ) -> (DATAREGFILE[0:7].in[0],OUTPUTREG[0:1].in[0]);

    // cluster 1
    ( BUS[0:6].out[0], BUS[14:20].out[0] ) -> (DATAREGFILE[8:15].in[0],OUTPUTREG[0:1].in[0]);

    // cluster 2
    ( BUS[7:13].out[0], BUS[21:27].out[0] ) -> (DATAREGFILE[16:23].in[0],OUTPUTREG[0:1].in[0]);

    // cluster 3
    ( BUS[14:20].out[0], BUS[0:6].out[0] ) -> (DATAREGFILE[24:31].in[0],OUTPUTREG[0:1].in[0]);

    // global
    ( BUS[28:35].out[0] ) -> ( DATAREGFILE[0:31].in[0:0] , OUTPUTREG[0:1].in[0] );
}
```

Appendix E

Experimental Data

E.1 Annealing Experiments

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
Program `paradd8.i` on machine configuration `small_single_bus.md` with maximally–bad initialization.						
0.05	0.5	11	52	1400	1524	8.312
0.05	0.55	11	51	1300	1431	7.797
0.05	0.6	11	51	1800	1990	10
0.05	0.65	11	52	1700	1867	10.609
0.05	0.7	11	51	1900	2146	11.141
0.05	0.75	11	51	1600	1782	8.953
0.05	0.8	11	52	2000	2169	10.047
0.05	0.85	11	51	2600	2895	14.063
0.05	0.9	11	51	3400	3719	18.125
0.05	0.95	11	49	6400	7158	31.86
0.05	0.99	11	52	11400	12211	62.281
0.1	0.5	11	52	1700	1816	9.031
0.1	0.55	11	51	1500	1684	8.797
0.1	0.6	11	51	1900	2079	10.157
0.1	0.65	11	52	1700	1854	9.688
0.1	0.7	11	52	2200	2398	12.563
0.1	0.75	11	52	1700	1857	9.219
0.1	0.8	12	59	2300	2499	13.734
0.1	0.85	11	51	2700	2896	15.094
0.1	0.9	11	52	3200	3442	15.578
0.1	0.95	11	52	3500	3707	19.781
0.1	0.99	11	52	8500	9155	47.531
0.20	0.5	11	51	1600	1781	9.578

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.20	0.55	11	51	2800	3068	16.281
0.20	0.6	11	51	1800	1961	9.578
0.20	0.65	11	52	2000	2143	10.735
0.20	0.7	11	51	2500	2753	15.109
0.20	0.75	11	51	2200	2388	12.079
0.20	0.8	11	52	2400	2525	13.156
0.20	0.85	11	52	2900	3125	16.109
0.20	0.9	11	52	3600	3878	19.594
0.20	0.95	11	52	6700	7224	33.984
0.20	0.99	11	51	27300	29377	145.719
0.40	0.5	11	52	1900	2024	10.656
0.40	0.55	11	52	2800	2976	15.063
0.40	0.6	11	52	2600	2791	15.468
0.40	0.65	11	51	2000	2177	11.64
0.40	0.7	11	50	2900	3233	17.922
0.40	0.75	11	52	2700	2877	15.657
0.40	0.8	11	51	3700	3954	19.984
0.40	0.85	11	51	3800	4080	20.734
0.40	0.9	11	52	6300	6776	33.343
0.40	0.95	11	52	10600	11314	60.578
0.40	0.99	11	49	41700	44639	234.969
0.60	0.5	11	52	2700	2865	16.828
0.60	0.55	11	52	2200	2351	12.421
0.60	0.6	11	52	2600	2741	15.671
0.60	0.65	11	51	3000	3201	17.953
0.60	0.7	11	52	3400	3575	20.391
0.60	0.75	11	51	3400	3662	19.375
0.60	0.8	11	52	4200	4457	24.219
0.60	0.85	11	52	6100	6409	33.859
0.60	0.9	11	51	9800	10512	57.204
0.60	0.95	11	51	15000	16066	86.063
0.60	0.99	11	49	71400	76149	390.812
0.80	0.5	11	51	2900	3113	16.5
0.80	0.55	11	51	4000	4226	22.766
0.80	0.6	11	52	3800	3955	21.813
0.80	0.65	11	51	4300	4582	24.094
0.80	0.7	11	51	5900	6221	33.469
0.80	0.75	11	52	4800	5036	27.25
0.80	0.8	11	51	8500	8919	46.453
0.80	0.85	11	51	7900	8368	46.141
0.80	0.9	11	51	13700	14481	79.234
0.80	0.95	11	49	24300	25463	134.922
0.80	0.99	11	49	103700	108995	585.922
0.90	0.5	11	51	2900	3063	17.031

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.90	0.55	11	52	3500	3657	19.438
0.90	0.6	11	52	3800	4020	22.203
0.90	0.65	11	51	5000	5259	29.61
0.90	0.7	11	52	5800	6059	32.234
0.90	0.75	11	52	6100	6386	35.953
0.90	0.8	11	49	9200	9595	51.875
0.90	0.85	11	51	10900	11385	60.922
0.90	0.9	11	51	17000	17681	94.343
0.90	0.95	11	49	32200	33773	177.047
0.90	0.99	11	49	157200	163447	878.421
0.99	0.5	11	51	5400	5577	30.688
0.99	0.55	11	52	7300	7462	43.203
0.99	0.6	11	49	8600	8875	48.437
0.99	0.65	11	52	5800	5991	35.297
0.99	0.7	12	52	8400	8634	48
0.99	0.75	11	51	10800	11123	62.438
0.99	0.8	11	51	17000	17462	95.844
0.99	0.85	11	49	16900	17510	95.188
0.99	0.9	11	51	27200	27969	150.625
0.99	0.95	11	50	56400	57803	312.703
0.99	0.99	11	49	269100	275221	1496.41
Program `paradd8.i` on machine configuration `small_single_bus.md` with list–scheduler initialization.						
0.05	0.5	11	51	1000	1089	8.718
0.05	0.55	11	51	1000	1089	8.718
0.05	0.6	11	51	1200	1307	10.328
0.05	0.65	11	51	1200	1307	10.062
0.05	0.7	11	51	1200	1307	10.328
0.05	0.75	11	51	1400	1512	11.219
0.05	0.8	11	51	1400	1512	11.359
0.05	0.85	11	51	1400	1512	11.219
0.05	0.9	11	51	1400	1511	11.172
0.05	0.95	11	51	1600	1729	12.485
0.05	0.99	11	51	2400	2608	17.625
0.1	0.5	11	51	1000	1089	8.828
0.1	0.55	11	51	1200	1307	10.281
0.1	0.6	11	51	1200	1307	10.422
0.1	0.65	11	51	1200	1305	10.328
0.1	0.7	11	51	1400	1512	11.438
0.1	0.75	11	51	1600	1730	12.672
0.1	0.8	11	51	1600	1729	12.61
0.1	0.85	11	51	1600	1726	12.719
0.1	0.9	11	51	1800	1955	14.031
0.1	0.95	11	51	2400	2606	17.718

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.1	0.99	11	51	6000	6519	41.047
0.2	0.5	11	51	1200	1307	10.406
0.2	0.55	11	51	1400	1515	11.453
0.2	0.6	11	51	1600	1730	12.625
0.2	0.65	11	51	1600	1730	12.516
0.2	0.7	11	51	1600	1729	12.75
0.2	0.75	11	51	1600	1726	12.687
0.2	0.8	11	51	1700	1834	13.344
0.2	0.85	11	51	2100	2273	15.703
0.2	0.9	11	51	2900	3153	20.969
0.2	0.95	11	51	5000	5408	34.157
0.2	0.99	11	51	13500	14499	88.563
0.4	0.5	11	51	1500	1612	12.265
0.4	0.55	11	51	1500	1612	12.312
0.4	0.6	11	51	1700	1829	13.375
0.4	0.65	11	51	1700	1827	13.282
0.4	0.7	11	51	1700	1827	13.281
0.4	0.75	11	51	1700	1822	13.39
0.4	0.8	11	51	2900	3144	21
0.4	0.85	11	51	2900	3138	20.937
0.4	0.9	11	51	4900	5305	33.89
0.4	0.95	11	51	7500	8046	50.718
0.4	0.99	11	51	23900	25558	148.125
0.6	0.5	11	51	1700	1825	13.359
0.6	0.55	11	51	1700	1830	13.422
0.6	0.6	11	51	2200	2396	17.297
0.6	0.65	11	51	2500	2738	18.25
0.6	0.7	11	51	3200	3527	22.968
0.6	0.75	11	51	2600	2775	18.735
0.6	0.8	11	51	2900	3088	21.906
0.6	0.85	11	51	3500	3733	25.297
0.6	0.9	11	51	6600	7053	45.172
0.6	0.95	11	51	11800	12578	79.844
0.6	0.99	11	51	38500	41254	248.781
0.8	0.5	11	51	2300	2430	17.266
0.8	0.55	11	51	4100	4392	29.235
0.8	0.6	11	51	2700	2839	19.469
0.8	0.65	11	51	4100	4416	29.219
0.8	0.7	11	51	3400	3602	25.125
0.8	0.75	11	51	4000	4296	29.609
0.8	0.8	11	51	4400	4683	32.469
0.8	0.85	11	51	5800	6161	40.157
0.8	0.9	11	51	9500	10104	64.031
0.8	0.95	11	51	20300	21686	138.968

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.8	0.99	11	49	81100	86125	537.812
0.9	0.5	11	51	2200	2356	16
0.9	0.55	11	51	4000	4275	28.985
0.9	0.6	11	51	2800	2955	20.672
0.9	0.65	11	51	2600	2785	18.484
0.9	0.7	11	51	4500	4744	32.687
0.9	0.75	11	51	5000	5276	35.031
0.9	0.8	11	51	6800	7091	48.234
0.9	0.85	11	51	9300	9808	63.875
0.9	0.9	11	51	11800	12423	81.422
0.9	0.95	11	51	26800	28246	179.703
0.9	0.99	11	49	117900	123690	784.266
0.99	0.5	11	51	5600	5788	37.86
0.99	0.55	11	51	5500	5698	39.078
0.99	0.6	11	51	6300	6567	44.813
0.99	0.65	11	51	5900	6085	39.844
0.99	0.7	11	51	7700	7978	54.016
0.99	0.75	11	51	7600	7910	53
0.99	0.8	11	51	12500	12898	84.547
0.99	0.85	11	51	16500	17070	111.188
0.99	0.9	11	51	25900	26596	173.984
0.99	0.95	11	50	43200	44667	290.484
0.99	0.99	11	51	243300	250199	1615.61
Program `paradd8.i` on machine configuration `large_multi_bus.md` with **maximally–bad** initialization.						
0.05	0.5	7	19	1900	2180	14.484
0.05	0.55	7	19	2400	2747	17.844
0.05	0.6	7	19	1900	2200	14.938
0.05	0.65	7	19	2400	2714	17.516
0.05	0.7	7	19	2300	2623	17.359
0.05	0.75	7	19	2500	2847	18.859
0.05	0.8	7	19	2300	2622	17.797
0.05	0.85	7	19	2800	3151	20.36
0.05	0.9	7	19	4000	4426	27.218
0.05	0.95	7	19	4200	4681	30.219
0.05	0.99	7	19	13700	14960	91.344
0.1	0.5	7	19	1400	1549	10.047
0.1	0.55	7	19	1700	1866	11.703
0.1	0.6	7	19	2900	3243	21.047
0.1	0.65	7	19	2400	2641	16.735
0.1	0.7	7	19	2800	3105	20.671
0.1	0.75	7	19	2800	3102	20.937
0.1	0.8	7	19	2600	2888	19.046
0.1	0.85	7	19	4200	4633	30.859

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.1	0.9	7	19	3900	4316	29.203
0.1	0.95	7	19	4600	5090	34.687
0.1	0.99	7	19	21800	23595	149.281
0.2	0.5	7	19	3000	3443	26.797
0.2	0.55	7	19	2900	3330	24.812
0.2	0.6	7	19	3700	4224	31.844
0.2	0.65	7	19	4100	4570	31.203
0.2	0.7	7	19	4100	4694	35.39
0.2	0.75	7	19	4100	4600	32.391
0.2	0.8	7	19	5000	5667	43.266
0.2	0.85	7	19	4600	5193	39.937
0.2	0.9	7	19	6000	6734	50.938
0.2	0.95	7	19	10400	11518	80.234
0.2	0.99	7	19	44700	49946	385.313
0.4	0.5	7	19	2600	2856	20.578
0.4	0.55	7	19	2600	2841	20.469
0.4	0.6	7	19	2800	3102	23.719
0.4	0.65	7	19	4500	4942	35.969
0.4	0.7	7	19	4000	4455	34.344
0.4	0.75	7	19	6200	7023	57.062
0.4	0.8	7	19	6100	6861	61.703
0.4	0.85	7	19	5900	6586	54
0.4	0.9	7	19	9700	10915	94.594
0.4	0.95	7	19	18900	21104	177.547
0.4	0.99	7	19	74400	83396	748.234
0.6	0.5	7	19	3300	3705	29.875
0.6	0.55	7	19	3500	3881	31.25
0.6	0.6	7	19	5000	5476	44.016
0.6	0.65	7	19	4600	5050	40.969
0.6	0.7	7	19	4700	5009	34.203
0.6	0.75	7	19	7200	7926	64.297
0.6	0.8	7	19	5400	5894	46.563
0.6	0.85	7	19	10800	11791	98.656
0.6	0.9	7	19	14400	16093	139.532
0.6	0.95	7	19	24700	27360	222.422
0.6	0.99	7	19	115600	128725	1141.95
0.8	0.5	7	19	5100	5517	49.156
0.8	0.55	7	19	6200	6726	59.515
0.8	0.6	7	19	7500	8200	75.594
0.8	0.65	7	19	6100	6604	56.281
0.8	0.7	7	19	7700	8378	72.562
0.8	0.75	7	19	10900	12047	111.36
0.8	0.8	7	19	9200	10090	95.046
0.8	0.85	7	19	9200	10015	94.843

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.8	0.9	7	19	16000	17475	167.625
0.8	0.95	7	19	42500	46375	447.532
0.8	0.99	7	19	177000	193317	1903.06
0.9	0.5	7	19	5700	6226	53.437
0.9	0.55	7	19	8300	9027	84.328
0.9	0.6	7	19	6300	6762	61.969
0.9	0.65	7	19	5300	5705	49.329
0.9	0.7	7	19	8700	9373	87.156
0.9	0.75	7	19	9100	9862	93.469
0.9	0.8	7	19	10100	10923	111.812
0.9	0.85	7	19	15000	16433	160.359
0.9	0.9	7	19	24000	26020	258.313
0.9	0.95	7	19	52600	56643	589.11
0.9	0.99	7	19	240600	259254	2730.47
0.99	0.5	7	19	8100	8468	76.843
0.99	0.55	7	19	6300	6571	61.016
0.99	0.6	7	19	10000	10449	105.015
0.99	0.65	7	19	10700	11231	106.203
0.99	0.7	7	19	15000	15970	155.422
0.99	0.75	7	19	15900	16597	160.469
0.99	0.8	7	19	18700	19700	183.86
0.99	0.85	7	19	21000	21959	205.797
0.99	0.9	7	19	37100	38809	370.172
0.99	0.95	7	19	75600	79587	790.094
0.99	0.99	7	19	398500	417893	4124.72
Program `paradd8.i` on machine configuration `large_multi_bus.md` with **list–scheduler** initialization.						
0.05	0.5	7	19	1300	1399	23.422
0.05	0.55	7	19	1300	1399	23.313
0.05	0.6	7	19	1300	1399	23.813
0.05	0.65	7	19	1300	1399	23.547
0.05	0.7	7	19	1300	1399	23.719
0.05	0.75	7	19	1300	1399	23.859
0.05	0.8	7	19	1300	1399	23.703
0.05	0.85	7	19	1300	1399	23.641
0.05	0.9	7	19	1500	1610	25.454
0.05	0.95	7	19	1900	2046	29.391
0.05	0.99	7	19	1800	1941	28.531
0.1	0.5	7	19	1200	1291	21.89
0.1	0.55	7	19	1200	1291	22.75
0.1	0.6	7	19	1200	1291	21.907
0.1	0.65	7	19	1200	1291	22.547
0.1	0.7	7	19	1200	1291	21.672
0.1	0.75	7	19	1200	1291	22.844

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.1	0.8	7	19	1500	1605	24.593
0.1	0.85	7	19	1500	1605	25.469
0.1	0.9	7	19	1600	1723	25.531
0.1	0.95	7	19	1800	1938	28.422
0.1	0.99	7	19	2800	3040	39.281
0.2	0.5	7	19	1800	1968	29.375
0.2	0.55	7	19	1800	1968	29.765
0.2	0.6	7	19	1800	1968	29.157
0.2	0.65	7	19	1600	1755	27.75
0.2	0.7	7	19	1600	1755	27
0.2	0.75	7	19	1900	2073	30.329
0.2	0.8	7	19	3000	3195	40.36
0.2	0.85	7	19	2700	2875	37.484
0.2	0.9	7	19	3100	3373	44.578
0.2	0.95	7	19	4400	4779	58.015
0.2	0.99	7	19	18600	20422	215.531
0.4	0.5	7	19	1900	2082	30.141
0.4	0.55	7	19	1900	2085	30.922
0.4	0.6	7	19	2000	2171	31.391
0.4	0.65	7	19	2200	2393	34.172
0.4	0.7	7	19	2600	2840	38.828
0.4	0.75	7	19	2600	2804	37.219
0.4	0.8	7	19	3300	3572	44.828
0.4	0.85	7	19	3700	4018	50.25
0.4	0.9	7	19	3400	3747	49.594
0.4	0.95	7	19	5600	6146	74.843
0.4	0.99	7	19	28300	31362	342.375
0.6	0.5	7	19	1800	1964	29.625
0.6	0.55	7	19	1800	1964	30.891
0.6	0.6	7	19	2100	2287	32.422
0.6	0.65	7	19	2400	2641	38.297
0.6	0.7	7	19	4100	4540	61.078
0.6	0.75	7	19	3200	3517	47.718
0.6	0.8	7	19	4800	5336	72.219
0.6	0.85	7	19	7200	8127	107.906
0.6	0.9	7	19	8900	9870	124.485
0.6	0.95	7	19	12900	14339	173.735
0.6	0.99	7	19	62300	70046	835.578
0.8	0.5	7	19	3500	3818	52.688
0.8	0.55	7	19	3900	4316	61.609
0.8	0.6	7	19	4700	5188	72.188
0.8	0.65	7	19	3600	3960	57.75
0.8	0.7	7	19	5200	5653	73.375
0.8	0.75	7	19	5500	6088	82.031

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.8	0.8	7	19	6300	7004	94.172
0.8	0.85	7	19	10400	11458	141.032
0.8	0.9	7	19	17100	19103	256.343
0.8	0.95	7	19	24900	27450	342.328
0.8	0.99	7	19	122700	136947	1782.05
0.9	0.5	7	19	3000	3289	48.594
0.9	0.55	7	19	4300	4673	65.828
0.9	0.6	7	19	4300	4696	67.796
0.9	0.65	7	19	6900	7621	105.234
0.9	0.7	7	19	5100	5474	78.859
0.9	0.75	7	19	11500	12804	169.297
0.9	0.8	7	19	11300	12322	165.578
0.9	0.85	7	19	13000	14098	189.281
0.9	0.9	7	19	20600	22502	288.906
0.9	0.95	7	19	37100	40960	541.125
0.9	0.99	7	19	164700	180951	2389.44
0.99	0.5	7	19	5900	6259	95.672
0.99	0.55	7	19	9000	9553	138.328
0.99	0.6	7	19	11300	11913	167.766
0.99	0.65	7	19	10900	11714	165.937
0.99	0.7	7	19	12000	12757	176.844
0.99	0.75	7	19	13700	14443	202.422
0.99	0.8	7	19	18000	19024	271.265
0.99	0.85	7	19	22800	24253	344.609
0.99	0.9	7	19	36400	38925	542.218
0.99	0.95	7	19	61800	64795	900.953
0.99	0.99	7	19	324900	343512	4755.63
Program `paradd16.i` on machine configuration `small_single_bus.md` with **maximally–bad** initialization.						
0.05	0.5	22	232	2100	2342	24.685
0.05	0.55	22	233	1700	1891	19.578
0.05	0.6	21	221	2100	2323	21.301
0.05	0.65	21	223	2400	2630	23.995
0.05	0.7	22	231	2000	2201	21.651
0.05	0.75	22	244	2400	2665	29.833
0.05	0.8	24	247	2300	2576	25.617
0.05	0.85	22	233	2700	2955	28.501
0.05	0.9	21	222	3500	3738	31.085
0.05	0.95	21	229	5000	5297	50.092
0.05	0.99	20	219	7900	8292	68.728
0.1	0.5	23	243	2000	2238	23.103
0.1	0.55	22	231	2600	2798	29.713
0.1	0.6	22	233	1800	2003	18.827
0.1	0.65	21	230	2400	2630	29.072

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.1	0.7	22	233	2500	2711	23.885
0.1	0.75	21	229	2400	2632	26.819
0.1	0.8	21	229	2800	3052	30.514
0.1	0.85	21	230	3300	3564	32.116
0.1	0.9	20	219	3900	4161	36.753
0.1	0.95	23	234	7400	7911	79.575
0.1	0.99	21	229	23600	24516	238.603
0.20	0.5	21	231	2900	3145	28.181
0.20	0.55	22	231	2600	2856	30.023
0.20	0.6	22	231	2500	2781	28.811
0.20	0.65	22	241	2400	2613	25.998
0.20	0.7	21	229	2700	2918	28.341
0.20	0.75	22	241	2900	3134	28.851
0.20	0.8	22	231	2800	2992	29.312
0.20	0.85	21	229	3800	4061	45.976
0.20	0.9	22	231	4300	4601	41.45
0.20	0.95	22	231	7900	8333	73.967
0.20	0.99	21	230	21100	22026	209.441
0.40	0.5	21	230	2600	2790	29.543
0.40	0.55	21	229	2600	2856	29.913
0.40	0.6	21	229	3000	3219	29.853
0.40	0.65	22	241	2800	2996	32.717
0.40	0.7	21	229	3300	3498	34.47
0.40	0.75	24	237	3100	3369	34.991
0.40	0.8	21	229	3900	4146	44.073
0.40	0.85	22	233	5000	5326	52.365
0.40	0.9	22	232	6900	7325	65.955
0.40	0.95	21	231	9700	10124	115.126
0.40	0.99	23	226	37200	38604	384.643
0.60	0.5	22	234	3000	3200	34.019
0.60	0.55	22	234	3100	3317	30.964
0.60	0.6	22	231	3200	3407	37.585
0.60	0.65	21	221	3400	3601	35.771
0.60	0.7	21	230	3300	3522	37.924
0.60	0.75	22	234	4600	4855	49.221
0.60	0.8	22	242	4300	4512	47.468
0.60	0.85	21	230	5900	6262	66.706
0.60	0.9	21	229	8000	8332	77.612
0.60	0.95	21	222	12200	12699	117.379
0.60	0.99	21	226	41300	42802	454.364
0.80	0.5	23	234	3300	3475	37.093
0.80	0.55	22	231	3500	3648	46.447
0.80	0.6	21	231	3600	3818	44.494
0.80	0.65	22	231	3800	3989	45.756

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.80	0.7	23	234	4500	4658	47.177
0.80	0.75	22	231	4100	4292	42.781
0.80	0.8	20	219	6100	6408	63.201
0.80	0.85	21	221	7100	7347	72.344
0.80	0.9	21	229	11300	11704	122.055
0.80	0.95	22	232	17400	18136	188.441
0.80	0.99	22	222	69000	71047	780.672
0.90	0.5	22	234	3200	3377	39.396
0.90	0.55	21	230	4000	4206	44.624
0.90	0.6	22	233	4000	4214	44.564
0.90	0.65	21	229	4000	4215	46.938
0.90	0.7	23	234	4700	4897	56.461
0.90	0.75	21	229	7100	7334	78.633
0.90	0.8	21	230	6400	6700	71.523
0.90	0.85	21	229	8300	8641	89.96
0.90	0.9	21	230	12900	13357	152.78
0.90	0.95	21	231	23100	23823	250.741
0.90	0.99	21	229	99700	102545	1064.52
0.99	0.5	22	231	5000	5148	60.638
0.99	0.55	21	229	6500	6668	81.918
0.99	0.6	24	237	5800	6025	66.015
0.99	0.65	20	219	6600	6838	77.021
0.99	0.7	22	231	6900	7082	80.196
0.99	0.75	21	231	7800	7985	92.533
0.99	0.8	21	231	10600	10824	117.919
0.99	0.85	21	231	14300	14658	159.8
0.99	0.9	21	229	22400	22846	275.386
0.99	0.95	21	221	39000	39638	440.804
0.99	0.99	22	222	178900	181595	2030.11
Program `paradd16.i` on machine configuration `small_single_bus.md` with **list–scheduler** initialization.						
0.05	0.5	22	229	800	843	25.812
0.05	0.55	22	229	800	843	25.828
0.05	0.6	22	229	800	843	25.782
0.05	0.65	22	229	800	843	25.797
0.05	0.7	22	229	800	843	25.829
0.05	0.75	22	229	1000	1053	29
0.05	0.8	22	229	1000	1053	28.703
0.05	0.85	22	229	1000	1053	28.719
0.05	0.9	22	229	1000	1053	28.719
0.05	0.95	22	229	1400	1482	35.375
0.05	0.99	22	229	1800	1900	41.312
0.1	0.5	22	229	800	843	25.829
0.1	0.55	22	229	800	843	25.875

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.1	0.6	22	229	800	843	25.766
0.1	0.65	22	229	1000	1053	28.954
0.1	0.7	22	229	1000	1053	28.906
0.1	0.75	22	229	1000	1053	29
0.1	0.8	22	229	1000	1053	28.656
0.1	0.85	22	229	1400	1484	35.859
0.1	0.9	22	229	1400	1481	35.688
0.1	0.95	22	229	1800	1898	41.922
0.1	0.99	22	229	3200	3362	62.75
0.2	0.5	22	229	1000	1049	28.765
0.2	0.55	22	229	1200	1260	31.859
0.2	0.6	22	229	1200	1260	31.859
0.2	0.65	22	229	1200	1260	31.859
0.2	0.7	22	229	1400	1463	34.985
0.2	0.75	22	229	1600	1686	38.438
0.2	0.8	22	229	1600	1681	35.351
0.2	0.85	22	229	1800	1885	37.754
0.2	0.9	22	229	2200	2309	43.383
0.2	0.95	22	229	3400	3557	59.195
0.2	0.99	22	229	6800	7138	104.089
0.4	0.5	22	229	1200	1259	31.782
0.4	0.55	22	229	1400	1462	34.906
0.4	0.6	22	229	1400	1460	34.843
0.4	0.65	22	229	1600	1682	38.156
0.4	0.7	22	229	1600	1680	37.829
0.4	0.75	22	229	1600	1680	37.859
0.4	0.8	22	229	2000	2093	40.688
0.4	0.85	22	229	2400	2532	46.427
0.4	0.9	22	229	3400	3557	58.594
0.4	0.95	22	229	4600	4797	74.688
0.4	0.99	22	225	9800	10326	141.423
0.6	0.5	22	229	1600	1705	39.078
0.6	0.55	22	229	1600	1718	38.266
0.6	0.6	22	229	1600	1708	37.907
0.6	0.65	22	229	1600	1703	38.141
0.6	0.7	22	229	2000	2110	44.469
0.6	0.75	22	229	2000	2117	44.641
0.6	0.8	22	229	2800	2973	52.135
0.6	0.85	22	229	3400	3606	59.225
0.6	0.9	21	222	4800	5074	81.857
0.6	0.95	20	219	7800	8061	112.181
0.6	0.99	21	222	18400	19123	262.127
0.8	0.5	22	229	1400	1435	33.25
0.8	0.55	22	229	1800	1859	40.015

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.8	0.6	22	229	1800	1859	40.547
0.8	0.65	22	229	2100	2179	46.281
0.8	0.7	22	229	2900	3011	57.157
0.8	0.75	22	229	3300	3456	63.343
0.8	0.8	21	221	3500	3619	60.527
0.8	0.85	22	229	4800	5003	79.184
0.8	0.9	21	222	5300	5493	81.728
0.8	0.95	22	223	11400	11822	157.376
0.8	0.99	21	221	49000	50821	622.334
0.9	0.5	22	229	2000	2078	43.125
0.9	0.55	22	229	2200	2310	47.375
0.9	0.6	21	222	2200	2277	47.062
0.9	0.65	22	229	3000	3120	57.985
0.9	0.7	22	229	3100	3203	61.188
0.9	0.75	22	229	3200	3274	62.062
0.9	0.8	22	229	4700	4921	80.415
0.9	0.85	22	229	5900	6158	87.366
0.9	0.9	22	229	8400	8837	137.458
0.9	0.95	20	220	14800	15236	201.01
0.9	0.99	23	227	67800	70338	909.007
0.99	0.5	22	229	4000	4154	72.734
0.99	0.55	22	225	4000	4118	70.125
0.99	0.6	22	229	4100	4263	76.125
0.99	0.65	22	229	5300	5450	90.672
0.99	0.7	22	229	6100	6272	106
0.99	0.75	22	229	8000	8231	132.656
0.99	0.8	22	229	9800	10091	150.457
0.99	0.85	21	221	10300	10465	140.242
0.99	0.9	22	229	16600	17026	240.015
0.99	0.95	22	229	33300	34071	459.861
0.99	0.99	21	221	141800	144682	2048.96
Program `paradd16.i` on machine configuration `large_multi_bus.md` with **maximally–bad** initialization.						
0.05	0.5	11	64	2700	3054	43.765
0.05	0.55	11	64	2900	3266	48.453
0.05	0.6	11	64	3200	3640	52.828
0.05	0.65	11	64	2800	3197	48.109
0.05	0.7	11	64	2900	3267	45.985
0.05	0.75	11	64	3500	3904	51.782
0.05	0.8	11	64	3400	3848	55.063
0.05	0.85	11	64	4000	4399	57.641
0.05	0.9	11	64	5300	5828	74.015
0.05	0.95	11	64	6200	6733	82.312
0.05	0.99	11	64	25100	26669	279.234

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.1	0.5	11	64	3500	3985	61.031
0.1	0.55	11	64	2900	3313	52.015
0.1	0.6	11	64	2800	3198	51.109
0.1	0.65	11	64	3200	3639	56.047
0.1	0.7	11	64	3500	3921	56.703
0.1	0.75	11	64	3700	4165	60.375
0.1	0.8	11	64	4600	5096	69.704
0.1	0.85	11	64	4900	5480	81.641
0.1	0.9	11	64	5000	5381	69.343
0.1	0.95	11	64	8100	8853	123.859
0.1	0.99	11	64	27600	29660	358.906
0.2	0.5	11	64	3100	3497	55.531
0.2	0.55	11	64	3200	3602	58.25
0.2	0.6	11	64	4200	4709	73.594
0.2	0.65	11	64	4000	4521	71.656
0.2	0.7	11	64	4000	4458	65.235
0.2	0.75	11	64	4400	4908	76.766
0.2	0.8	11	64	4400	4835	62.049
0.2	0.85	11	64	5600	6118	74.367
0.2	0.9	11	64	6800	7441	92.783
0.2	0.95	11	64	12500	13513	151.127
0.2	0.99	11	64	43000	46186	538.364
0.4	0.5	11	64	3700	4130	62.266
0.4	0.55	11	64	3500	3947	62.203
0.4	0.6	11	64	3600	4004	60.875
0.4	0.65	11	64	3800	4201	62.484
0.4	0.7	11	64	4000	4495	74
0.4	0.75	11	64	4600	5055	75.219
0.4	0.8	11	64	6000	6570	85.733
0.4	0.85	11	64	6500	7016	92.734
0.4	0.9	11	64	8300	8918	113.072
0.4	0.95	11	64	13300	14075	182.492
0.4	0.99	11	64	64500	68756	850.372
0.6	0.5	11	64	4200	4554	74.328
0.6	0.55	11	64	4100	4422	69.328
0.6	0.6	11	64	4000	4335	71.25
0.6	0.65	11	64	4500	4838	81.797
0.6	0.7	11	64	4700	5092	82.797
0.6	0.75	11	64	5900	6337	97.406
0.6	0.8	11	64	6200	6690	95.187
0.6	0.85	11	64	7300	7811	105.062
0.6	0.9	11	64	9800	10396	143.807
0.6	0.95	11	64	19100	20236	278.2
0.6	0.99	11	64	81200	85916	1225.78

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.8	0.5	11	64	5200	5571	90.578
0.8	0.55	11	64	4100	4385	73.313
0.8	0.6	11	64	5300	5709	101.766
0.8	0.65	11	64	5000	5445	93.187
0.8	0.7	11	64	5300	5693	94.391
0.8	0.75	11	64	6800	7236	126.219
0.8	0.8	11	64	8200	8684	129.015
0.8	0.85	11	64	9800	10365	162.674
0.8	0.9	11	64	14100	14856	229.82
0.8	0.95	11	64	27000	28312	421.957
0.8	0.99	11	64	119500	124765	1888.69
0.9	0.5	11	64	6200	6608	119.75
0.9	0.55	11	64	6000	6352	117.813
0.9	0.6	11	64	7200	7693	142.562
0.9	0.65	11	64	6600	6914	121.282
0.9	0.7	11	64	8800	9293	162.485
0.9	0.75	11	64	7600	8051	151.516
0.9	0.8	11	64	8400	8869	143.185
0.9	0.85	11	64	11500	12014	205.235
0.9	0.9	11	64	17200	17864	293.492
0.9	0.95	11	64	37500	39100	641.503
0.9	0.99	11	64	152700	158737	2633.36
0.99	0.5	11	64	7300	7576	149.735
0.99	0.55	11	64	7900	8199	169.578
0.99	0.6	11	64	7800	8160	163.063
0.99	0.65	11	64	7800	8060	154.156
0.99	0.7	11	64	10800	11176	228
0.99	0.75	11	64	11800	12256	250.672
0.99	0.8	11	64	15100	15655	276.508
0.99	0.85	11	64	18300	18778	324.937
0.99	0.9	11	64	26700	27499	497.676
0.99	0.95	11	64	53300	54908	980.53
0.99	0.99	11	64	225700	230957	4512
Program `paradd16.i` on machine configuration `large_multi_bus.md` with **list–scheduler** initialization.						
0.05	0.5	13	86	1000	1144	73.109
0.05	0.55	13	86	1000	1144	73.047
0.05	0.6	13	86	1000	1144	73.078
0.05	0.65	13	86	1000	1144	73.016
0.05	0.7	13	86	1000	1144	78.266
0.05	0.75	13	86	1200	1367	78.032
0.05	0.8	13	86	1400	1602	83.156
0.05	0.85	13	86	1400	1599	83.125
0.05	0.9	13	86	1800	2074	93.859

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.05	0.95	13	86	2400	2732	109.719
0.05	0.99	11	64	3000	3318	114.203
0.1	0.5	13	86	1000	1134	73.047
0.1	0.55	13	86	1000	1134	72.906
0.1	0.6	13	86	1200	1372	78.266
0.1	0.65	13	86	1200	1369	78.156
0.1	0.7	13	86	1400	1589	83.062
0.1	0.75	13	86	1400	1589	83.015
0.1	0.8	13	86	1400	1588	83.156
0.1	0.85	13	86	1600	1822	88.578
0.1	0.9	13	86	2200	2488	103.75
0.1	0.95	11	64	3500	3815	119.609
0.1	0.99	11	64	6300	6813	180.391
0.2	0.5	13	86	1200	1366	78.422
0.2	0.55	13	86	1200	1366	78.235
0.2	0.6	13	86	1400	1587	83.235
0.2	0.65	13	86	1400	1588	83.063
0.2	0.7	13	86	1400	1588	83.078
0.2	0.75	13	86	1600	1815	87.765
0.2	0.8	13	86	2200	2478	95.798
0.2	0.85	11	64	2900	3186	100.104
0.2	0.9	11	64	2700	2926	95.016
0.2	0.95	11	64	5000	5378	127.304
0.2	0.99	11	64	7400	7815	159.519
0.4	0.5	11	64	2000	2163	89.719
0.4	0.55	11	64	2000	2161	89
0.4	0.6	11	64	1700	1852	84.218
0.4	0.65	11	64	2100	2270	91
0.4	0.7	11	64	2300	2502	95.765
0.4	0.75	11	64	2200	2395	95.047
0.4	0.8	11	64	2800	3035	93.234
0.4	0.85	11	64	3200	3419	100.784
0.4	0.9	11	64	4200	4444	115.546
0.4	0.95	11	64	5300	5634	135.094
0.4	0.99	11	64	15000	15988	292.701
0.6	0.5	11	64	2500	2771	103.828
0.6	0.55	11	64	2500	2731	101.25
0.6	0.6	11	64	2200	2417	96.312
0.6	0.65	11	64	2600	2801	102.485
0.6	0.7	11	64	2500	2694	102.265
0.6	0.75	11	64	3000	3281	115.218
0.6	0.8	11	64	2900	3145	98.251
0.6	0.85	11	64	4500	4888	130.367
0.6	0.9	11	64	5400	5751	142.885

p	α	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.6	0.95	11	64	9200	9829	215.701
0.6	0.99	11	64	42300	45213	812.708
0.8	0.5	11	64	3400	3707	131.797
0.8	0.55	11	64	3200	3473	126.938
0.8	0.6	11	64	3900	4192	141.359
0.8	0.65	11	64	3700	3935	131.109
0.8	0.7	11	64	4200	4502	142.718
0.8	0.75	11	64	5000	5367	171.516
0.8	0.8	11	64	4700	5035	142.935
0.8	0.85	11	64	6100	6554	171.156
0.8	0.9	11	64	9700	10285	238.823
0.8	0.95	11	64	18300	19478	409.259
0.8	0.99	11	64	64600	68368	1394.77
0.9	0.5	11	64	3800	4082	137.422
0.9	0.55	11	64	3300	3528	125.703
0.9	0.6	11	64	4000	4243	138.719
0.9	0.65	11	64	4000	4291	146.344
0.9	0.7	11	64	4700	5009	158.281
0.9	0.75	11	64	5500	5762	175.422
0.9	0.8	11	64	5700	6002	160.731
0.9	0.85	11	64	6700	7127	183.724
0.9	0.9	11	64	11400	12192	291.569
0.9	0.95	11	64	19800	20804	457.788
0.9	0.99	11	64	93400	98353	2101.05
0.99	0.5	11	64	4600	4814	161.672
0.99	0.55	11	64	5400	5717	189.187
0.99	0.6	11	64	5600	5947	187.094
0.99	0.65	11	64	6600	6963	217.906
0.99	0.7	11	64	6200	6455	203.516
0.99	0.75	11	64	8800	9198	273.594
0.99	0.8	11	64	11600	12057	333.38
0.99	0.85	11	64	14600	15161	378.844
0.99	0.9	11	64	20200	20877	488.232
0.99	0.95	11	64	41700	43302	1034.2
0.99	0.99	11	64	195600	201369	4693.36

E.2 Aggregate Move Experiments

move fraction	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
Program `paradd8.i` on machine configuration `small_single_bus.md`.					
0	11	51	26900	28318	135.435
0.2	11	49	30800	32476	194.67
0.4	11	49	30600	31811	231.743
0.6	11	49	34300	35400	303.436
0.8	11	49	34900	35824	342.663
1	11	49	28600	29466	300.452
1.2	11	49	30000	30741	354.47
1.4	12	51	29100	29625	375.73
1.6	11	49	26700	27137	355.261
1.8	11	49	27900	28316	361.28
2	11	49	30200	30703	421.456
Program `paradd8.i` on machine configuration `large_multi_bus.md`.					
0	7	19	42500	46375	378.083
0.2	7	19	51100	56905	640.621
0.4	7	19	46300	53378	704.804
0.6	7	19	57300	64827	962.834
0.8	7	19	60800	69549	1048.09
1	7	19	57300	67050	1197.15
1.2	7	19	66600	77564	1468.05
1.4	7	19	66600	78568	1474.75
1.6	7	19	59000	70150	1439.79
1.8	7	19	57600	68926	1358.77
2	7	19	61000	71796	1481.02
Program `paradd16.i` on machine configuration `small_single_bus.md`.					
0	21	223	17300	18036	184.795
0.2	23	233	21000	21409	402.088
0.4	22	231	17800	17986	412.934
0.6	24	233	17100	17239	480.18
0.8	26	243	15400	15522	502.122
1	28	261	15700	15796	592.392
1.2	28	254	15200	15276	653.429
1.4	29	248	17200	17266	722.429
1.6	27	242	14000	14063	631.157
1.8	36	279	11400	11417	562.99
2	30	281	12600	12618	691.093
Program `paradd16.i` on machine configuration `large_multi_bus.md`.					
0	11	64	27000	28312	441.575
0.2	11	64	42500	45252	1131.89
0.4	11	64	45700	49276	1520.15
0.6	11	64	41100	44899	1469.67

move fraction	schedule length	minimum energy	accepted reconfigs	total reconfigs	clock time
0.8	11	64	46900	51248	2030.18
1	11	64	45300	50023	2094.74
1.2	11	64	46200	51823	2275.97
1.4	11	64	43700	48768	2383.33
1.6	11	64	54600	61040	3197.42
1.8	11	64	47200	53231	2815.34
2	11	64	48700	54865	3243.3

E.3 Pass Node Experiments

R prob.	S prob.	sched. length	min. energy	broken edges	pass nodes	accepted reconfigs	total reconfigs	clock time
Program `paradd8.i` on machine configuration `cluster_with_move.md` with **maximally–bad** initialization.								
0.1	0.1	15	64	0	1	124500	152598	1780.81
0.1	0.3	8	28	1	2	117900	151066	2655.73
0.1	0.5	10	48	1	32	90500	121802	4534.59
0.1	0.7	20	106	2	3	50900	67233	5036.05
0.1	0.9	27	129	5	1	39000	57746	6619.19
0.3	0.1	11	50	0	1	143700	175197	1718.03
0.3	0.3	18	76	0	6	115900	139523	1922.25
0.3	0.5	11	47	0	18	94000	124686	4021.81
0.3	0.7	35	315	1	75	55100	72374	3584.84
0.3	0.9	26	147	6	4	36900	49147	3166.31
0.5	0.1	16	79	0	4	114800	126305	1131.75
0.5	0.3	11	42	0	3	121400	142499	1753.48
0.5	0.5	15	89	0	24	83100	104151	2497.39
0.5	0.7	28	249	1	40	58000	69926	1871.92
0.5	0.9	39	1187	0	141	48200	58357	2838.05
0.7	0.1	14	54	0	3	126200	134480	907.36
0.7	0.3	9	32	1	2	99200	106142	840.891
0.7	0.5	32	381	0	52	49500	55087	824.516
0.7	0.7	33	375	0	63	61700	69465	1336.36
0.7	0.9	42	452	0	63	57000	64941	1427.67
0.9	0.1	11	39	1	0	89700	90886	466.922
0.9	0.3	18	112	0	10	62000	63503	373.687
0.9	0.5	26	262	2	32	30800	31575	248.516
0.9	0.7	28	210	5	14	24400	25028	251.829
0.9	0.9	26	199	5	18	20800	21228	202.203
Program `paradd8.i` on machine configuration `cluster_with_move.md` with **list–scheduler** initialization.								
0.1	0.1	11	51	0	0	91600	115070	1879.23
0.1	0.3	11	51	0	0	82800	111779	2010.23
0.1	0.5	11	51	0	0	98900	132443	4472.92
0.1	0.7	11	51	0	0	40000	48976	2543.73
0.1	0.9	11	51	0	0	33900	42651	2831.97
0.3	0.1	11	49	0	3	110000	138577	2021.92
0.3	0.3	11	51	0	0	84200	105628	1708.94
0.3	0.5	11	51	0	0	72400	99872	3037.66
0.3	0.7	11	51	0	0	54100	61834	1936.06
0.3	0.9	11	51	0	0	36300	43312	1944.89
0.5	0.1	11	49	0	3	111400	130123	1647.52
0.5	0.3	11	51	0	0	77600	89697	1282.91
0.5	0.5	11	51	0	0	64700	73479	1398.58

R prob.	S prob.	sched. length	min. energy	broken edges	pass nodes	accepted reconfigs	total reconfigs	clock time
0.5	0.7	11	51	0	0	54500	60885	1533.95
0.5	0.9	11	51	0	0	38600	42704	1203.45
0.7	0.1	11	51	0	0	81100	86332	957.907
0.7	0.3	11	51	0	0	87500	96984	1195.13
0.7	0.5	11	51	0	0	54300	58660	915.203
0.7	0.7	11	51	0	0	42200	44933	759.641
0.7	0.9	11	51	0	0	38700	41220	786.407
0.9	0.1	11	51	0	0	62700	63936	594.672
0.9	0.3	11	51	0	0	44300	45207	438.703
0.9	0.5	11	51	0	0	29000	29561	323.156
0.9	0.7	11	51	0	0	26800	27174	309.406
0.9	0.9	11	51	0	0	23800	24026	275.031

Program `paradd8.i` on machine configuration `cluster_without_move.md` with **maximally–bad** initialization.

0.1	0.1	8	28	0	0	200800	241020	2597.63
0.1	0.3	9	31	0	3	107900	147537	2766.5
0.1	0.5	12	60	0	44	134600	160890	5259.03
0.1	0.7	29	116	2	4	52300	70250	7325.42
0.1	0.9	28	133	1	6	42700	54433	7179.45
0.3	0.1	8	28	0	0	195300	223238	2197.05
0.3	0.3	13	69	0	7	127600	157227	2341.28
0.3	0.5	13	68	0	28	119900	143935	4040.59
0.3	0.7	28	231	0	70	82500	101363	4699.44
0.3	0.9	25	369	0	167	75400	91456	5800.64
0.5	0.1	9	31	0	1	175900	192615	1413.19
0.5	0.3	9	33	0	5	181100	205153	2328.94
0.5	0.5	11	45	0	14	119000	132310	2338.56
0.5	0.7	23	184	0	45	82500	94396	2709.38
0.5	0.9	31	444	0	99	63500	74160	3209.78
0.7	0.1	10	36	0	2	162000	169476	1074.75
0.7	0.3	14	81	0	10	94800	103050	1083.33
0.7	0.5	34	264	0	41	79800	87522	1492.75
0.7	0.7	35	308	0	49	75100	81104	1400.05
0.7	0.9	32	527	0	80	65000	70077	1599.88
0.9	0.1	12	42	0	1	127500	129333	578.718
0.9	0.3	18	72	0	6	105100	106850	584.469
0.9	0.5	52	506	0	35	43300	44269	383.938
0.9	0.7	42	443	1	45	37500	38227	308.938
0.9	0.9	29	189	5	17	23700	24092	197.406

Program `paradd8.i` on machine configuration `cluster_without_move.md` with **list–scheduler** initialization.

0.1	0.1	8	28	0	0	146400	185129	2627.73
0.1	0.3	9	30	0	3	142500	183607	3018.05
0.1	0.5	9	36	0	0	79700	102325	3464.34

R prob.	S prob.	sched. length	min. energy	broken edges	pass nodes	accepted reconfigs	total reconfigs	clock time
0.1	0.7	9	36	0	0	61000	73100	3756.77
0.1	0.9	9	35	0	0	37300	45163	1827.11
0.3	0.1	8	28	0	0	123300	148195	1905.97
0.3	0.3	9	35	0	0	66700	78992	1120.2
0.3	0.5	9	35	0	0	87600	102394	2272.59
0.3	0.7	9	35	0	0	54900	62066	1714.33
0.3	0.9	9	32	0	0	32200	36909	949.922
0.5	0.1	9	30	0	1	133400	149738	1666.27
0.5	0.3	9	33	0	1	103600	119620	1492.5
0.5	0.5	10	48	0	0	64600	74815	1418.53
0.5	0.7	10	39	0	0	49000	53517	1183.06
0.5	0.9	9	35	0	0	43000	47027	1030.13
0.7	0.1	11	37	0	2	118300	126133	1205.06
0.7	0.3	9	33	0	4	93900	101751	1094.47
0.7	0.5	9	32	0	0	75000	80396	1034.28
0.7	0.7	10	39	0	0	70800	74769	1113.72
0.7	0.9	9	35	0	0	40200	42101	566.281
0.9	0.1	9	35	0	0	61800	63092	508.844
0.9	0.3	9	32	0	0	54100	55513	476.734
0.9	0.5	9	36	0	0	25500	25983	249.156
0.9	0.7	9	35	0	0	3700	3710	39.765
0.9	0.9	10	39	0	0	15900	16036	157.344
Program `paradd16.i` on machine configuration `cluster_with_move.md` with **maximally–bad** initialization.								
0.1	0.1	14	101	2	1	92200	107245	2354.24
0.1	0.3	18	161	0	13	100600	166282	6462.95
0.1	0.5	76	1142	0	126	74400	136120	15615.7
0.1	0.7	76	2896	1	302	43200	68386	15240.3
0.1	0.9	62	751	11	2	31800	50137	13326.8
0.3	0.1	18	147	1	2	110900	126181	2472.17
0.3	0.3	20	163	0	12	91000	130704	4239.57
0.3	0.5	69	1106	0	113	71900	105676	10352.8
0.3	0.7	67	738	10	4	43400	61037	7328.33
0.3	0.9	63	818	11	12	18600	22728	2643.86
0.5	0.1	16	141	1	1	87700	96183	1655.75
0.5	0.3	38	218	0	13	100500	123867	3085.71
0.5	0.5	91	2523	1	141	41000	50735	3506.41
0.5	0.7	74	2120	2	149	43200	54028	4030.41
0.5	0.9	58	715	13	11	39600	49169	4556.4
0.7	0.1	19	160	1	6	84100	89756	1175.4
0.7	0.3	50	494	0	19	80800	91459	1527.83
0.7	0.5	83	2013	1	132	47600	54384	2019.39
0.7	0.7	75	1967	5	118	38400	42778	1479.9
0.7	0.9	58	955	15	32	33300	37146	1211.91

R prob.	S prob.	sched. length	min. energy	broken edges	pass nodes	accepted reconfigs	total reconfigs	clock time
0.9	0.1	25	230	2	5	48000	48696	411.131
0.9	0.3	90	982	2	19	50300	51863	542.51
0.9	0.5	60	786	14	14	27000	27721	385.895
0.9	0.7	58	827	19	30	18600	19061	241.007
0.9	0.9	58	1042	18	40	17100	17544	214.339
Program **paradd16.i** on machine configuration **cluster_with_move.md** with **list–scheduler** initialization.								
0.1	0.1	22	229	0	0	79400	99590	2880.41
0.1	0.3	19	158	0	11	84000	146456	6379.08
0.1	0.5	22	229	0	0	42800	71419	6020.86
0.1	0.7	22	229	0	0	32700	47375	6538.22
0.1	0.9	22	229	0	0	26600	34105	4393.41
0.3	0.1	22	229	0	0	68200	77659	1950.03
0.3	0.3	16	155	0	9	88000	121612	4227.79
0.3	0.5	22	229	0	0	72300	103070	6628.21
0.3	0.7	22	229	0	0	34900	43118	2933.59
0.3	0.9	22	229	0	0	26000	31094	2439.3
0.5	0.1	22	229	0	0	70300	79163	1860.69
0.5	0.3	22	229	0	0	54300	66031	1840.31
0.5	0.5	22	229	0	0	61600	78777	4002.59
0.5	0.7	22	229	0	0	33500	39840	1887.35
0.5	0.9	22	229	0	0	32800	37894	1926.29
0.7	0.1	22	229	0	0	71500	75094	1347.19
0.7	0.3	22	229	0	0	57200	63470	1372.56
0.7	0.5	22	229	0	0	39500	43662	1224.21
0.7	0.7	22	229	0	0	36400	38799	1126.17
0.7	0.9	22	229	0	0	27400	29552	984.776
0.9	0.1	22	229	0	0	39000	39451	610.969
0.9	0.3	22	229	0	0	34400	35026	572.984
0.9	0.5	22	229	0	0	24300	24821	490.105
0.9	0.7	22	229	0	0	17400	17580	314.463
0.9	0.9	22	229	0	0	15600	15775	288.525
Program **paradd16.i** on machine configuration **cluster_without_move.md** with **maximally–bad** initialization.								
0.1	0.1	14	109	0	2	168000	197925	4747.22
0.1	0.3	23	173	0	13	99300	141054	5314.17
0.1	0.5	73	1633	1	162	63600	86446	9840.42
0.1	0.7	86	4529	0	397	63800	89383	25903.2
0.1	0.9	75	5779	0	493	54000	75801	24823.7
0.3	0.1	14	101	0	3	135500	156101	3316.59
0.3	0.3	28	245	0	15	100600	126611	4148.57
0.3	0.5	59	1465	0	124	76100	99710	11619.4
0.3	0.7	63	1648	0	182	69200	88152	10441.6
0.3	0.9	64	4074	0	316	62800	79702	15007.7

R prob.	S prob.	sched. length	min. energy	broken edges	pass nodes	accepted reconfigs	total reconfigs	clock time
0.5	0.1	15	95	0	2	132000	145325	2382.28
0.5	0.3	37	356	0	17	90600	105869	2446.48
0.5	0.5	51	1136	0	96	77300	91321	5928.42
0.5	0.7	69	2297	0	187	70900	84288	7608.74
0.5	0.9	62	2773	1	186	50000	58450	5406.31
0.7	0.1	17	117	0	5	125000	130894	1649.06
0.7	0.3	56	782	1	40	64800	71284	1329.74
0.7	0.5	70	1978	0	130	67200	73868	2907.77
0.7	0.7	63	2205	0	159	64500	70654	3340.99
0.7	0.9	72	2792	2	153	41700	45182	1608.66
0.9	0.1	29	245	1	8	76300	77285	553.646
0.9	0.3	30	350	1	17	69300	70532	601.004
0.9	0.5	73	1930	0	86	31700	32454	479.88
0.9	0.7	79	2214	3	100	32100	32744	396.62
0.9	0.9	77	2386	4	102	31500	32170	389.73
Program `paradd16.i` on machine configuration `cluster_without_move.md` with **list–scheduler** initialization.								
0.1	0.1	14	107	0	3	120500	150110	4312.78
0.1	0.3	15	128	0	0	75500	108206	4529.82
0.1	0.5	15	119	0	0	55600	84334	8539.75
0.1	0.7	15	118	0	0	37500	40904	2089.49
0.1	0.9	16	154	0	0	30600	33323	1742.43
0.3	0.1	14	109	0	2	105800	123074	3014.68
0.3	0.3	15	104	0	4	77700	98930	3358.51
0.3	0.5	16	156	0	0	64100	85411	6462.7
0.3	0.7	17	148	0	0	51300	56849	2633.9
0.3	0.9	15	130	0	0	35300	38732	1616.25
0.5	0.1	17	138	0	7	93700	106456	2232.03
0.5	0.3	15	124	0	0	71100	83262	2188.37
0.5	0.5	14	116	0	0	63300	74016	2967.13
0.5	0.7	15	124	0	0	47900	50602	1514.22
0.5	0.9	15	128	0	0	36500	38721	1238.41
0.7	0.1	15	127	0	0	61400	64589	1167.97
0.7	0.3	15	119	0	0	69700	76141	1494.4
0.7	0.5	16	134	0	0	59400	64701	1760.02
0.7	0.7	15	132	0	0	38800	39870	853.447
0.7	0.9	15	121	0	0	43100	44561	995.551
0.9	0.1	15	120	0	0	40300	40656	536.892
0.9	0.3	16	145	0	0	61200	62439	919.642
0.9	0.5	16	146	0	0	24400	24565	394.768
0.9	0.7	18	178	0	10	9000	9027	174.01
0.9	0.9	14	117	0	0	2500	2503	71.272

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison–Wesley, Reading, Massachusetts, 1986.

[2] Siamak Arya. An optimal instruction–scheduling model for a class of vector processors. *IEEE Transactions on Computers*, C–34(11):981–995, November 1985.

[3] Todd M. Austin and Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351, Gold Coast, Australia, May 1992.

[4] Michael Butler, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales, and Michael Shebanow. Single instruction stream parallelism is greater than two. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 276–286, Toronto, Canada, May 1991.

[5] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 292–300, Portland, Oregon, December 1992.

[6] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 180–192, Palo Alto, California, October 1987.

[7] Scott Davidson, David Landskov, Bruce D. Shriver, and Patrick W. Mallett. Some experiments in local microcode compaction for horizontal machines. *IEEE Transactions on Computers*, C–30(7):460–477, July 1981.

[8] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C–30(7):478–490, July 1981.

[9] Sadahiro Isoda, Yoshizumi Kobayashi, and Toru Ishida. Global compaction of horizontal microprograms based on the generalized data dependency graph. *IEEE Transactions on Computers*, C–32(10):922–933, October 1983.

[10] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the eighth Annual ACM Symposium on Principles of Programming Languages*, pages 207–218, Williamsburg, Virginia, January 1981.

[11] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, June 1988.

[12] Soo-Mook Moon and Kemal Ebcioğlu. An efficient resource–constrained global scheduling technique for superscalar and VLIW processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 55–71, Portland, Oregon, December 1992.

[13] Alexandru Nicolau. Percolation scheduling: A parallel compilation technique. Technical Report 85–678, Cornell University, Department of Computer Science, May 1985.

[14] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, Cambridge, England, 1988.

[15] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceed-*

ings of the 14th Annual Microprogramming Workshop, pages 183–198, Chatham, Massachusetts, October 1981.

[16] B. Ramakrishna Rau, Christopher D. Glaeser, and Raymond L. Picard. Efficient code generation for horizontal architectures: Compiler techniques and architectural support. In *Proceedings of the 9th Annual International Symposium on Computer Architecture*, pages 131–139, Austin, Texas, April 1982.

[17] Michael D. Smith, Mark Horowitz, and Monica S. Lam. Efficient superscalar performance through boosting. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248–259, Boston, Massachusetts, October 1992.

[18] Mark Smotherman, Sanjay Krishnamurthy, P. S. Aravind, and David Hunnicutt. Efficient DAG construction and heuristic calculation for instruction scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 93–102, Albuquerque, New Mexico, November 1991.

[19] Mario Tokoro, Eiji Tamura, and Takashi Takizuka. Optimization of microprograms. *IEEE Transactions on Computers*, C–30(7):491–504, July 1981.

[20] Andrew Wolfe and John P. Shen. A variable instruction stream extension to the VLIW architecture. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–14, Santa Clara, California, April 1991.