Optimizations for Acyclic Dataflow Graphs for Hardware-Software Codesign

by

Quaeed Motiwala

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

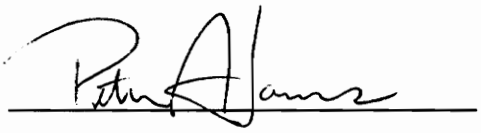
in partial fulfillment of the requirements for the degree of

Master of Science

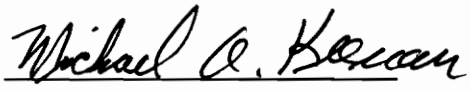
in

Electrical Engineering

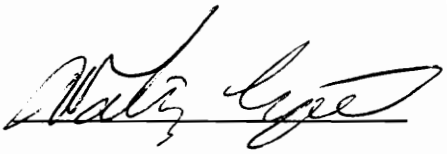
APPROVED:



Dr. Peter M. Athanas, Chairman



Dr. Michael A. Keenan



Dr. Walling R. Cyre

September, 1994

Blacksburg, Virginia

Optimizations for Acyclic Dataflow Graphs for Hardware-Software Codesign

by

Quaeed Motiwala

Committee Chairman: Dr. Peter Athanas

Electrical Engineering

(ABSTRACT)

Most computationally-intensive programs spend a majority of their time within a small portion of the executable code. A novel computer architecture and compiler, called PRISM-2, is introduced, which improves the performance of these programs by synthesizing the "most often" executed instructions. This is accomplished by augmenting a general-purpose processor with an array of FPGAs. The information regarding the structures to be synthesized is extracted from the high-level language (HLL) specification presented at the input of the PRISM-2 compiler. This behavorial specification is transformed into an internal, dataflow graph (DFG) format. Linear and loop optimizations are then performed to optimizae this representation. Linear optimizations are the main topic of discussion in this thesis. The optimizaed DFG is then synthesized on the reconfigurable platform.

Acknowledgments

I would like to thank Dr. Peter Athanas for the constant guidance and support he provided throughout my stay at Virginia Tech. He not only provided new ideas and techniques to overcome hurdles in the project, but also led by example, by implementing some of toughest ones. I would also like to thank him for his advice on numerous occasions on non-academic topics. I would also like to thank Dr. Michael Keenan for not only serving on my committee but also sitting down and debugging code on numerous occasions. I would like to thank him for his attention to detail which often made me realize aspects that I had overlooked. I would like to thank Dr. Walling Cyre for serving on my committee.

I am indebted to John Shalf for all the help he provided. To all the students in the Computer Research Lab, thank you for the pleasant company. I would like to thank my parents for their infinite love and encouragement without which this degree would have been impossible to attain. I would also like to thank my brother for always being on my side whenever I needed him. Finally, I dedicate this thesis to my lovely sister, Nilofer, for, she was the greatest source of inspiration.

Table of Contents

List of Figures

Chapter 1

Introduction

The computer industry has experienced four generations of development, marked by the changing of building blocks from relays and vacuum-tubes (1940-1950s), to discrete diodes and transistors (1950-1960s), to small and medium-scale integrated (SSI/MSI) circuits (1960-1970s), and to large and very large scale integrated (LSI/VLSI) devices (1970s and beyond). There has been an exponential increase in processor speeds since the appearance of the first processor. It has seen the emergence of progressive terms: Megaflops, Gigaflops and Teraflops. An electronic greeting card of today has about the same computational power as a high performance machine 25 years ago. A machine shared formerly by hundreds of users is now owned outright by a single user. Performance increases can be attributed to advances in both technology-dependent and technology-independent areas. Processor performance can be measured in terms of the number of operations a processor can perform per second, or more commonly in terms of millions of instructions per second (MIPS). To reflect on the variables that control processor performance, an equation for processor speed can be written in terms of MIPS as

$$\text{MIPS} = (\text{instruction/cycle}) * (\text{cycles/second}) * 10^{-6}$$

The first factor is a function of the architecture and the second a function of technology. During the first two decades of computers, both forces made an equal contribution, but for the past twenty years, computer designers have been largely dependent on technology advances [1]. Densities of over twenty million transistors per chip have been achieved due to the advances in VLSI technology. VLSI technology is becoming well understood and no longer provides a competitive edge by itself [2]. This has led to widespread research in new architectural directions. Efforts are being directed on the first factor of the "MIPS Equation," instructions per cycle. The various alternatives to achieve this goal could be divided into three broad categories [3].

1. Achieve the same result with a fewer number of instructions.
2. Build specific-hardware into the architecture to improve its efficiency.
3. Execute many instructions concurrently.

The first alternative can be achieved by using a better class of algorithms, and then tuning the architecture to this class of algorithms. This is the most cost effective of all the three approaches. The second solution can be realized by adding hardware assists like instruction buffers and application-specific instructions. The last and the most expensive on the list is parallelism. Advances in VLSI technology have made parallel hardware viable. The bringing together of advances in integrated circuit technology, novel computer architectures and improvements in compiler technology has resulted in an almost two-fold increase in processor speeds every four years. Innovations in computer architecture will be required to sustain the growth rate; hence it is the focus of research of many computer designers.

In this thesis, a hybrid option consisting of Options 2 and 3 is explored to increase the effective number of instructions per cycle. Exploiting data parallelism and extracting

information about hardware assists for a specific application program during compile time have already been demonstrated in [4, 6]. The hardware information is used to program an array of field programmable gate arrays (FPGAs) to create hardware assists for the core processor. The application program is then executed in parallel on the new architecture. Such an architecture is referred to as an *adaptive architecture* [4]. Adaptive architectures reap large performance benefits of application-specific processors while maintaining their general-purpose nature.

1.1 Motivation

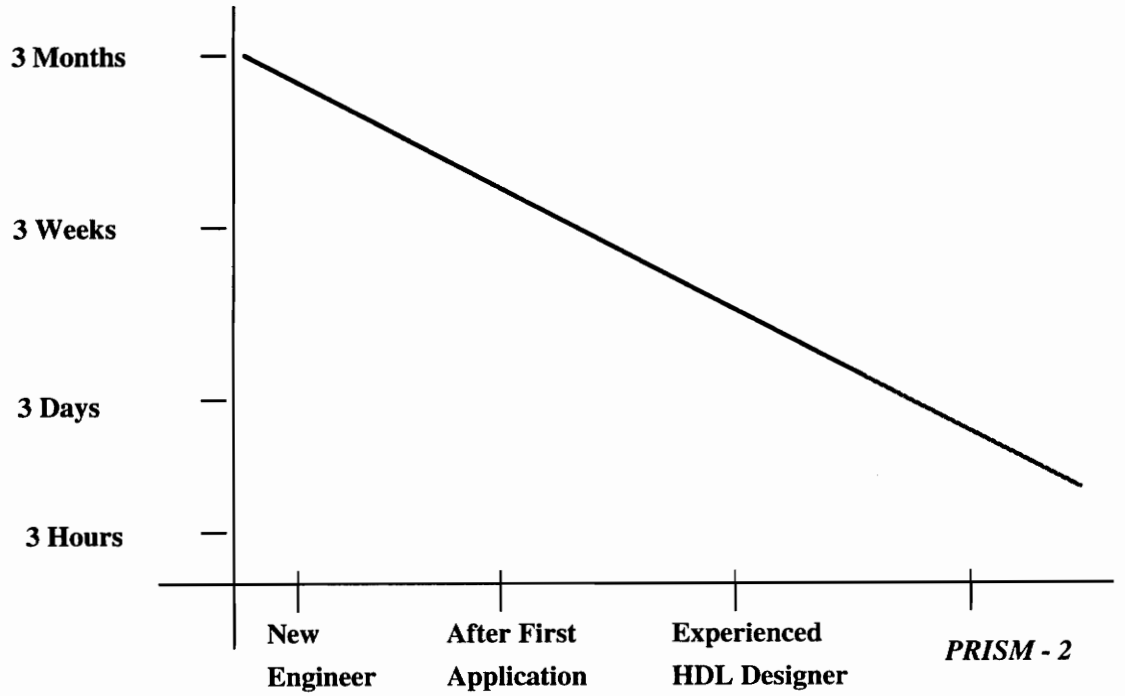


Figure 1.1. Application development time for customizing hardware structures.

The idea of adaptive architectures has been around for a long time. The reason for the idea not taking root in mainstream computing has been mainly because the job of extracting information for synthesizing hardware structures from the HLL code depended mainly on the programmer. The programmer not only had to be well versed in the high-level language but also in the area of hardware description languages. Figure 1.1 shows the application development time for synthesizing hardware structures. The time axis is proportional to the experience of the engineer. The objective of PRISM-2 is to not only to eliminate the *experience* variable but to also speed-up this process by automatically extracting the desired information from the input HLL and synthesize structures for that portion of the code.

1.2 Thesis organization

This thesis describes a compiler that takes the user's program as input and produces an optimized output in terms of dataflow graphs (DFGs) suited for hardware synthesis. Such a compiler is referred to here as PRISM-2-- an acronym for *Processor Reconfiguration through Instruction Set Metamorphosis.*

Chapter 2 presents a brief review of adaptive computing and the previous work done in that area. It then introduces the PRISM-2, compiler and architecture, and outlines the objective. The terminology for dataflow graphs is then presented. All DFGs described here, are composed of basic building blocks. Each of these blocks is examined in detail. Distinctions between a conventional high language compiler and PRISM-2 are then made.

Chapter 3 describes the different techniques for the linear optimization of DFGs. The process of mapping high-level language constructs onto DFGs is first described. Example architectures for their hardware implementation are also presented. The *filter philosophy* on which the optimizations are based is then introduced. A filter is a black box whose input and output is a DFG, and the goal is to optimize a certain aspect of the DFG. A variety of filters are presented.

Chapter 4 reports on the performance of this compiler. The effect of the various optimizations on example HLL code is shown. A comparison is made between the execution times of a few HLL programs in software and then, after their synthesis.

Chapter 5 concludes this thesis by outlining the shortcomings of this compiler and the future work to be done in this area.

Chapter 2

Background and Terminology

2.1 Review of adaptive computing

Broadly speaking, computing machines can be divided into two major categories, general-purpose architectures and application-specific architectures. A relatively new area of research is incorporating adaptation in computer architecture. An *adaptive* machine is one in which the configuration and the fundamental operations of the machine can *adapt* to the portions of the program that consume the most execution time. Adaptive computing platforms retain the identity of general-purpose architectures yet reap the benefits of application-specific machines. The idea of adaptive architectures has been around for a long time but its feasibility had not been demonstrated until recently. In 1960, Estrin [5] proposed a machine architecture made of a fixed, general-purpose core augmented by application-specific sub-structures. In 1978, Rauscher and Agarwala [6] introduced a method of task adaptation in machines with writable *control stores*. In this method a new set of instructions better suited for the task of executing the input program was derived from the user's program itself. These new instructions were written in the control store thus adding to the functionality of the core processor. RAM-based

reprogrammable gate array devices made an appearance in the mid-1980s [7]. Their presence in the market heralded a new era in adaptive computing. In 1990, Gokhale et al. [8] and Bertin et al. [9] used RAM based, field programmable gate arrays (FPGAs) to augment a processor. Gokhale et al. reported large speedups, of up to 2700, for certain image-processing applications [8]. The reason for the above architecture not taking root in mainstream computing was the immense burden on the programmer. Firstly, the programmer had to have a good idea of the portions in their code which dominated execution time. Secondly, and more importantly, the person had to have the hardware-design expertise to be able to synthesize that portion of the code. Thus one not only had to have knowledge in *high-level language* programming, but also had to have some experience in programming with *hardware description languages.*

Recently, a prototype general-purpose computing platform called PRISM (Processor Reconfiguration through Instruction-Set Metamorphosis) was demonstrated [4]. It consisted of a compiler that took a program in a high level language, such as C, as its input, and produced data outlining the portions of the program that could be executed in hardware. Such a compiler is referred to as a *configuration compiler* and the output "hardware data" is referred to as the hardware image [4]. As in the case of Gokhale *et al.* and Bertin *et al.*, with an array of FPGAs augmenting the general-purpose processor, the array could be configured easily from the hardware image and thus tune itself to an application-specific task. The programmer had a minimal part to play in the entire process.

2.2 PRISM-2 Compiler and Architecture: Systems Overview

A block diagram of an adaptive computing system utilizing PRISM-2 is shown in Figure 2.1.

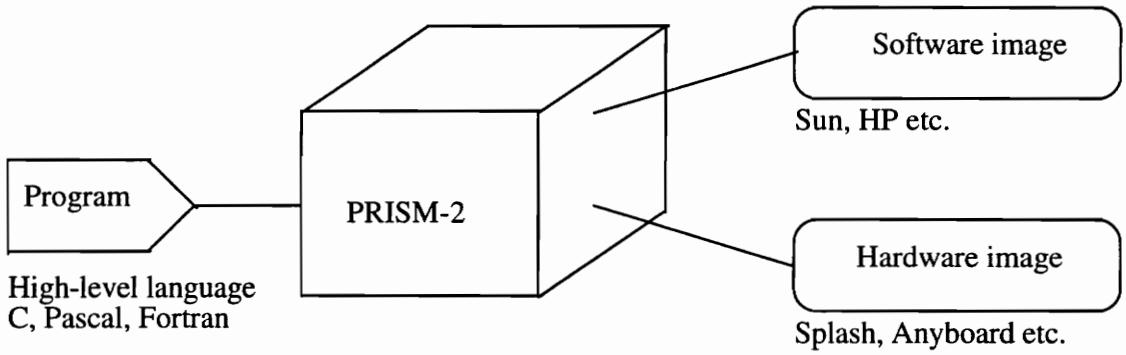


Figure 2.1: An adaptive computing system using PRISM-2.

The goal of the PRISM-2 compiler is primarily to identify computationally intensive portions of the input program and to synthesize hardware structures for these portions. The entire process needs to be fully encompassed within the compilation process so that the entire procedure is transparent to the user or the programmer. The user's high-level language (HLL) program is fed as input to the PRISM-2 compiler. PRISM-2 transforms the input HLL into an internal dataflow graph (DFG) representation and performs various optimizations on the DFGs. The output of the PRISM-2 compiler consists of two files; a *hardware image file* and a *software image file*. The hardware image file specifies the hardware structures needed to execute the program (synthesizable portion of the code). The software image file contains the code to be executed in software, and information on how this code is to be executed on the hardware specified in the hardware image file. The software executable can be run on any generic platform such as Sun, HP or DOS. The hardware executable can be mapped on to any reconfigurable

media like the SPLASH board [10] or AnyBoard [11]. The SPLASH-2 board is currently in use at the Computer Research Laboratory, Virginia Tech.

Both the hardware and the software image files, together, semantically represent the original input program, but when executed in this manner will exhibit a speedup over a conventional compiled version of the original code. The reason for the speedup is that, the architecture was able to add the operations of the frequently accessed code to its repertoire and then perform all references to them as fundamental operations.

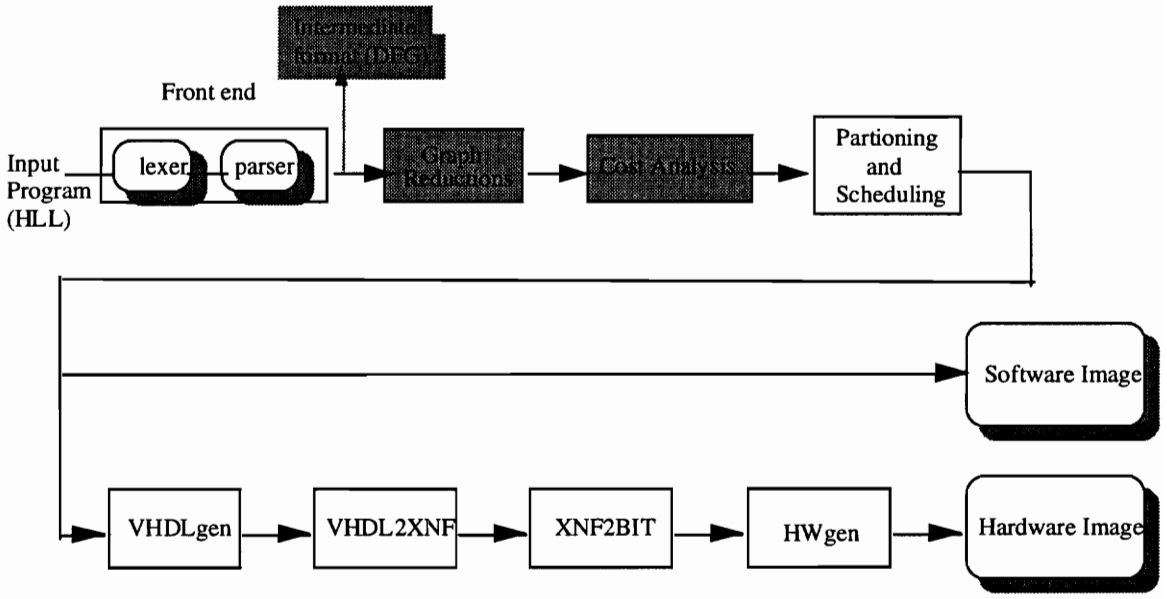


Figure 2.2: The components of the PRISM-2 compiler.

The main components of PRISM-2 are as shown in Figure 2.2. The shaded blocks in the above figure have been implemented by the author and will be the main topics of discussion in this thesis. The front end is responsible for *lexical* and *syntax* analysis, *parsing,* and mapping the HLL constructs onto the DFG format. The lexical analyzer and

the parser used here are standard UNIX™ utilities, Lex and Yacc. A Lex lexer is usually faster than a lexer written in C by hand [10]. The lexical analyzer divides the input into units called *tokens*. The parser establishes a relationship among the tokens following a set of rules referred to as a *grammar*. A Yacc parser is generally not as fast as a parser written by hand, but the ease in writing and modifying the parser is worth any speed loss. Whenever the input does not match any of the rules the parser reports a *syntax error*. An error recovery routine written by the user can determine the action to be taken in this case. The mapping of various C constructs is explained in detail in the next chapter.

The output DFG from the front end is optimized by the *graph reductions* block. There are two possible optimizations, *linear optimizations* and *loop optimizations*. In linear optimizations, parts of the DFG are scanned for specific patterns, and on matching these patterns a simple replacement is made. Loop optimization is a method of optimizing loop speed execution by replacing the entire loop with the code that comprises the loop body, duplicated the number of times that the loop would execute at run time. This optimization, clearly, can be done only when the loop-control criteria can be computed at compile time. Only linear optimizations are discussed in this thesis. Loop optimizations, though, being an integral part of the PRISM-2 compiler, are out of the scope of this thesis [21, 22]. The *cost analysis* block evaluates the cost of synthesizing the DFG in hardware. Every DFG node is *scored* in terms of the area it would occupy, and the execution time to evaluate the computation specified by the node. The method of scoring and the associated *cost function* are discussed in Chapter 3.

Once the graph is scored, it is given to the *partitioning and scheduling* block. The input DFG is first given to the *scheduling* step. Scheduling is the task of determining the

start time (time at which the operation starts execution), subject to the precedence constraints specified by the DFG [14]. Different scheduling algorithms have been proposed [14, 23, 24], but it is not the intent to describe them here. They are described in detail in the references. Once the DFG has been scheduled, functional units (adders, multipliers, etc.) are bound to the DFG nodes. This step is also referred to as *allocation*. Depending on the resources allocated and the cost to synthesize these resources, a decision to map the DFG nodes onto, hardware structures or software constructs, is made. This is also referred to as *hardware-software partitioning*. The DFG nodes that cannot be synthesized are written to a software image file, in the form of software constructs.

The next step is to generate a description of the hardware in terms of a hardware description language (HDL). The HDL used for specification here is VHSIC (Very High Speed Integrated Circuit) popularly referred to as VHDL. A standard utility, the Xilinx netlist format (XNF) generator, converts the hardware description into a netlist format. This format can be further translated into the bit pattern required to program the reconfigurable media (an array of FPGAs). This task is accomplished by the blocks *xnf2bit* and *hwgen*. The configured media constitutes the hardware image file.

The PRISM-2 and conventional compilers

The reference to PRISM-2 here encompasses the entire class of *hardware compilers* (including silicon compilers [12]). *Hardware compilation* can be defined as a translation process from a higher-level description in the behavioral domain to hardware structures in the physical domain [12]. The PRISM-2 compiler is analogous to software

compilers. Many of the techniques of the PRISM-2 compiler are borrowed and adapted from the rich field of compiler design [13]. Although there is a striking similarity between the two, there are some differences.

1. Conventional compilers are interested in what the high-level program does and how fast it can be done, and consequently operate only in the behavioral domain. The PRISM-2 compiler besides being interested in the fast execution times, also performs the mapping of the representation onto a set of components and connections. Thus, it performs a mapping from the behavioral domain to the physical domain. Such a mapping is also referred to as *high-level synthesis* [25]. The back end of a compiler performing high-level synthesis is much more complex than that of conventional compilers, because of the requirements on timing and interface of the internal operations of the target architecture.

2. The PRISM-2 compiler, like many vector processor compilers performs loop expansion on iterative constructs with data-independent exit conditions. In conventional compilation the optimized output code is usually executed on a machine with some *cache* memory on it. The *principle of locality*[1] on such machine architectures is better satisfied when the loop is not unwound. In case of the PRISM-2 compiler, unwinding the loops exposes the underlying parallelism. The loop can be executed much faster when unrolled, by the PRISM-2 compiler. As stated earlier, only certain types of loops can be unrolled.

[1] The *principle of locality* states that if an item is referenced in memory, then the neighboring items will be referenced soon.

3. Conventional compilers have a fixed length format for variables. For example, if a variable *a* was declared as an integer in C, then it would be allocated 32 bits in the memory mapping step. The PRISM-2 compiler does away with the fixed lengths. For example, if the variable *a* was involved in a multiplication with a constant (also 32 bits wide), and if both the operands could be represented by a smaller number of bits (say ten), then the PRISM-2 compiler would recognize this fact and perform such an optimization. As a direct consequence of this optimization a (10 x 10) bit multiplier would be instantiated instead of a (32 x 32) bit. Thus, conventional compilers optimize only with a speed constraint while the PRISM-2 compiler optimizes with *speed and area* constraints.

2.3 Dataflow graphs as intermediate representation

This section introduces the terminology for DFGs, and the different nodes used. An example DFG file in the Xgrab[2] format is also provided. Finally, the use of DFGs over control flow graphs (CFGs) for intermediate program representation is justified.

2.3.1 Terminology

A *graph G(V,E)* is a pair (V,E), where V is a set and E is a binary relation on V. The elements of the set V are called the *vertices* and those of the set E are called the *edges* of the graph. In a *directed graph* (or *digraph*) the edges are ordered pairs of vertices; in an *undirected graph* the edges are unordered pairs. An edge (directed or undirected) is

[2] Xgrab (X graph browser) is a tool for the automated layout and editing of directed graphs. It was developed by Greg Barnes at Tera Computers, and the GRAB group at UC Berkeley.

incident on a vertex when the vertex is its end-point. Directed and undirected graphs can be weighted (vertex weighted or edge weighted). The *degree* of the vertex is the number of edges incident on it (also called the fan-in). An edge with two identical end-points is called a *loop*. A *path* is an alternating sequence of vertices and edges. A *cycle* is a closed walk with distinct vertices (except the first and last vertices). A graph with no cycles is called *acyclic*. A graph is *connected* if all vertex pairs are joined by a path. Vertices of a tree are also called *nodes*. With all these definitions let us formally define a DFG. A *dataflow graph* is a directed graph whose nodes correspond to the operators and the arcs are the pointers for forwarding operands in data packets. These data packets are also called *tokens* [4].

2.3.2 Nodes used in Xgrab format

The node types presented below are the basic building blocks used for representing computations expressed in HLLs. If we assume the syntax of the HLL to be that of C, then *all* the C constructs (though not all have been implemented) can be mapped onto the DFG format using these basic building blocks. The mapping of these constructs is explained in detail in the next chapter. Without any loss of generality, this section assumes the HLL as C and uses the C syntax for the representation of the DFG nodes.

1. *Operator node:* This node denotes the operations that are members of a HLL's fundamental arithmetic and Boolean set (for example, +, -, &, |, etc.). Figure 2.3.1 shows the computation of ($c = a / 2$). Associated with every operator node are output and input

ports. The output port contains the result of the operation performed on the input ports. The input port order is important in *non-associative* operations like subtraction, division and exponentiation. For example, (*a* / 2) and (2 / *a*) are not equivalent expressions. The order of the operation is preserved by assigning the *ends* (subtrah*end*, divid*end*, etc.) to the first port and the *ors* (subtract*or*, divis*or* etc.) to the next one.

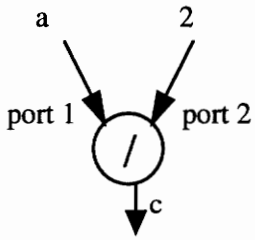


Figure 2.3.1. An Operator node.

2. *Conditional node:* This node represents a comparison between the two input arcs. The output is a Boolean value indicating the result of the comparison (for example, <, >, ==, !=). For example, in Figure 2.3.2 if *a* was less than *b* then the value of *flag* would be true (logic '1').

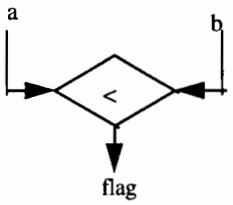


Figure 2.3.2: A Conditional node.

3. *Select node:* A Select node is shown in Figure 2.3.3. The output of this node depends on the *control* line. If the *control* line is true, then the *true* input is propagated to the output, otherwise the *false* input is presented at the output.

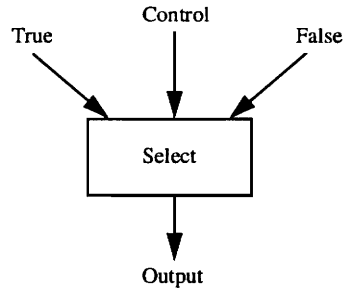


Figure 2.3.3: A Select node.

4. *Closure node:* A Closure node is shown in Figure 2.3.4. This node latches the value of the input arc, *value*, when the *closure_en* line is asserted. The node then holds on to this value and reflects it at the output. The *implied clock,* clocks the Closure node operation of latching. It is *implied* because it is not shown explicitly in the DFG representation.

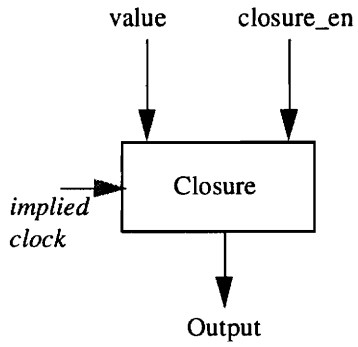


Figure 2.3.4: A Closure node.

5. *Function node:* This node executes a call to another imbedded function (which, itself could be represented as a DFG). The inputs to the function node are the arguments of the function and the output of this node is the return value from the function/DFG. Figure 2.3.5 shows a Function node.

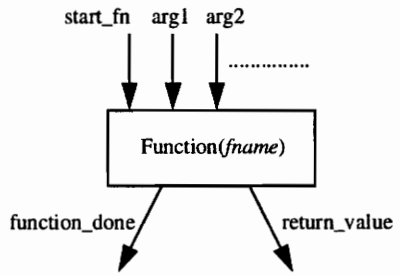


Figure 2.3.5. A Function node.

6. *Store node:* A Store node is shown in Figure 2.3.6. This node is used for storing data into memory. In the figure, when the completion signal *done* (Boolean) is asserted, it implies the address and data signals are stable. When this is true, data from *data* is stored into memory at the computed address. The output signal *store_done* (Boolean) is asserted to indicate the completion of the store operation. This node, like the Closure node, assumes an implied clock, which sequences a finite state machine (FSM) to perform the store operation.

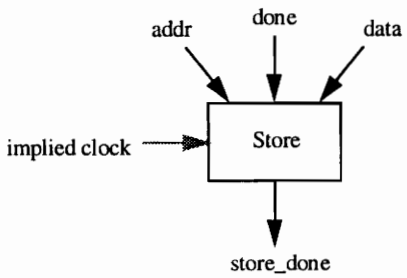


Figure 2.3.6: A Store node.

7. *Load node:* This node like the Store node, is also used for memory references, but is used to load data. An example Load node is shown in Figure 2.3.7. When the *done* completion signal is asserted, data from memory location *addr* are put on the output, *data,* of the Load node. In both the cases (Load and Store nodes) the memory is assumed to be local and resident. *Load_done* is asserted when the load operation is complete.

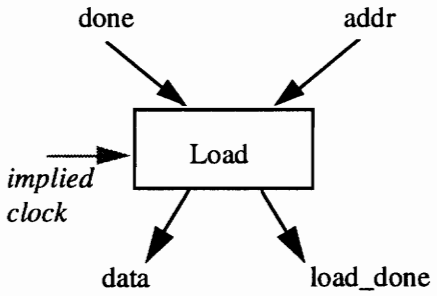


Figure 2.3.7: A Load node.

Before the Loop controller node is explained, it would be better to gain some insight into the concept of *completion* signals. All nodes depend on completion signals produced by other nodes. This is because executing the program in the dataflow domain leads to the issue of *synchronization*. The problem of synchronization is solved here by generating the various completion signals (*dones*) and making dependent actions wait on this completion signal. For example, consider a Store node. If there were no completion signal, *done*, then the store operation could occur while the *addr* lines (memory address) were not stable, and hence data in that memory location could be overwritten. Since this operation is performed only when the *done* signal is asserted this situation is avoided.

8. *Loop controller node:* This is the only node that produces four *different* output signals. The loop controller node has four inputs and four outputs. It controls the execution of loops. An example of the Loop controller node is shown in Figure 2.3.8.

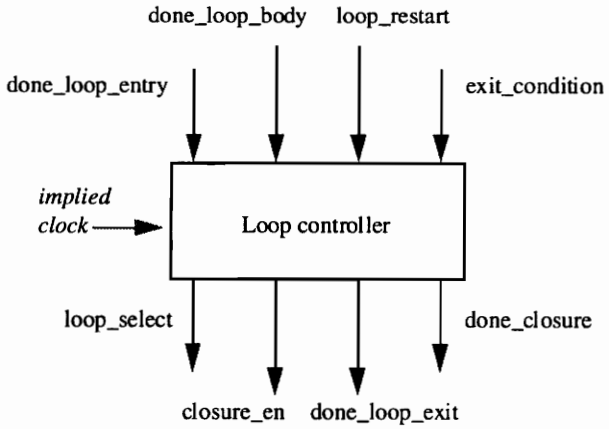


Figure 2.3.8: A Loop controller node.

Before the explanation for the signals of the Loop controller node it would be useful to see how loops (for example, *while* and *for* loops) in a high-level language are represented in terms of DFGs. Figure 2.3.9 shows the DFG representation of a loop. This structure is examined in detail in Chapter 3 but is presented here to explain the signals of the Loop contoller node. The variables present within a loop, and modified by the loop during successive loop iterations are referred to as *inner-loop variables*. Depending on the number of inner-loop variables, an equal number of Closure and Select nodes are instantiated for that loop. In Figure 2.3.9 only one Closure and Select node pair is present implying the declaration of a single *inner-loop variable*.

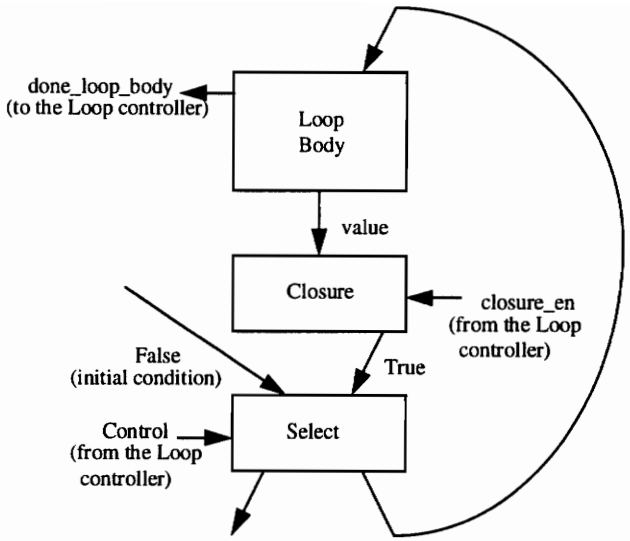


Figure 2.3.9. A DFG representation of a loop.

A brief explanation of the eight Boolean signals produced by the Loop controller node follows.

• *done_loop_entry:* This completion signal is asserted true by the presence of all variables outside the loop, on which the loop depends. These variables are also referred to as *outer-loop variables*. Assertion of this signal allows the loop to begin execution.

• *done_loop_body:* This completion signal is produced by the last action in the loop body (refer to Figure 2.3.9). It allows the evaluation of the exit condition for the loop by delaying all further iterations until then.

• *exit_condition:* When the exit condition for the body of the loop is true, then this signal is asserted.

• *loop_restart:* When asserted the state of the entire loop is reset to the beginning (loop *restarts*). This signal is used only in cases of nested loops.

• *loop_select:* This signal acts as the *control* signal for the Select node shown in Figure 2.3.9. Depending on the state of *loop_select*, the Select node chooses the *True* or

False branches. The *True* branch represents the output of the Closure node (value of the inner-loop variable after being modified in the loop body) and the *False* branch represents the initial condition of the inner-loop variable

- *closure_en*: As the name suggests this signals serves as the enable for the Closure node in Figure 2.3.9.
- *done_loop_exit:* This signals the completion of the loop. It is a completion signal to the nodes outside of the loop depending on the *loop outputs*.
- *done_closure:* This signal is asserted when the data dependencies within the loop are satisfied.

Like the Load and Store nodes, the Loop controller node has imbedded within it a state machine controlling its functions. The state machine is as shown in Figure 2.3.10. The state diagram depicts five states; *idle*, *start*, *run*, *finish_zero_iteration*, and *finish*. If the input conditions do not match those shown in the figure then the state of the Loop controller is preserved. It shows the production of the output signals by the loop controller based on the states of the input signal.

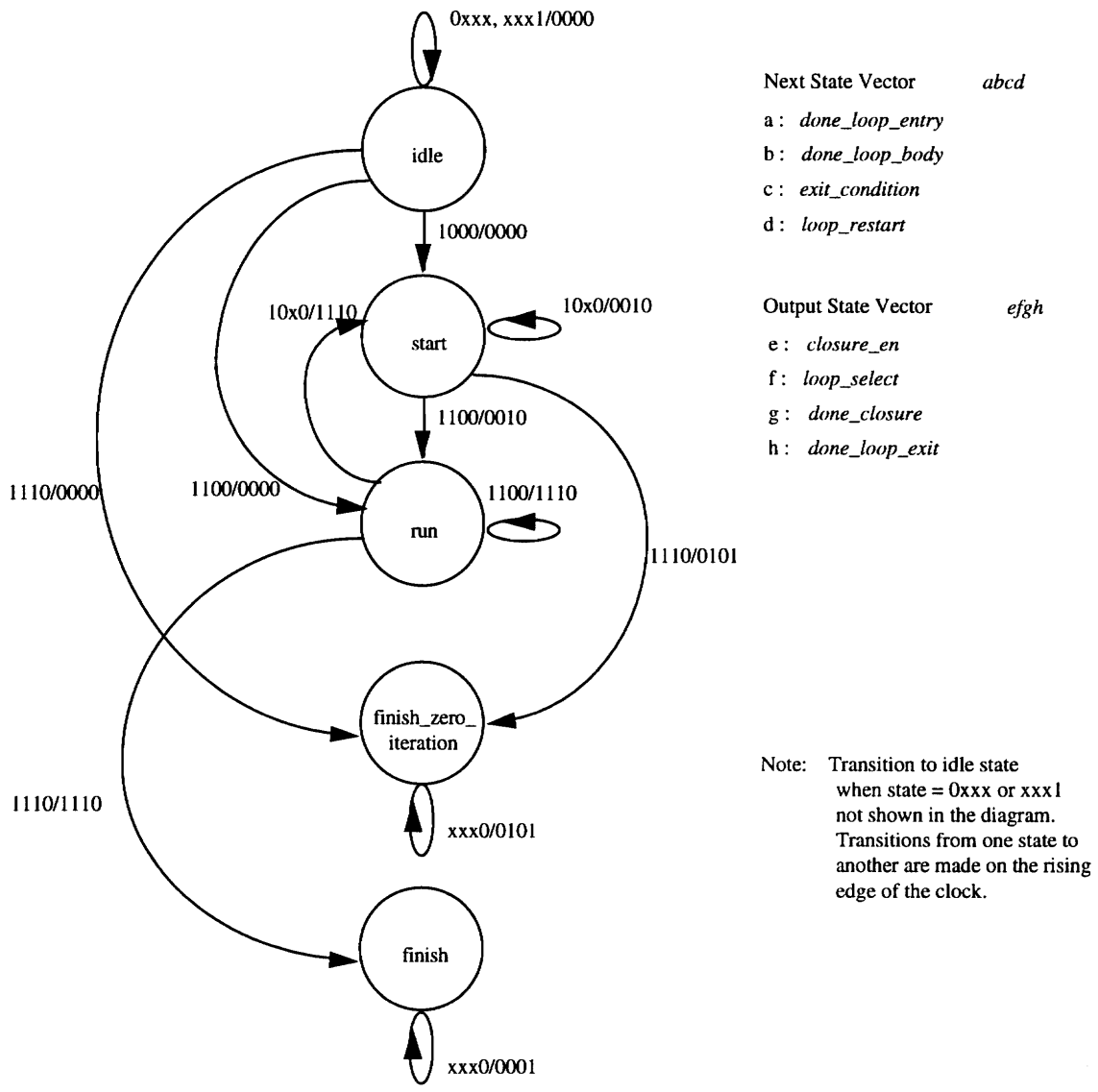


Figure 2.3.10. State diagram for the loop controller hypernode.

A brief explanation of the five states is presented.

•*idle:* This is the initial default state of the Loop controller. The Loop controller stays in this state till the assertion of *done_loop_entry*. At any point in time, if the *done_loop_entry* signal is disabled, the Loop controller returns to this state. In this state none of the outputs are asserted.

•*start:* The Loop controller goes into this state when the signals *done_loop_body* and *loop_restart* are low, and the outer-loop variables are present. In this state only the *done_closure* signal is asserted.

•*run:* In this state all the outputs, except for *done_loop_exit*, are asserted. The Loop controller finds itself in this state when the signals *done_loop_body* and *done_loop_entry* are asserted, and the signals *exit_condition* and *loop_restart* are false.

•*finish_zero_iteration:* The Loop controller goes in this state when all the inputs, except for *loop_restart* are asserted. This state asserts the *done_loop_exit* and the *loop_select* output signals.

•*finish*: This state represents the completion state and hence only the done_loop_exit signal is asserted here. A transition is made to this state from the *run* state when all the inputs, except for loop_restart, are true.

2.3.3 DFG representation in Xgrab

Figure 2.3.11 shows a simple program in C. Figure 2.3.12 is the corresponding grab file as visualized by Xgrab and Figure 2.3.13 is the actual grab file. For brevity, Figure 2.3.13 does not list all the node and edge attributes.

```
function_example(a,b)
char a,b;
    {
    char d, e[20], c=10;
    if (a==1)
       c=a+2;
    d=fft(c,a);
    e[a]=d;
    return (e[a]);
    }
```

Figure 2.3.11. An example function in C

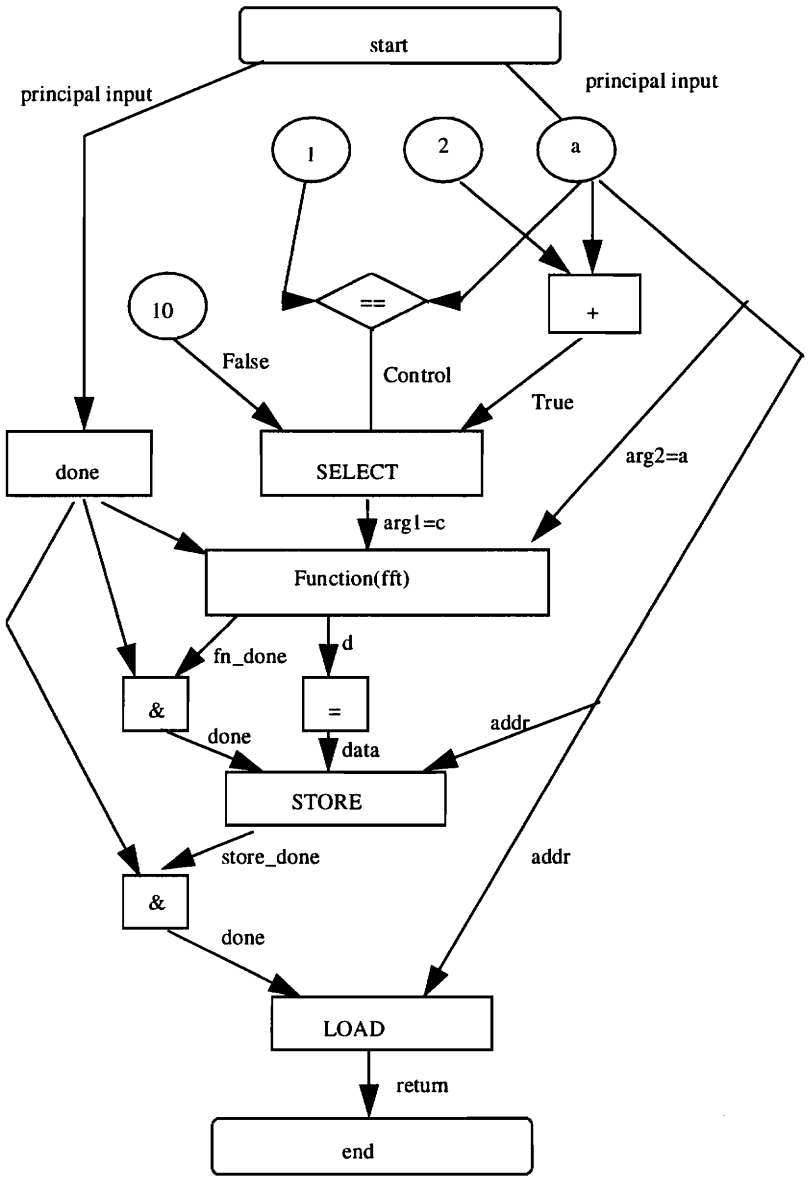


Figure 2.3.12. The DFG file of the C program in Figure 2.3.11, as visualized in Xgrab.

```
*NAME*
func_test DFG
*NODES*
n2
#19     "start"
#26     "Load"              shape=rect        fanin=2 inpin1=addr     inpin2=done
outpin1=return
#29     "Store"             shape=rect        fanin=3 inpin1=addr     inpin2=data
inpin3=done     outpin1=store_done
#36     "a"                 shape=circle      fanin=1
#39     "="                 shape=rect        fanin=1
#45     "Function(fft)"     shape=rect        fanin=3 inpin1=c        inpin2=a
inpin3=a        outpin1=d       outpin2=fn_done
#48     "2"                 shape=circle      fanin=0
#49     "+"                 shape=rect        fanin=2
#52     "10"                shape=circle      fanin=0
#55     "1"                 shape=circle      fanin=0
#56     "=="                shape=diamond     fanin=2
#59     "SELECT"            shape=rect        fanin=3
#62     "done"              shape=rect        fanin=1
#65     "&"                 shape=rect        fanin=2
#72     "&"                 shape=rect        fanin=2
#20     "%end"
*EDGES*
net_edgelabels
#19     #62     label=principal input
#19     #36     label=principal input
#26     #20     label=return
#29     #72     store_done      size=1
#36     #26     label=addr
#36     #29     label=addr
#36     #56     size=8
#36     #49     size=8
#36     #45     arg2=a          size=8
#39     #29     label=data      size=8
#45     #65     label=fn_done   size=0
#45     #39     label=d         size=0
#48     #49     size=32
#49     #59     label=true      state=true      size=8
#52     #59     label=false     state=false     size=8
#55     #56     size=32
#56     #59     label=control   size=8
#59     #45     arg2=c          size=8
#62     #72     size=1
#62     #65     size=1
#62     #45     size=1
#65     #29     label=done      size=1
#72     #26     label=done      size=1
```

Figure 2.3.13. The grab file for the example C program.

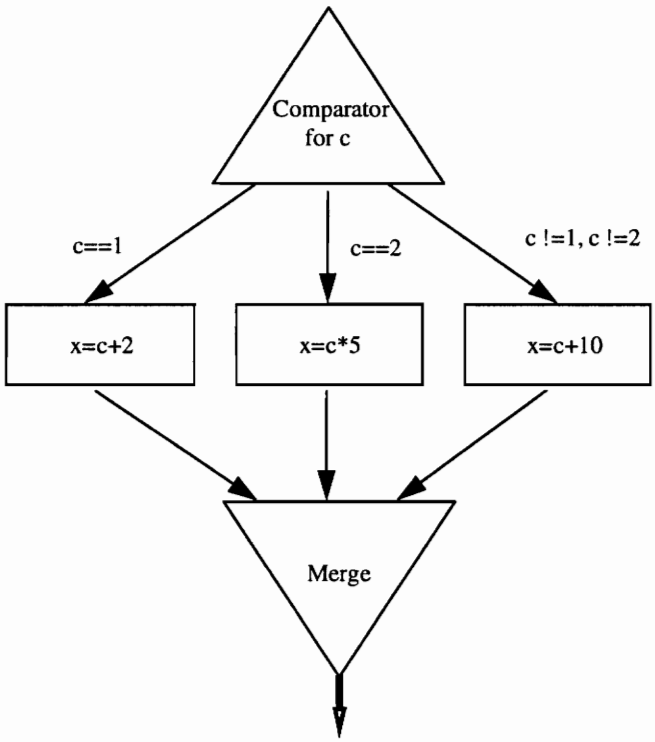
All nodes deriving their inputs from the *start* node represent the *principal inputs* to the graph. All nodes connected at the input ports of the end node are the *primary outputs* of the graph. In the C code, the line *e[a]=d* implies the base address of the character array, *e*, to be *0*. Hence the a^{th} element of the array would be at memory location *a*. The base address allocation is done while mapping the HLL code to a DFG. This is explained in detail in Chapter 3. The *Store*, *Load* and the *Function* nodes have attributes such as *inpin1*, *inpin2* etc. These attributes are used to connect the arguments to the proper input ports. In the case of nodes producing more than one data type, the *outpin* attribute must be present. The *shape* attribute in the NODES section specifies the shape in the visualization domain. The *size* attribute in the EDGES section specifies the width of the data path. This information is useful in cost analysis and is explained in detail in the next chapter.

2.3.4 Dataflow and Control flow graphs

There are two possible alternatives that the intermediate program could have taken, control flow and data flow models. Before the advantages and disadvantages of either approach are discussed, it is appropriate to define *control flow graphs* (CFGs). A CFG is a directed graph whose edges show the sequence of execution choices between different basic blocks [16]. The CFG represents the execution semantics of the program, while dependencies are viewed as precedence constraints between statements that must be maintained for correct execution [17]. Traditionally compilers have used the CFG representation augmented with dependence information such as *def-use* chains [13]. A

def-use chain for variable x is a node pair (n1, n2) such that n1 defines x, and n2 uses x, and the definition of x at n1 reaches the use of x at n2 [19].

Figure 2.3.14 shows the CFG for the construct shown in the same figure, and Figure 2.3.15 shows the corresponding DFG.




```
if (c==1)
   x = c + 2;
else if (c==2)
      {
         x = c * 5;
         else
            x = c + 10;
      }
```

Figure 2.3.14. The control flow graph and C source.

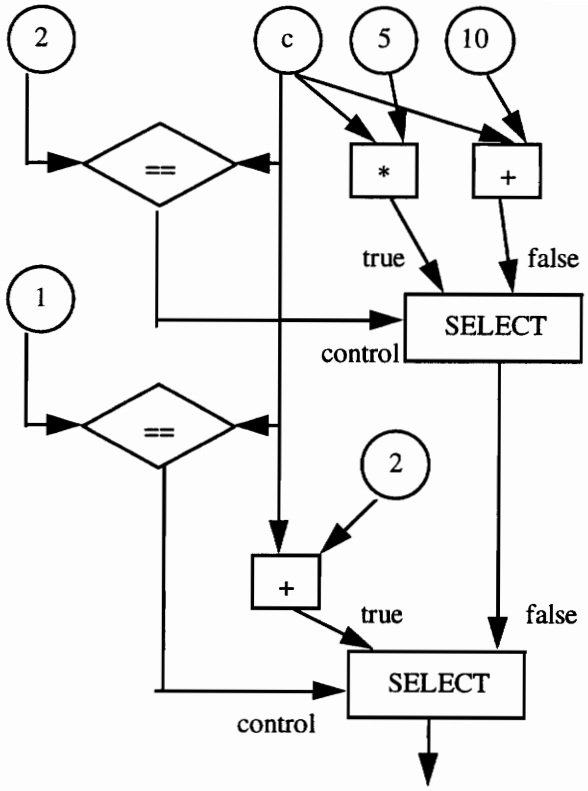


Figure 2.3.15. The DFG for the C source in Figure 2.3.14.

Drawbacks of the CFG model [27]:

1. When a program is transformed, dependence information may need to be modified. For example, after *eliminating dead code* it would be necessary to modify the def-use chains if CFGs were to be employed as intermediate representation. This modification would be very expensive (repeated program passes) as compared to the DFG approach.

2. CFGs can exploit only coarse-grain(task-level) parallelism while DFGs can exploit both coarse-grain and fine-grain parallelism (task-level and instruction-level).

3. It is easier to detect critical, computationally intensive portions of a program from the DFG representation than from the CFG one.

The instructions in dataflow computing do not impose any constraints on sequencing except the data dependencies in the program and hence parallelism can be extracted easily from the representation [18]. Each node in a DFG represents a task to be executed according to the precedence constraints represented by arcs (which also represent the data flow). Thus, the dataflow representation can be used to establish area/performance bounds and hence make area/performance tradeoffs [19].

Strictly speaking, the representation here, is not *purely* dataflow because it has some control flow associated with it. For example, loop constructs like FOR and WHILE have control flow imbedded in them. The asynchronous nature of the dataflow model of computation allows the exploitation of maximum inherent parallelism in many application programs and hence leads to the choice of DFGs as intermediate program representation.

Chapter 3

Linear Optimizations for Dataflow Graphs

This chapter examines various graph manipulation techniques for linear optimizations. Before doing so, the mapping of high-level language (HLL) constructs onto dataflow graphs (DFGs) is explained. The *filter philosophy,* on which the graph optimizations are based, is introduced and explained in detail. The implementation of the various *optimizing filters* is then described. Every optimizing filter reads a DFG in the Xgrab format, and produces an optimized DFG in the Xgrab format. The writing and reading routines are briefly described at the end of the chapter.

3.1 Mapping of high-level language constructs to dataflow graphs

Without loss of generality, the high-level language (HLL) syntax in this section is assumed to be that of C. In this section various HLL constructs are illustrated and their corresponding DFG representations are presented. Example architectures for the execution of the DFG model are also shown.

3.1.1 Conditional Constructs:

Figure 3.1.1 shows the HLL code and the corresponding DFG implementation. Figure 3.1.2 shows an architecture for one possible implementation. In Figure 3.1.1, the Select node (described in Section 2.3) corresponds to a multiplexor in hardware while the conditional node, (*select_ctrl_0 = a < 0*), corresponds to a comparator. The number of the conditional and Select nodes in a mapping depends on the number of *if..else* clauses in the HLL program, and on the number of variables in each such clause. In Figure 3.1.2, the NOT block *complements* the input bit pattern and passes the output to the Adder block. The Adder adds the complemented input to '1' in a twos complement fashion to produce the expression *(-a)*. The comparator output is asserted either *true* or *false* depending on whether the variable *a* is less than zero or not. The 2:1 multiplexor makes a choice, between the constant zero and the output of the twos complement adder, based on the output of the comparator. The *switch* construct can also be mapped onto a similar DFG.

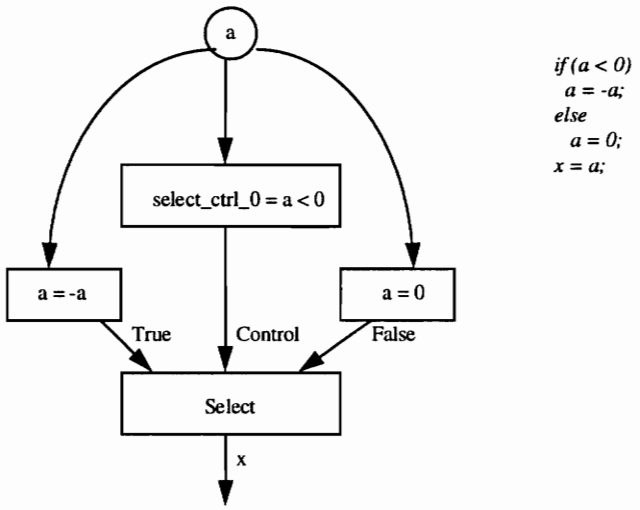


Figure 3.1.1. A DFG mapping for a conditional construct.

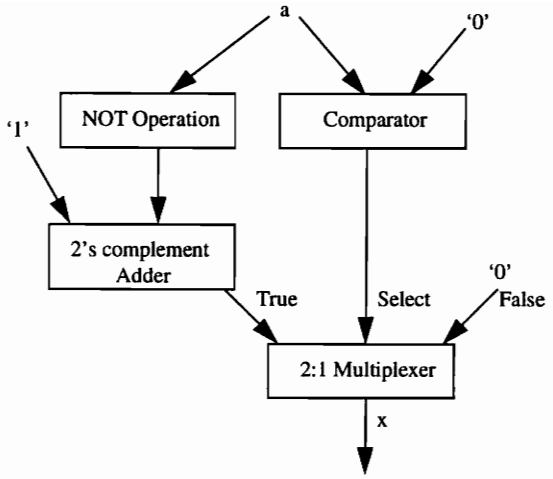


Figure 3.1.2. Example architecture for executing the DFG of Figure 3.1.1.

3.1.2. Loop Constructs

Consider the following simple loop in C:

```
j=1;
for (i=0; i<5; i++)
   j=j*2;
```

Figure 3.1.3 shows a DFG representation for the variable *j*. Note that this figure is the same as that presented earlier (Figure 2.3.9), except for minor modifications. The representation for variable *i* is similar to the one shown for variable *j* in Figure 3.1.3, except for the following two changes:

1. The loop body is replaced by $i=i+1$.
2. The initial condition, $j=1$, is replaced by $i=0$.

The Loop controller node is instantiated only once for every loop. In the representation for variable *i* (not shown), there is another arc originating from the Closure node for *i* and incident on the comparator node, $(i < 5)$. The output of this comparator node feeds the

exit_condition pin of the Loop controller node. This is illustrated in Figure 3.1.4. Figure 3.1.5 shows an example hardware structure needed to execute the DFG for the above loop (shown only for variable *i*). The Loop Controller node of the DFG has been mapped to a state machine. State machines consist of *combinational* (gates) and *sequential* (flip-flops) logic. Detailed information on specifying state machines in hardware description languages can be found in [20]. Depending on the loop body, appropriate *functional units* are instantiated. In this case the multiplier represents the functional unit. The Select node, as shown earlier, has been mapped to a 2:1 Multiplexer. The Closure node performs the function of a register, and consequently is mapped onto a *hardware register.*

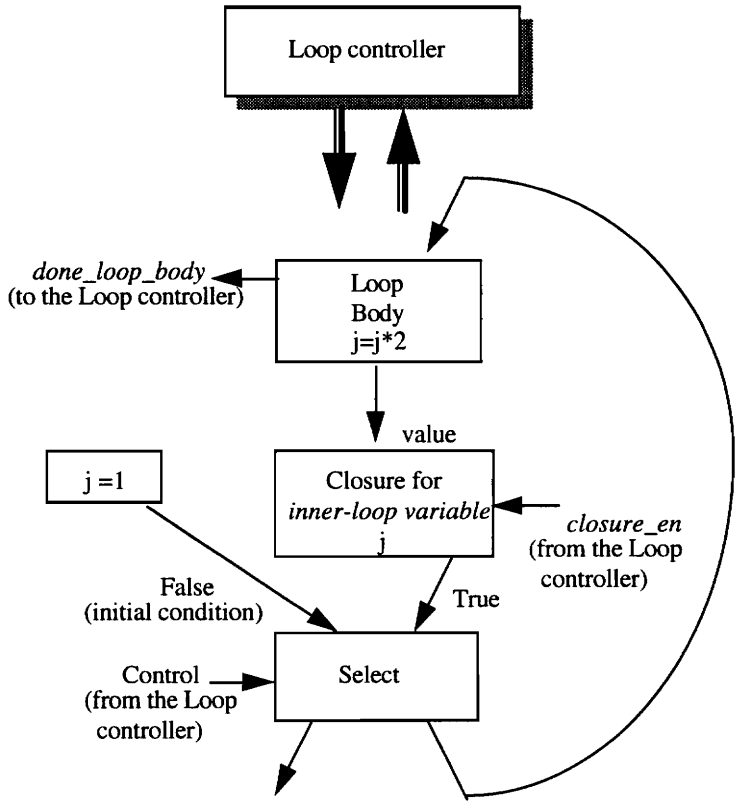


Figure 3.1.3. DFG representation of the loop for variable *j*.

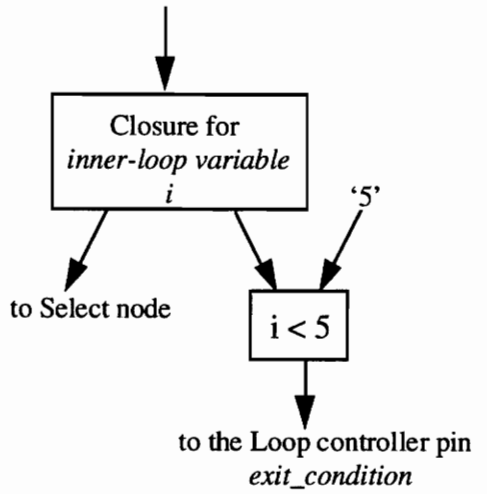


Figure 3.1.4. Computation of the exit condition for loop.

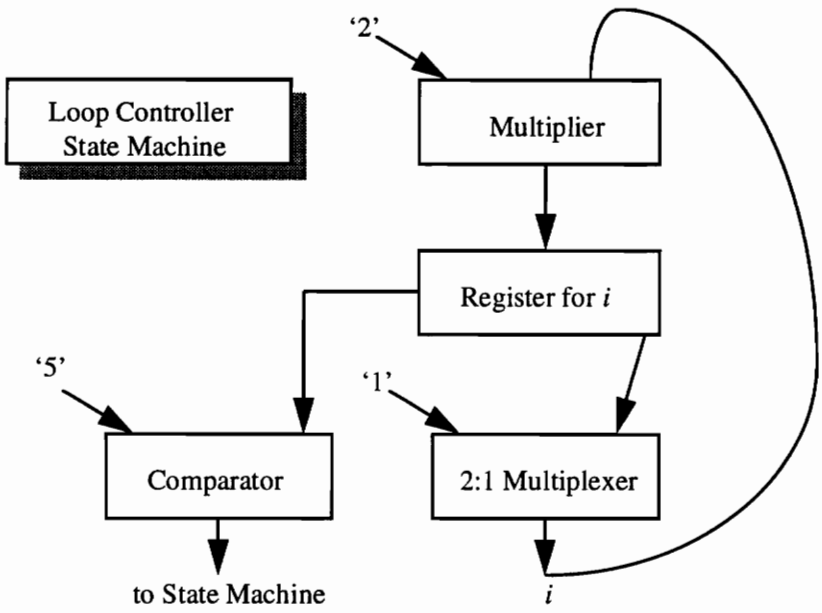


Figure 3.1.5. Hardware to execute the DFG (shown only for variable *i*)

3.1.3 Array and Structure Constructs

This sub-section and the following one utilize the Load and the Store nodes. These two nodes are used for memory references and their operation has already been explained in an earlier section. The front end of the PRISM-2 compiler recognizes a reference to an array and assigns a *base address* to it. All references to the array are then directed to this memory location. Consider the following reference to an array:

```
char b[20];
b[5]=99;
return b[5];
```

Figure 3.1.6 shows the DFG for the above code. The base address for the array *b* is fixed to *(0)* by the front end of PRISM-2. When the address *((0) + 5)* is *ready* the *done* signal for the Store node is asserted. Data *(99)* is stored in the memory location specified by the *addr* line. The statement *return b[5]* causes the Load node to put the data from address, *((0) + 5)* on the output, *data*, of the Load node. The above operations are just references to an on-board memory, and hence no specific hardware structure other than a memory module is required. Since the name of the array, *b*, is also a pointer, a similar mapping occurs for pointers too. In case of the *structure* construct, every element of the structure is assigned a base address and mapped in a similar manner as illustrated in Figure 3.1.6.

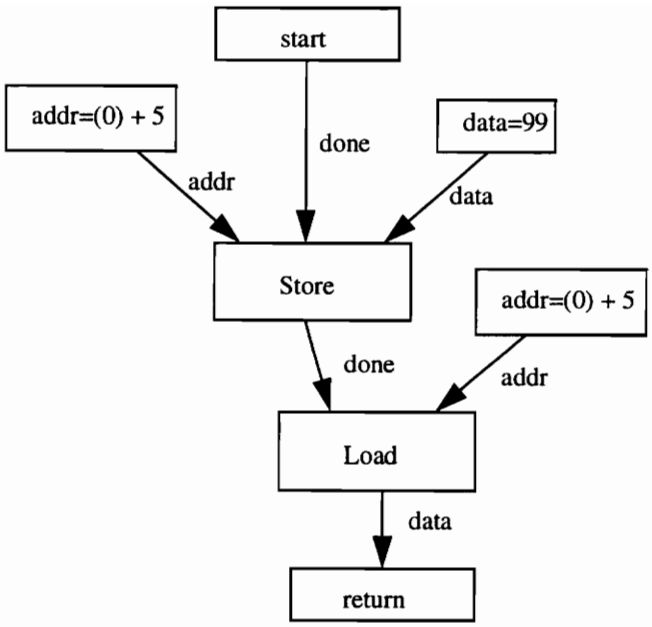


Figure 3.1.6. Dataflow graph for an array reference.

3.2 Graph Reductions

This section explains in detail the *graph reductions* block of Figure 2.2. It explains the different techniques used to create *efficient* dataflow graphs. Efficient dataflow graphs, here, refer to DFGs having faster execution time, optimized area in terms of functional units, and easier hardware implementation. It is important that the functionality of the original graph be preserved by the *transformations* carried out on the graph in order to optimize it. A second criterion for transformations is that they should, on the average, speed up the execution of the DFG. The DFG generated by the front end of PRISM-2 is manipulated to produce an optimized DFG. As mentioned earlier, there are two types of optimizations, also known as *reductions,* possible: linear reductions and loop reductions. Only linear reductions are discussed here. Figure 3.2.1 shows the big picture of the *graph reductions* block.

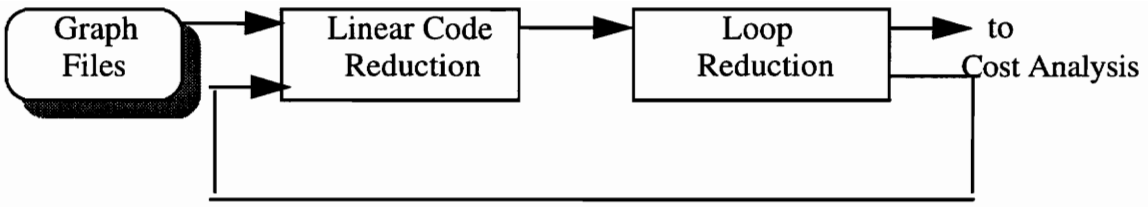


Figure 3.2.1. Graph Reductions: The big picture.

The filter philosophy:

Optimizations are carried out on the DFGs in a series of independent steps, comprised of discrete *filters*. Each filter optimizes a certain aspect of the DFG and propagates the output to another filter. This discrete step approach, wherein, optimization is implemented one at a time is more advantageous than performing a series of optimizations all at once. The reason for this is simple; optimizations can be *ordered* in this approach. For example, if carrying out optimization *A* after optimization *B* would result in a more efficient graph, then *filter A* could be scheduled after *filter B*. In the *non-filter* approach where the optimizations are based on a series of rules, the order of filter *A* and *B* would be predetermined (in fact there may not be a discrete filter *A* and *B*). Figure 3.2.2 illustrates the filter approach. The 'best approach' block in the figure requires user intervention, that is, the user is prompted for the best approach. The pitfall to be avoided in this approach is the generation of a DFG representation after a certain reduction, which hides or distorts functionality of the graph. The filter approach also ensures that if *filter n* was to produce a representation reducible by *filter n-1*, then, it will be done so in the next step. A possible cause of instability in this approach is the production of a graph by the n^{th} *filter* that is reducible by the $n+1^{th}$ *filter*, which in turn is again reducible by the n^{th} *filter*.

This would set up *oscillations* between the two filters. The decision to end these oscillations are left on the user, who, may decide to end the optimization process.

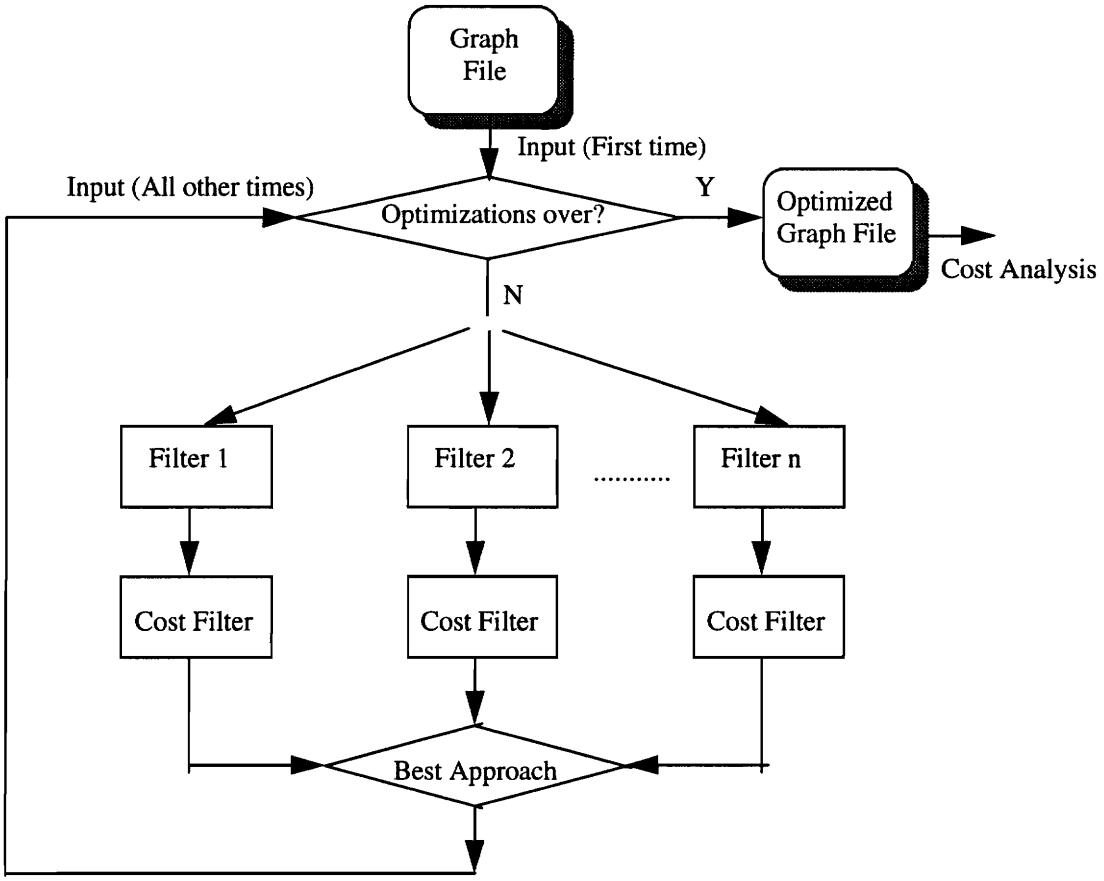


Figure 3.2.2. The Filter philosophy.

The linear optimizations are similar to the standard optimizations performed by a conventional compiler. Before describing the various reduction filters, the decomposing filter, also referred to as the *flattener,* shall be introduced first. The objective of this filter is to map the input dataflow format in terms of the basic DFG nodes. Next, the *cost filter* is presented. This filter provides a *cost function* to the DFG after the application of every

optimization step. Based on this cost function, the compiler can determine the usefulness of an applied transformation. An important point to note here is that only the cost filter is dependent on the target platform[1]; the other filters perform machine-independent optimizations. Due to this dependency, the ordering of the optimizing filters is largely, target machine-dependent.

3.2.1 Decomposition filter: *The flattener*

To effect parallel computation, the description of the program must be broken into many small fragments. The resulting graph after fragmentation is called the *decomposed* or *flattened graph.* The static quantities into which the program is decomposed are called *scheduling quanta* [21], for, they are the largest units that the *partitioning and scheduling* block of Figure 2.2 can handle. The scheduling quanta in this case refer to the basic building blocks introduced in Section 2.3.

Before any operations or optimizations are performed on the graphs, the graphs are passed through the flattener. The flattener, like the front end, is comprised of a lexer and parser. The *unflattened nodes* are divided into a series of *tokens* (done by the lexer) and depending on the *grammar* (specified in the parser), a new flattened version of the graph is created. The order of complexity for flattening the graph is directly proportional to the number of nodes in the graph, n, i.e., O(n). Figure 3.2.3 shows an *unflattened graph* being flattened. From the figure it can be seen that the compiler has no idea of the

[1] The target platform refers to the reconfigurable architecture on which the dataflow graph is to be synthesized. In this case the target platform is comprised of an array of field programmable gate arrays (FPGAs) [23].

functional units to be allocated from the unflattened graph. It is only after the flattening operation that this information becomes clear.

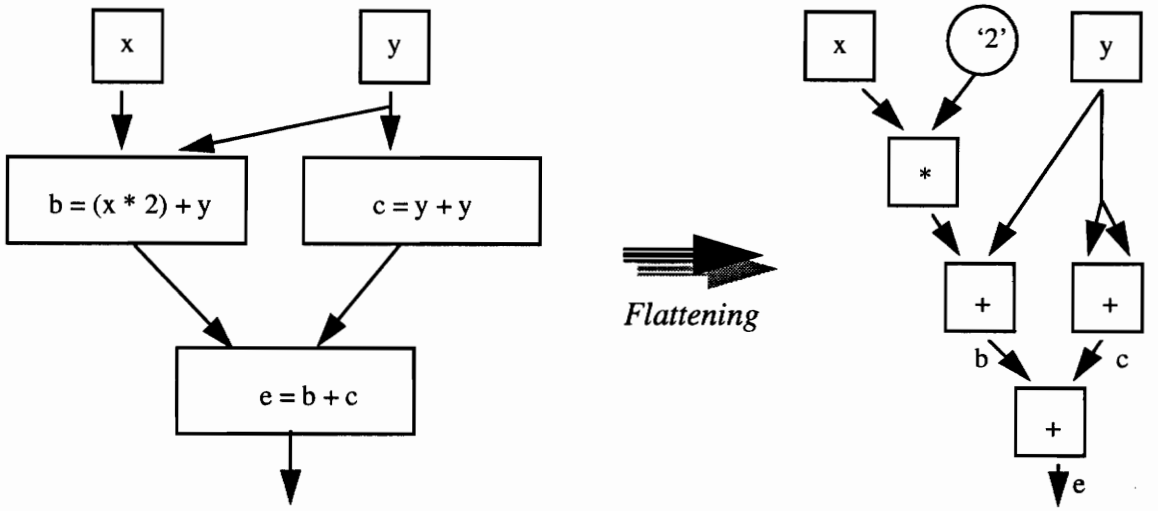


Figure 3.2.3. The Flattening operation.

3.2.2 Cost Filter

An *optimization problem* is one whose solution can be measured in terms of a *cost* function such that the cost function attains a *maximum* or *minimum* value [14]. The objective of the cost filter is to provide a cost function. The cost function or the *cost vector* is evaluated in terms of two components: operation cost and communication cost.

1. Operation cost:

Operation cost is the cost to perform an operation. For example, consider the node *(+)* which has two input variables *a* and *b*. Then, the *operation cost* would be the

cost of adding two variables on the target architecture. This cost is calculated in terms of the *time* to perform addition (delay) and the number of *configurable logic blocks*[2] (CLBs) it would take to synthesize an adder. Both these variables were found out experimentally by synthesizing various nodes (+, *, <<, etc.) with different inputs of varying sizes (variables and constants). Table 3.2.1 presents the synthesis results of the DFG nodes having a non-zero operation cost.

The fundamental operations of the C language can be broadly divided into the following four categories:

- *ADD*: This category encompasses the add (+), subtract (-), increment (++), and the decrement (--) operations.
- *MULTIPLY*: This includes the multiplication (*), division (/), and the modulo (%) operations.
- *SHIFT*: The left (<<) and right (>>) shift operations fall under this category.
- *LOGICAL*: This includes all the bitwise logical operations like *oring* (|), *anding* (&), *exoring* (^), and *complementing* (~). It also encompasses the relational operations like *or* (||), *and* (&&), *equal* (==) and *not equal* (!=).

Besides the fundamental operations considered above, there are two other types of nodes that present an operation cost when synthesized:

- *SELECT:* This category comprises of the Select node (multiplexer in hardware).

---

[2] Each field-programmable gate array device used here is comprised of a large number of configurable logic blocks (CLBs) among other logic elements. The number of CLBs occupied in a particular mapping gives an estimate of the area occupied by that mapping.

- *LOOP*: The Loop controller node (state machine in hardware) falls under this category.

The Load and the Store node, as stated earlier, do not require any hardware. The delay for a store or a load operation is assumed to be one clock cycle of the target architecture.

Table 3.2.1. Synthesis of the DFG nodes.

No.	Node Category	Input 1 (Type: Size)	Input 2 (Type: Size)	CLBs used	Max. Delay (Max. Clock)
1	ADD	VAR : 4	VAR : 4	5	26.0 (38.5)
2	ADD	VAR : 8	VAR : 8	11	61.8 (16.2)
3	ADD	VAR : 16	VAR : 16	26	97.3 (10.3)
4	ADD	VAR : 4	CON : 4	4	25.0 (40)
5	ADD	VAR : 8	CON : 8	9	42.2 (23.7)
6	ADD	VAR : 16	CON : 16	15	70.5 (14.2)
7	MULTIPLY	VAR : 4	VAR : 4	20	71.5 (14)
8	MULTIPLY	VAR : 8	VAR : 8	77	160 (6.2)
9	MULTIPLY	VAR : 16	VAR : 16	313	301 (3.3)
10	MULTIPLY	VAR : 4	CON : 4	7	40.9 (24.5)
11	MULTIPLY	VAR : 8	CON : 8	12	74.5 (13.4)
12	MULTIPLY	VAR : 16	CON : 16	23	154.2 (6.5)
13	SHIFT	VAR : 8	VAR : 4	23	51.8 (19.3)
14	SHIFT	VAR : 16	VAR : 4	60	63.9 (15.7)
15	SHIFT	VAR : 32	VAR : 4	114	70.3 (14.2)
16	SHIFT	VAR : 8	CON : 4*	5	25.0 (40)
17	SHIFT	VAR : 16	CON : 4*	12	25.0 (40)
18	SHIFT	VAR : 32	CON : 4*	29	25.0 (40)
19	LOGICAL	VAR : 4	VAR : 4	4	25.0 (40)
20	LOGICAL	VAR : 8	VAR : 8	6	25.0 (40)
21	LOGICAL	VAR : 16	VAR : 16	10	25.0 (40)
22	LOGICAL	VAR : 4	CON : 4	4	25.0 (40)
23	LOGICAL	VAR : 8	CON : 8	6	25.0 (40)
24	LOGICAL	VAR : 16	CON : 16	10	25.0 (40)

* Shifting a variable by a constant can be accomplished just by the reorganization of the input bit pattern. The resulting hardware in practice is very simple and does not require the number of CLBs shown in the table. The above misleading figures can be attributed to the inability of the synthesis tool to realize simple structures.

25	SELECT	VAR : 8	VAR : 8	9	25.0 (40)
26	SELECT	VAR : 16	VAR : 16	17	25.0 (40)
27	LOOP	-	-	13	46.7 (21.4)

In the above discussion, *VAR* stands for variable and *CON* for constant, *Size* is the number of bits, *Max Delay* is the maximum delay in nanoseconds and *Max Clock* refers to the maximum clock speed in MHz. Max Clock and Max Delay are related in the following manner: *Max Clock = 1/Max Delay*. In case of the SHIFT category, the inputs are ordered. For example, if *input 1* was a variable of eight bits, and *input 2* was a variable of four bits, then input1 could be shifted a maximum of fifteen places ("1111"). The following conclusions can be drawn from the results of Table 3.2.1.

1. *ADD* category: Consider the case where both the operands are variables; as the number of bits of the operands increases, the number of CLBs also increases by the same factor. For example, a 4x4 adder requires 5 CLBs, a 8x8 adder requires 11 CLBs (~5 * 2), while a 16x16 adder requires 26 CLBs (~11 * 2). The delays also increase by the same factor. The same conclusions can be drawn in the case where one of the inputs is a constant. The complexity of the operations in the ADD category can be approximated by the order of (m + n) = O(m + n), where m and n are the sizes of the input operands and O denotes the *order of*.

2. *MULT* category: Consider the case where both the operands are variables; as the number of bits of the operands increases, the number of CLBs increases by the *square* of the factor. For example, a 4x4 multiplier[3] requires 20 CLBs, an 8x8 multiplier requires 77

[3] There exist various configurations of multiplier circuits in hardware. The configuration used here is a parallel multiplier. Different results may be obtained by changing the *type* of multiplier.

(~20 * 4), and a 16x16 multiplier requires 313 (~77 * 4). The delay rises by the same factor instead of the square of the factor. When one of the operands is a constant the number of CLBs and the delay increase by the same factor instead of the square of the factor. For the (VAR x VAR) case, the complexity of operations in the MULT category can be characterized by O(m * n), while in the (VAR x CON) case it is characterized by O(m + n). Variables m and n bear the same meaning as in the ADD case.

3. *SHIFT* category: As mentioned earlier the SHIFT category is order dependent. If *input 2* is a constant then the delay also remains constant (=25.0 ns.). The number of CLBs increases by the same factor as the number of bits. If *input 1* and *input 2* are both variables, then both, the delay and the number of the CLBs, increase by the same factor. The above table does not reflect the true computational complexity of a shift operation and hence the complexity of the shift operation cannot be categorized from the above results.

4. *LOGICAL* category: The delays for all cases are the same (=25 ns) irrespective of the number of bits and the type of input (variable or constant). For the (VAR x VAR) case, the number of CLBs can be calculated in the following manner: Consider the 4x4 case (*i = 1*), 8x8 case (*i =2*), 16x16 case (*i=3*) and so on. Then, the number of CLBs (and hence the computational complexity) can be written in terms of *i* as

$$Number\ of\ CLBs = 2 + (2 \wedge i)$$

5. *LOOP* category: The Loop controller state machine when synthesized on the Splash-2 platform occupied 13 CLBs and it can be run at a maximum clock rate of 21.4 MHz.

6. *SELECT* category: An 8x8 multiplexer occupied 9 CLBs and could be operated at the maximum clock rate for Splash-2 (40 MHz). As the number of bits at the input of the multiplexor increases, the number of CLBs occupied also increases by the same amount.

2. Communication cost

Communication cost forms the second factor of the cost function. Consider the following two lines of code:

$$d = b + c;$$
$$g = d - 5;$$

The cost of the addition node (performing the operation of adding variables b and c) will be comprised of just the *operation cost*. In order for g to be computed, d has to be computed first and then it has to be *communicated* to the *subtract* node which computes the variable g. This communication occurs through the edge connecting the *add* and *subtract* nodes. The cost of communicating d is referred to as the communication cost. The wider the path between the two nodes, the smaller is the cost. The total cost on the subtraction node is the sum of the *operation cost* and the *communication cost*.

Once every node of the DFG is assigned a cost vector, the total cost of the DFG can be calculated. This is done by traversing all the nodes starting from the *start node* and adding the *cost vector* through the *end node*. Thus after each optimization or operation of the graph, the cost filter associates a cost vector with the *end node* of the DFG. This, then reflects the total cost of the entire DFG. Decision to perform optimizations in a certain order may be made by the user by taking the product of the two components (area-delay). In some cases where the timing constraints are very stringent, the user may add some

weight to the communication cost, whereas, in cases where real estate on the chip is of prime importance, the user may add weight to the area component. The order of complexity for the cost filter is O(n), where n is the number of the nodes in the graph.

3.2.3 Constant Folding filter

The constant folding filter has its analog in conventional compilers. It performs *function-preserving transformations*. Function-preserving transformations optimize without changing the function they compute. Constant folding is the process of eliminating constants in computational expressions to reduce run-time costs. Figure 3.2.4 shows a DFG before and after it has passed through a constant folding filter. The unoptimized DFG would take 616 ns. to execute and would occupy 128 CLBs on the target architecture[4]. The optimized version would execute almost three times faster and occupy about one third the area, a total improvement by a factor of nine in the area-speed product.

[4] The mapping of functions onto CLBs was done by writing the code shown in Figure 3.2.4 in VHDL. The VHDL code was then synthesized using standard tools that take as input a hardware description and generate at the output, a file to configure the target platform.

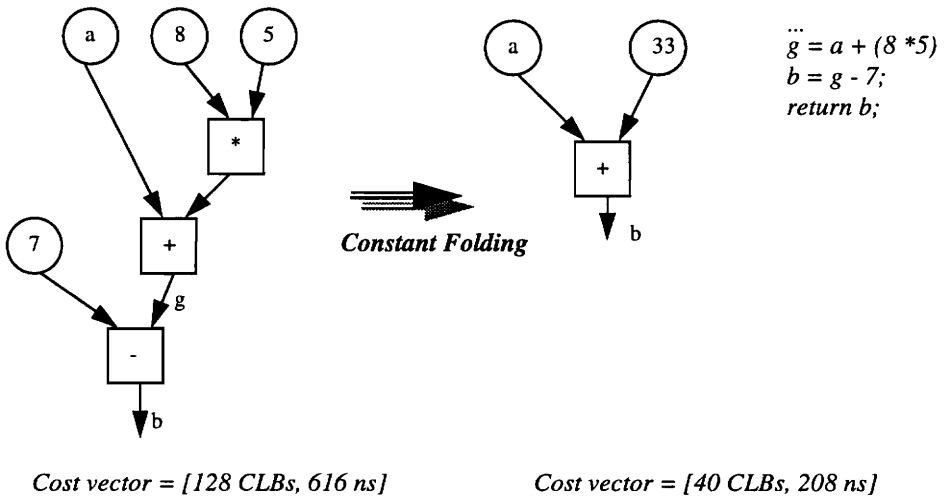


Figure 3.2.4. An example DFG before and after a passing of the constant folding filter.

The cost filter performs the reduction of the above expressions in two passes. In the first pass, the expression (8*5) is reduced to (40). In the second pass the expression (40 -7) is reduced to 33. It was possible to reduce the second expression because variable *g* was utilized only in the computation of *b*. If this was not the case, the destruction of (*g = a + 40)* would not be justified. The constant folding filter is iterated until no further optimizations are performed by the filter.

3.2.3.1 *Implementation*

The constant folding filter is implemented as a *template matching* filter. Template matching filters transform a specified template to an optimized one. Since a graph with n nodes has a maximum of n^2 edges, and template matching requires all the edges to be traversed the order of complexity for the constant folding filter is $O(n^2)$. Figure 3.2.5 lists all the templates to be matched in the constant folding step. It also displays the optimized

templates by their sides. *Template 1* shows the reduction of two constants and an operator node into a single constant node. *Template 2* reduces constants connected by the add and subtract operators into a constant node. For example, template 2 reduces expressions of the form *(g = a + 8 - 5)* to *(g = a + 3)*. *Template 3* reduces expressions of the form *(g = a * 8 * 9)* to *(g = a * 72)*. This filter also reduces trivial math expressions like the following:

$a \& 1 = a$,	$a \& 0 = 0$,	$a \& a = a$,
$a \mid 1 = 1$,	$a \mid 0 = 0$,	a | a = a,
$a + 0 = a$,	$a * 1 = a$,	$a * 0 = 0$,
$a - 0 = a$,	$a / 1 = a$,	NOT 1 = 0.
a ^ 0 = a,	a ^ a = 0,	NOT 0 = 1.

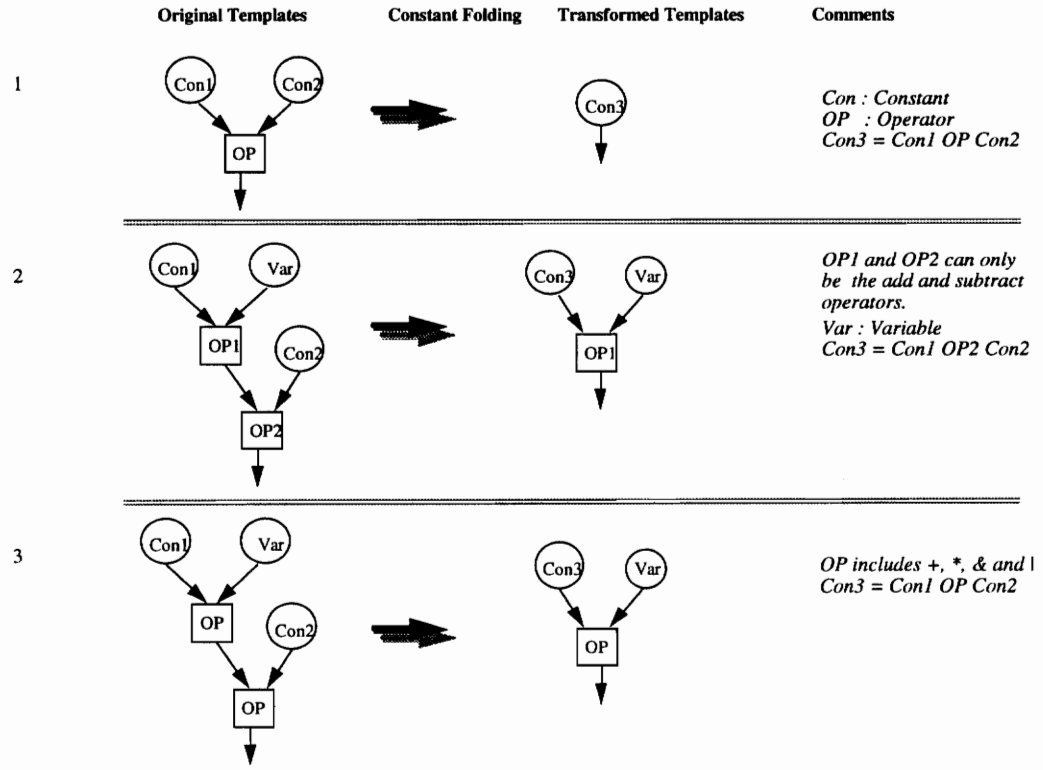


Figure 3.2.5. The original and the transformed templates for the constant folding filter.

3.2.4 Strength Reduction Filter

Reduction of strength refers to the replacement of an expensive operation by one with a lower cost. For example, multiplication in hardware can be replaced by a series of left shifts and additions. Multiplication in hardware is a very expensive operation, both in terms of resources and the time taken to compute it. Comparatively, a shift operation by a constant in hardware is very cheap, since no *functional units* are needed to perform the operation. The bit positions can be easily changed by connecting appropriate paths, which, reduces to an *interconnection* problem. Figure 3.2.6 shows an example pass of the *strength reduction* filter.

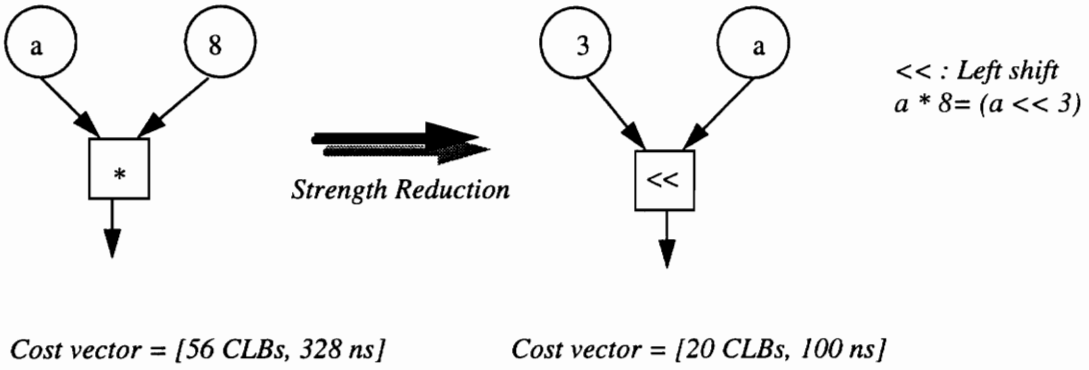


Figure 3.2.6. Strength Reduction: Example 1

As seen from the above figure, both the number of CLBs and the delay, are reduced by a factor of three, a total reduction by a factor of nine in the area-delay product. Consider Figure 3.2.7. There is no significant improvement in the number of CLBs or in the computational delay. The reason for this is the "incorrect" number of CLBs allocated to the shifter by the synthesis tool. Similarly, the reduction in the speed-area product

achieved earlier can be expected to be more than nine. A rule of thumb is to try and replace multiplication by a series of shifts and additions, whenever possible.

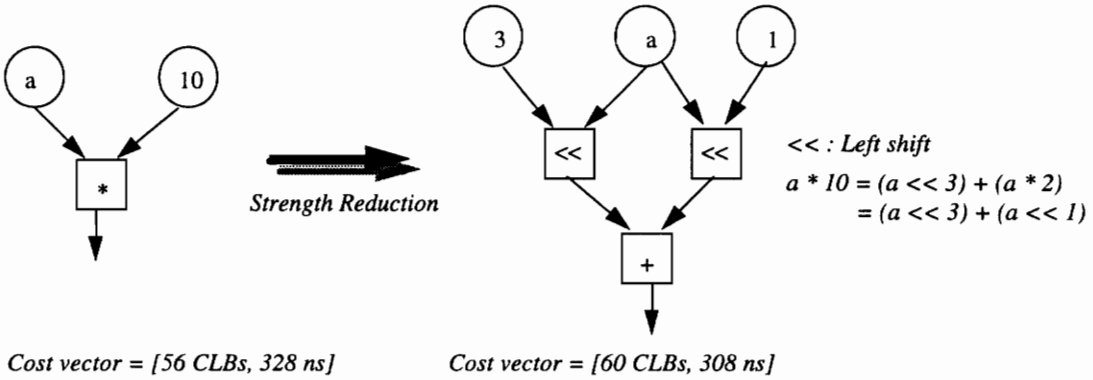


Figure 3.2.7. Strength Reduction: Example 2.

3.2.4.1 *Implementation*

The Strength Reduction filter, like the Constant Folding filter is implemented as a template matching filter and has the same complexity order as the constant folding filter. Figure 3.2.8 shows the original and the transformed templates.

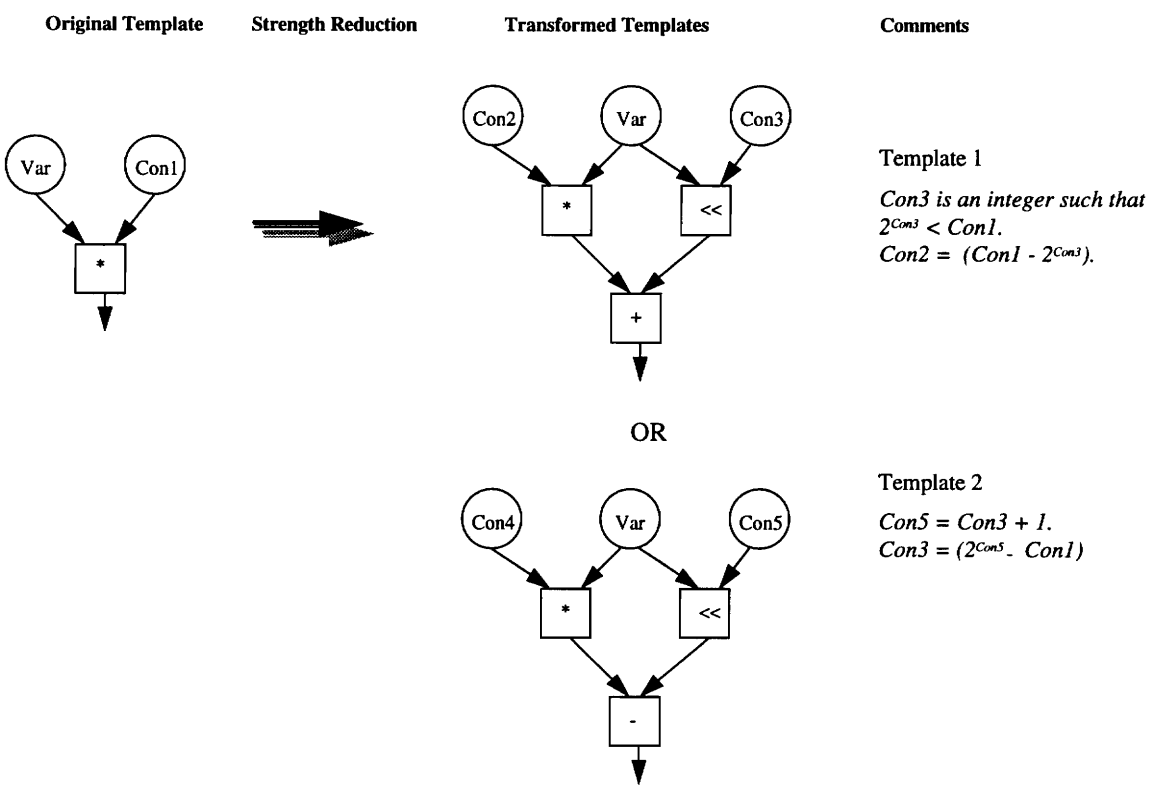


Figure 3.2.8. Strength Reduction Filter: Original and Transformed Templates.

Figure 3.2.8 shows two possible output templates, *template 1* and *template 2*. Depending on the values of the constants, *Con3* and *Con2*, the appropriate template is chosen. If *Con3* is less than *Con2*, then template 2 is chosen, else template 1 is selected. Multiple applications of this filter will reduce the multiplication node generated in the output template, into a series of left shifts. The output template is not a fully optimized one. A pass of the constant folding filter is necessary after this reduction to reduce multiplication by ones and zeros.

3.2.5 Dead-Code Elimination Filter

This filter removes computations that have no later dependencies. This way, computations that do not lend themselves to the final result are eliminated. A considerable saving in chip area is achieved due to the *non-instantiation* of functional units required for the computations, the results of which are never used. For example consider the following piece of code

```
int a = 0, b=5;
if (a != 0)
   b = a * 5;
x=b;
```

The computation of the variable *b* will never occur as the variable *a* is always equal to zero, but the computation will be present in the unoptimized DFG. The computation of *b* will be detected as dead-code and eliminated by this filter. The variable *x* will be assigned a value 5.

3.2.5.1 Implementation

The implementation of the dead-code elimination filter is carried out in two steps.

Step 1: A node that does not have any outgoing edge does not have any later dependencies. All such nodes are found and eliminated. This filter is a recursive one, i.e., removal of any single node in a pass always warrants another pass of this filter. This way the output DFG will be assured to be free of any dead-code. Step1 can be implemented in time O(n), where n is the number of nodes in the graph.

Step 2: The second step is a template matching step. Figure 3.2.9 illustrates the original and the optimized templates. As seen from the figure, if a conditional node has

two constants coming in as the inputs, then the condition can be evaluated at compile time and the result of the condition evaluation can hence be known. If this result is being used to make decisions, like in the case of the Select node, the appropriate branch can be taken at compile time itself. The implications of this optimization are a tremendous saving in terms of CLBs and execution time. The reason for this is that a functional unit which would otherwise have been instantiated to evaluate the condition, is not. Secondly, the need for the multiplexor is circumvented by an early branch prediction. Step 2 requires $O(n^2)$ time to be carried out.

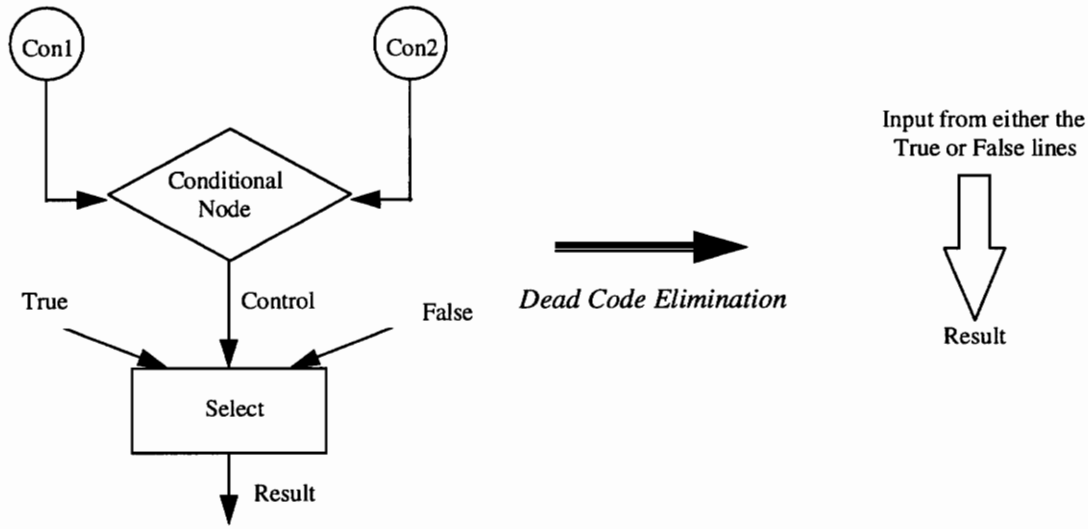


Figure 3.2.9. Original and optimized templates for dead code elimination.

3.2.6 Common Subexpression Filter

In software, an occurrence of an expression E is called a common subexpression if E was previously computed and the values of the variables in E have not changed since the previous computation. The objective of the common subexpression filter (CSE filter) is to avoid recomputation of expression E and to use the previously computed value of E. This results in a tremendous saving in hardware implementation because of the elimination of a functional unit to compute the expression E again. Figure 3.2.10 illustrates common subexpression elimination. As seen from the figure the time taken for the DFG to execute in both the cases is the same. This is because the delay is dominated by the longest path ($a + b + 5$) which is present in both the cases.

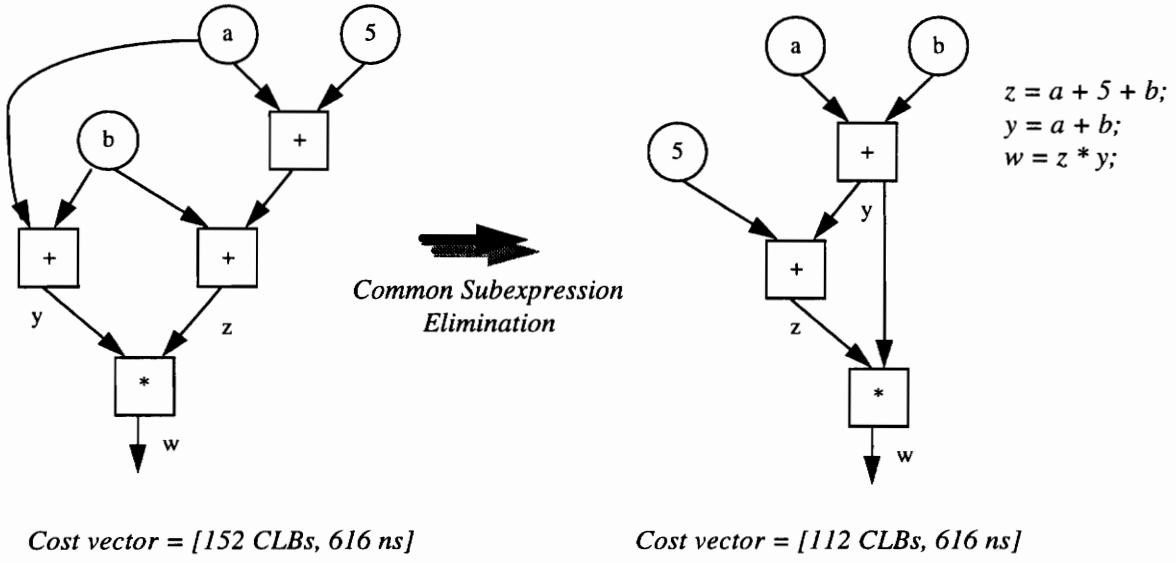


Figure 3.2.10. Common subexpression Elimination.

3.2.6.1 Implementation

Figure 3.2.11 illustrates the algorithm to find subexpressions. Whenever the algorithm comes across an operator node, it searches a symbol table for a node that performs the same operation. The symbol table has three fields, a label field and two other fields for the inputs of the node (in the case of *unary* operators the third field is made NULL). If such a node exists in the symbol table then fields two and three are compared to the inputs of the node in question. If the inputs and the fields match, a subexpression has been detected by the algorithm and the *Eliminate Subexpression* routine is called. This routine is illustrated in Figure 3.2.12. The routine *eliminates* the node and the input(s) (depending on whether the node is a binary or a unary one). Thus the subexpression has been eliminated successfully. The order of complexity of this algorithm is proportional to the number of edges, $e = n^2$ in the input graph.

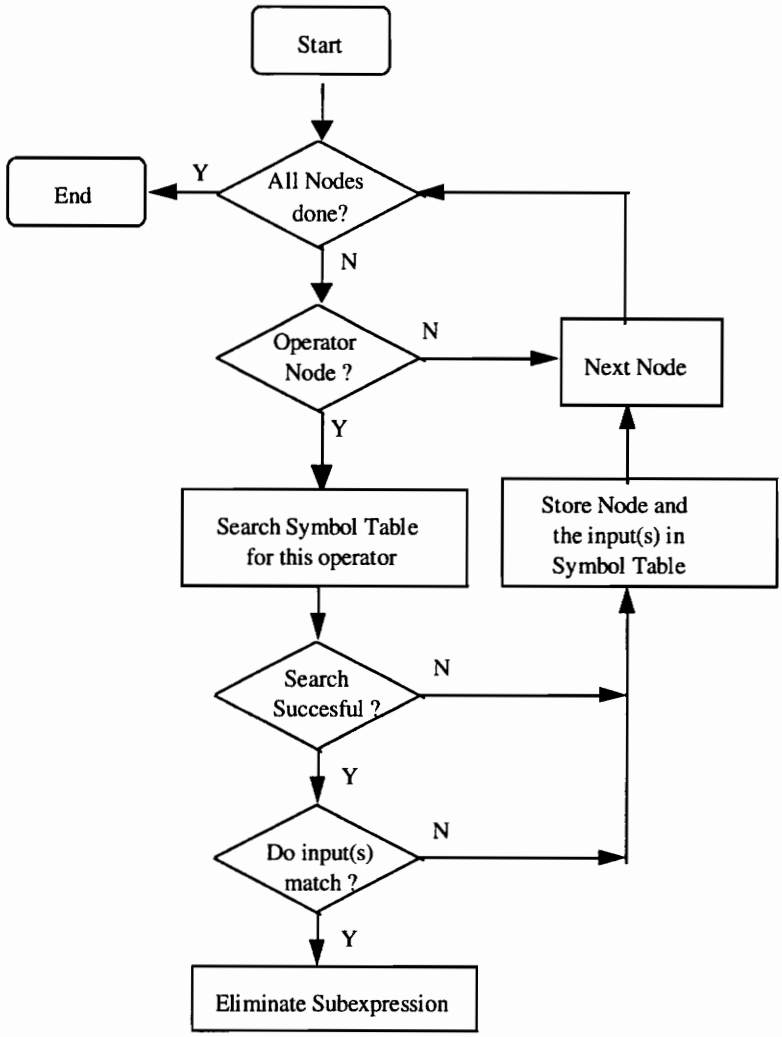


Figure 3.2.11. *Find Subexpressions*: Routine 1.

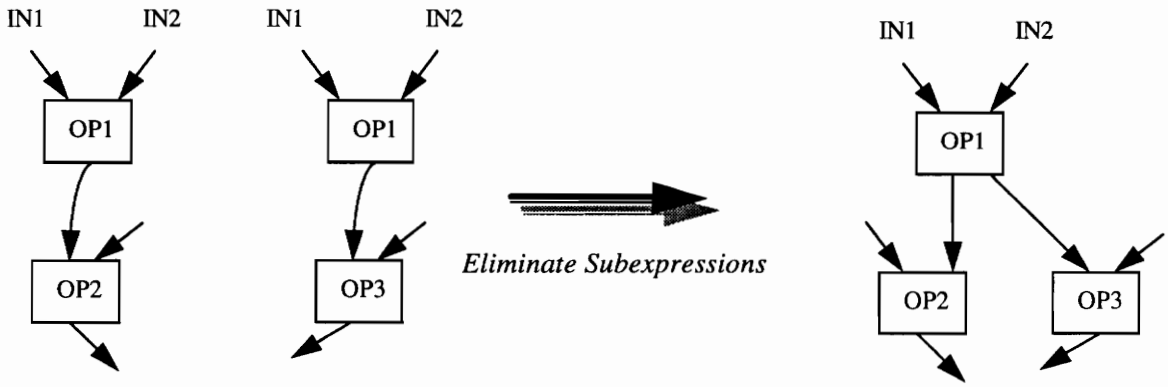


Figure 3.2.12. *Eliminate Subexpressions*: Routine 2

The algorithm of Figure 3.2.11 has an obvious drawback. Consider the following two statements:

$$d = a + c + b;$$
$$e = a + b;$$

Since the algorithm has only a *depth* of two, i.e., it can compare only two inputs of an operator node at a time, it cannot identify $(a + b)$ as a subexpression. If, on the other hand the expression for variable d was written as $(d = a + b + c)$, then the algorithm would be able to identify $(a + b)$ as a subexpression. The drawback described above can be overcome by adding Routine 3 (illustrated in Figure 3.2.13) to the subexpression filter. The execution of Routine 3 can be shown by taking the previous example of finding a common subexpression in

$$d = a + c + b;$$
$$e = a + b;$$

In the computation of variable d, the two addition nodes are linked by an edge and the order of the operations can be interchanged. Thus the criterion for swapping the inputs is satisfied and the computaton of variable d changes to $d = a + b + c$. As demonstrated earlier Routine 1 can easily find the subexpression $(a + b)$. For the computation of variable e there is no outgoing edge from the addition operator to any other operator, and hence Routine 3 is not executed for this particular computation. Routines 1, 2 and 3 can successfully find common subexpressions at any depth in the DFG.

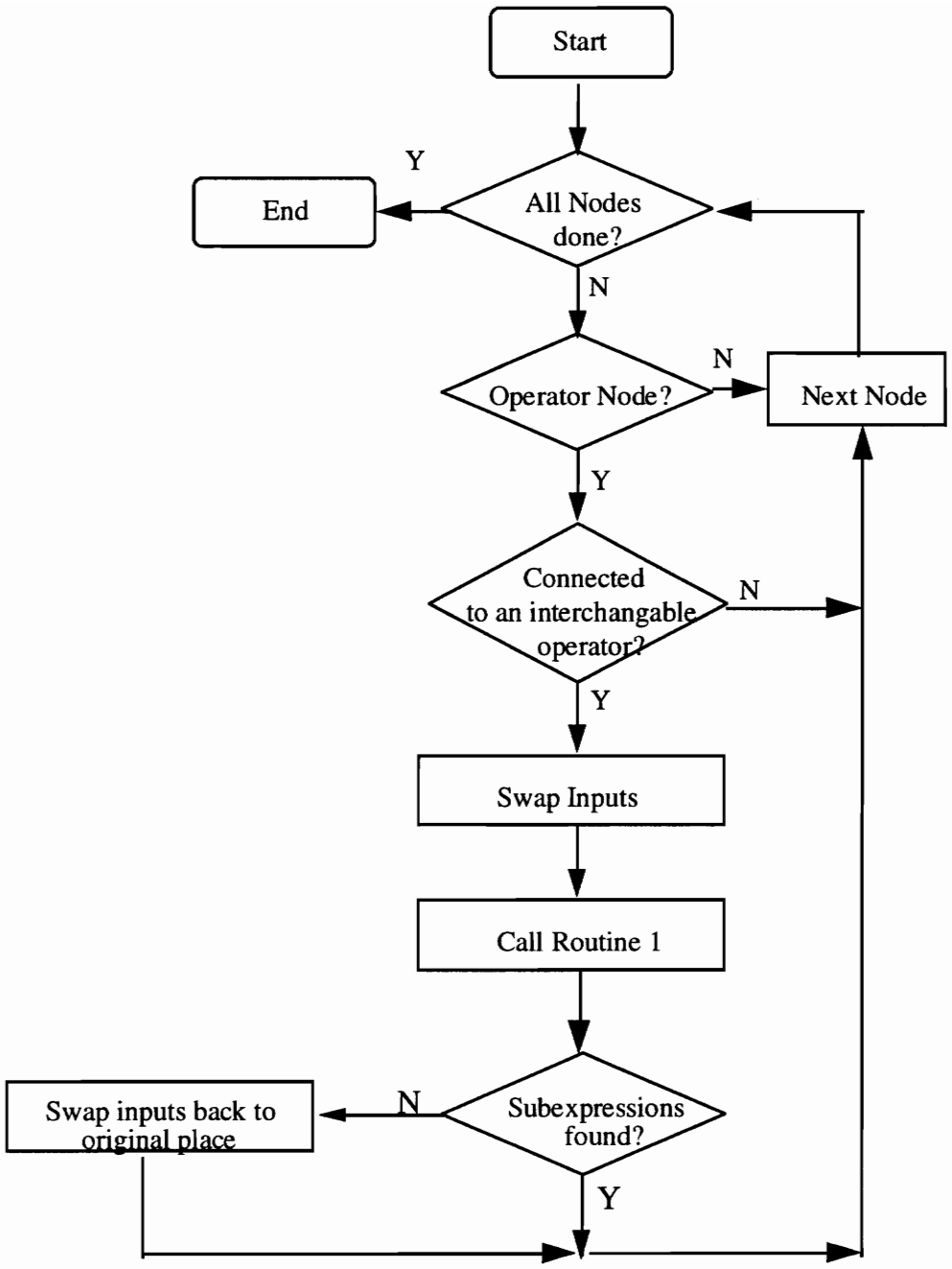


Figure 3.2.13. *Swap Interchangable Operators:* Routine 3.

3.3.7 Path Reduction Filter

Path reduction or *tree-height* reduction tries to achieve an expression split, such that the parallelism available in hardware can be exploited best. In the simplest form, the reduction algorithm uses the commutatity and associativity of addition and multiplication operations. Consider the computation of the variable *e* as illustrated below.

$$e = (a + 5) * 6$$

The same computation can be performed as

$$e = (a * 6) + 30$$

It can be seen from the above that in the first case the multiplication has to be *scheduled* after the addition operation while in the second case it can be scheduled earlier. In one of the earlier reductions, the strength reduction filter transformed multiplications to a series of adds and left shifts. The above computation can be further broken down to

$$e = (a << 2) + (a<<1) + 30$$

The above computation exposes the expressions *(a * 2)* and *(a * 4)* which could be potential subexpressions in other computations using the variable *a*. The path reduction filter does not directly bring down the computation cost but helps the following filters to do so.

3.3.7.1 Implementation

The path reduction filter, like some of the previous filters is also implemented as a template matching filter. Figure 3.2.14 shows the original and transformed templates.

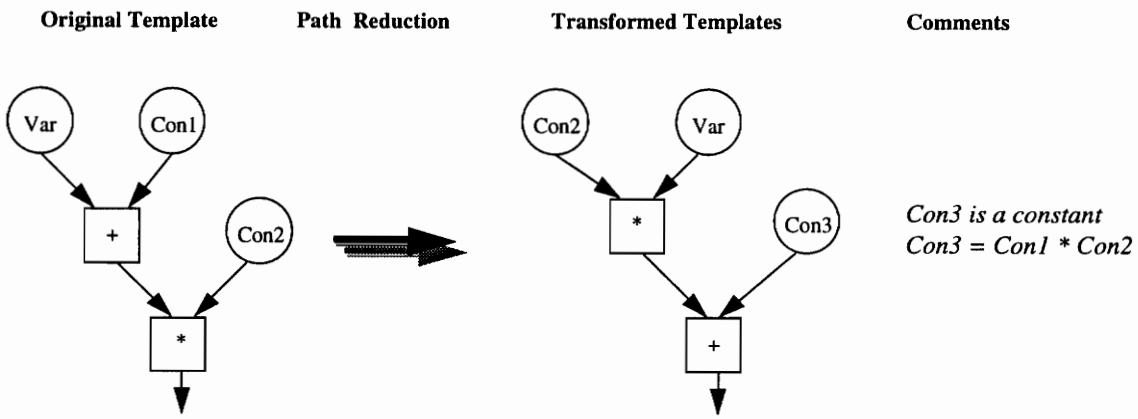


Figure 3.2.14. The original and transformed templates for the path reduction filter.

3.3 Reading and writing Xgrab files

Since every filter is a discrete block, it would be required to have a capability to read and write a dataflow graph (DFG) in a format which the next filter can interpret. As stated earlier, the format used here is that of Xgrab. The graph file of Figure 2.3.13 can be represented in the following manner:

NAME
{name of the function}
NODES
{node listing)
EDGES
{connections between the nodes}

The *node listing* lists all the nodes with their attributes, and the *connections* specify the edges between the nodes along with their attributes. Writing the dataflow graph to a Xgrab file is very straightforward. The *data structures* used for storing the DFG are

written in the above format to an output file. This is accomplished by a simple C routine. Graph reading is described in detail in the next section.

3.3.1 Graph reading

This section describes in brief the procedure to read a file given in the Xgrab format. The *reader* like the front end is composed of a lexer and a parser. As described earlier, the lexer breaks the input into a series of *tokens* and the parser contains rules to recognize these tokens and perform *actions* on their recognition. The intent here is not to describe the Lex and Yacc code, but to give a big picture of the *rules* that interpret the input Xgrab file in a dataflow format. Shown below are a set of simple rules which accomplish the above task of parsing.

Rule 1. DFG: (*NAME*) (name of function) (nodelist) (*EDGES*) (edgelist)

Rule 2. nodelsit:	(nodelist) (node)	{actions}
	| (node)	{actions}
Rule 3. edgelist:	(edgelist) (edge)	{actions}
	| (edge)	{actions}
Rule 4. node:	(node name) (attributes)	{actions}
Rule 5. edge:	(edge name) (attributes)	{actions}

When the example file of Figure 2.3.13 is presented as input to Lex, it is broken into a series of tokens that the above rules can interpret. For example, Rule 1 states that DFG (dataflow graph) is made up of nodes and edges, the two of them duly separated by identifiers. Rules 2 and 3 are recursive rules and specify *nodelist* and *edgelist* to be a list

of nodes and edges respectively. The parser can recognize infinite instances of nodes and edges using Rules 2 and 3. Rule 4 states that a node is made up of a *node name* and an *attribute*, while Rule 5 states the same for an edge. Symbols that are actually returned by the lexer as tokens are called *terminal* symbols. Examples of some terminal symbols in the above rules are node name, edge name and attributes. Symbols that appear on the left-hand side of some rule are called *non-terminal* symbols. Nodelist, edgelist, node and edge are some non-terminal symbols specified in the above rules. The above rules are also referred to as the *grammar*. Every time a grammatical rule is satisfied, the actions at the end of the rule are executed. The actions perform the task of transforming the input program into DFG format.

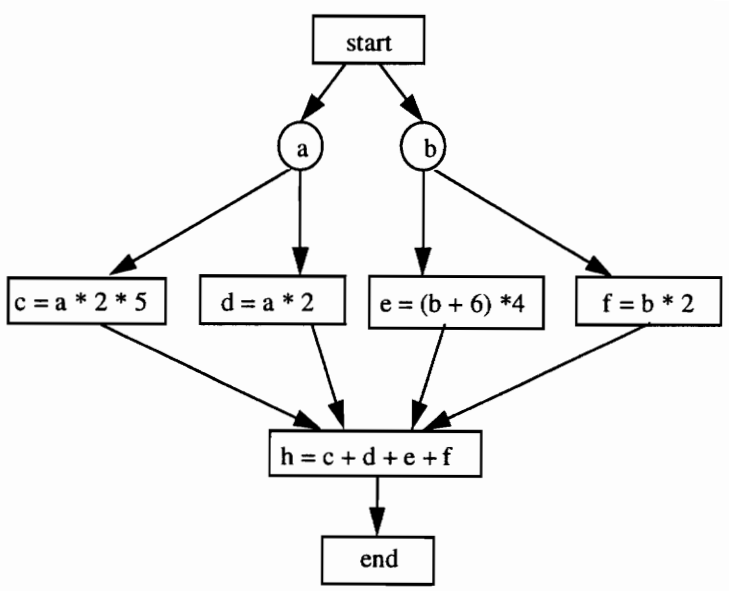
Chapter 4

Results

This chapter reports on the performance of PRISM-2. In the first section, a high-level language (HLL) specification and the corresponding dataflow graph (DFG) representation are presented. The DFG resulting from such a HLL specification is *unoptimized.* Various reductions, based on the *filter philosophy* discussed earlier, are performed on the DFG to optimize its temporal and spatial properties. The effect of *ordering* the filters on the final optimized DFG is then investigated. In the second section, the concept of *speedup* is introduced. Execution time of example programs executed entirely in software is compared to the time to execute them in a hardware-software domain determined by the PRISM-2 compiler.

4.1 Optimization of dataflow graphs

Figure 4.1.1 shows example HLL code and the corresponding DFG representation. This DFG is produced by the front-end of PRISM-2. The rest of the section is devoted to the methodology for optimizing the DFG shown in Figure 4.1.1.




```
function_test (a , b)
int a, b;
        {
        int c, d, e, f, h;
        c = a * 2 * 5;
        d = a * 2;
        e = (b + 6) * 4;
        f = b * 2;
        h = c + d + e + f;
        return h;
        }
```

Figure 4.1.1. An example HLL code and the corresponding DFG.

Before describing the methodology, a summary of implemented *filters* are listed below.

1. Flattening filter or the flattener
2. Cost filter
3. Constant folding filter
4. Strength reduction filter
5. Dead-code elimination filter
6. Common sub expression filter
7. Path reduction filter

There are a few constraints placed on the filter philosophy of Figure 3.2.2. They are as follows:

1. As explained in Section 3.2.1, the graph has to pass through the flattener before any reductions are applied to it.

2. The first reduction filter applied after the flattener should preferably be the constant folding one. This would eliminate operations performed on constants and hence avoid further reduction on such operations. For example, *(5 * 8)* can be reduced to *(40)* in the first pass. Instead, if the first pass was that of strength reduction, then the constant *5* would be shifted left by *3* bits, instantiating a shift operator. It may be possible in some cases for a certain reduction step to produce a better *cost function* than the constant folding one; hence to override the "best approach" block in the first pass this constraint is placed.

3. The strength reduction filter is always followed by the constant folding filter and the cost function of the strength reduction filter considers both the optimizations. This is because the strength reduction filter introduces numerous multiplication by ones and zeros. For example, the expression *(a * 9)* is reduced to *((a << 3) + (a * 1))* by the strength reduction filter. It is the constant folding filter that reduces *(a * 1)* to *(a)*, which then gives the correct effect of strength reduction. If this constraint is not followed, the cost function presented after strength reduction will always be greater than that before the reduction. This would mean that the "best approach" block will never allow this reduction to take place.

4. The path reduction filter is always followed by the subexpression elimination filter. The reason for this is that the path reduction filter merely *exposes* the underlying common expressions. It is the common subexpression filter that eliminates them.

5. Some filters may produce redundant code in the process of optimization. To eliminate the redundancies every filter has a dead-code elimination filter *imbedded* within it.

With the above constraints in mind, and adhering to the filter philosophy, the rest of the section presents the steps to produce an optimized DFG from the one illustrated in Figure 4.1.1.

Step 1. The graph of Figure 4.1.1 is given to the flattener. The graph produced by the flattener is illustrated in Figure 4.1.2. The cost filter assigns a cost vector, [432 CLBs, 1280 ns] to the above DFG.

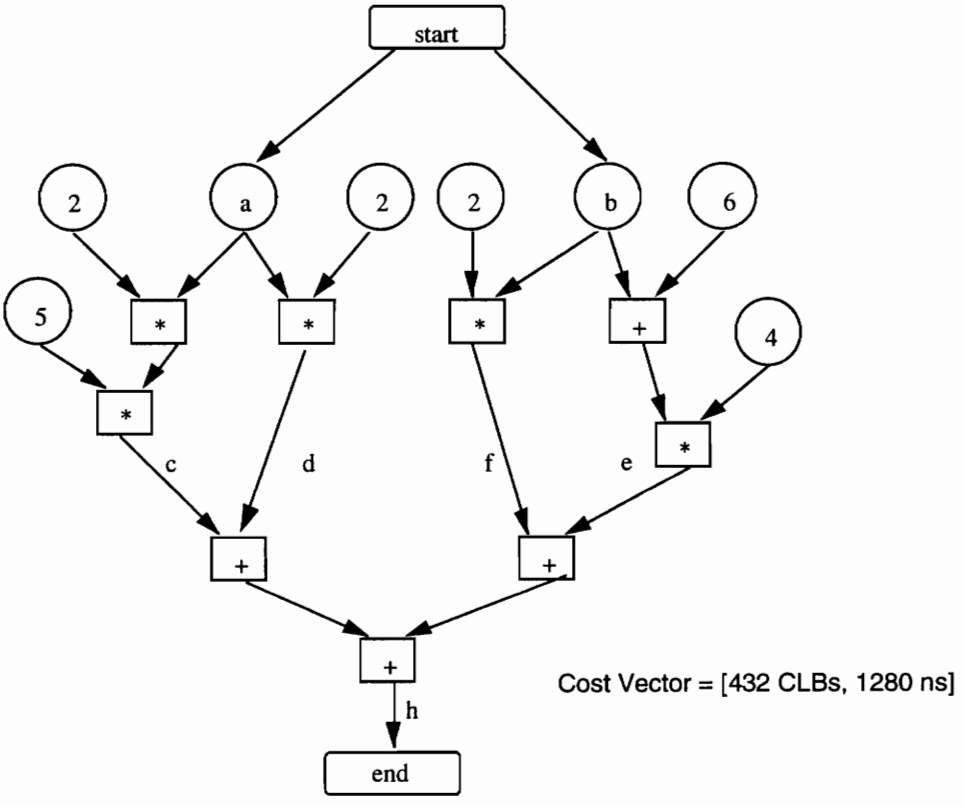


Figure 4.1.2. The flattened version of the DFG illustrated in Figure 4.1.1.

Step 2. Constraint 2 specifies the next reduction to be constant folding. A partial DFG, illustrating the modification to Figure 4.1.2, is shown in Figure 4.1.3. The only modification is in the computation of variable *c*. The cost filter determined a cost vector, [376 CLBs, 952 ns] to execute the modified DFG.

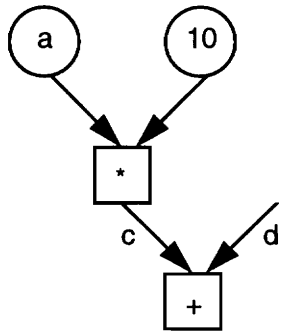


Figure 4.1.3. Partial DFG of Figure 4.1.2 after constant folding reduction.

Step 3. Various reduction filters listed earlier in the section are applied to the DFG of Figure 4.1.3. The results of applying the various reductions are listed in Table 4.1.1 in terms of the number of CLBs and the corresponding execution times.

Table 4.1.1. Various optimizations during Step 3.

No.	Optimization	Number of CLBs	Execution time
1	Path reduction and Common subexpression	376	952
2	Common sub expression	376	952
3	Strength reduction and Constant folding	292	857

In Table 4.1.1, Row 1 illustrates constarint 4 and Row 3 depicts constraint 3. To give a feel for figures, if Step 3 consisted of only strength reduction, the number of CLBs would be 708 and the total execution time would be 1368 ns! From the above Table, the "best approach" block of Figure 3.2.2 chooses Step 3. The resulting DFG is shown in Figure 4.1.4.

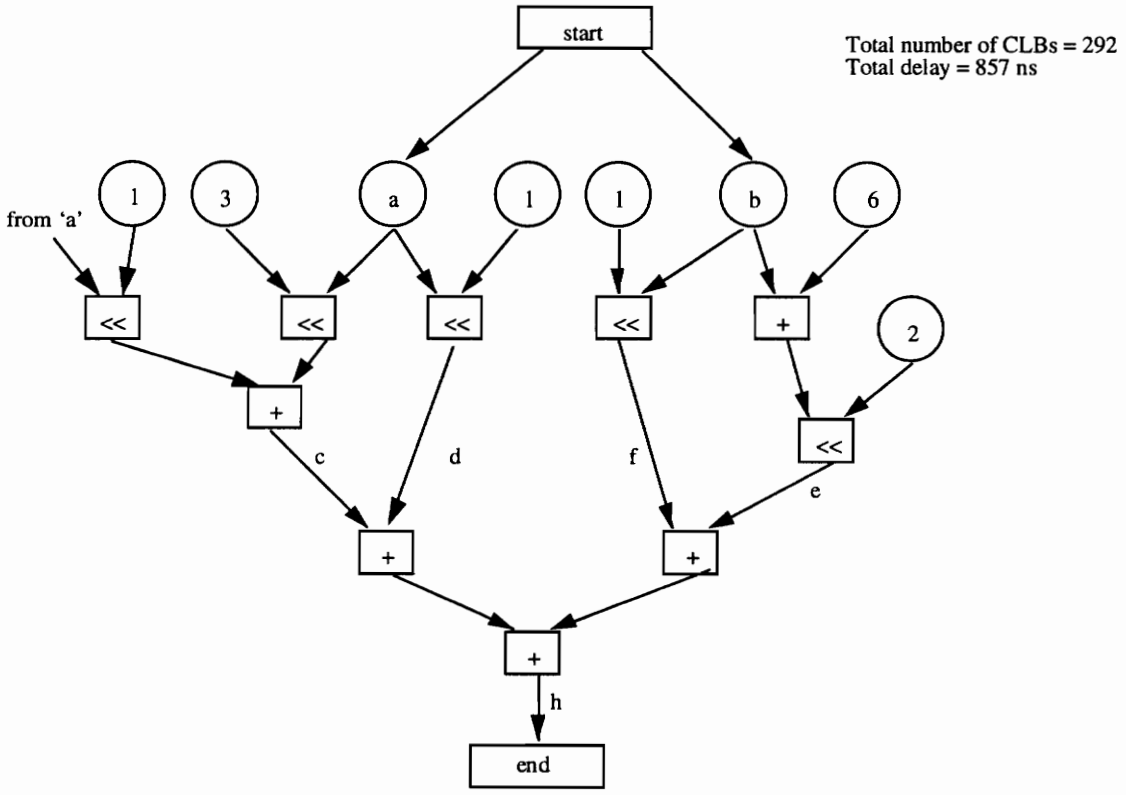


Figure 4.1.4. The resulting DFG after Step 3.

Step 4. This is similar to Step 3. Table 4.1.2 shows the results of Step 4 in a similar fashion. As expected, Row 2 is selected as the best optimization step.

Table 4.1.2. Various optimizations during Step 4.

No.	Optimization	Number of CLBs	Execution time
1	Path reduction and Common subexpression	292	857
2	Common sub expression	272	857
3	Strength reduction and Constant folding	292	857

Step 5. The result of any further optimizations yields the same results depicted in Table 4.1.2. Thus the output graph is indeed an optimized DFG.

As noted from Table 4.1.2, the number of CLBs has fallen from 432 to 272, a 37% reduction. The actual reduction is slightly more because the shift operator, as noted in Section 3.2.2, occupies fewer number of CLBs in practice. The execution time has decreased from 1280 ns to 952 ns, a 10% reduction. The final optimized DFG is shown in Figure 4.1.5.

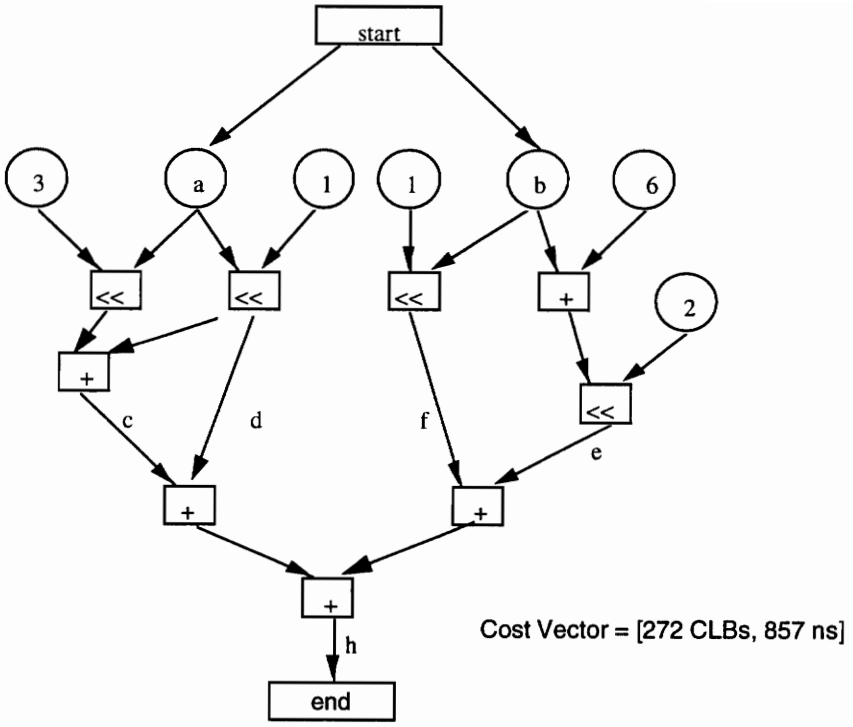


Figure 4.1.5. The final optimized DFG.

4.2 Comparison of execution times

In this section, the time taken by a conventional compiler[1] to execute the HLL code is compared to the time it would take for the corresponding DFG to execute on the reconfigurable architecture of PRISM-2. The *scoring step* that assigns the cost function to the DFG gives the second term. The first term is determined through standard utilities like the *time*[2] and *profile* commands. The ratio of the two *times* is referred to as the *speedup*. The assumption made here is that the entire DFG can be synthesized on the reconfigurable platform.

---

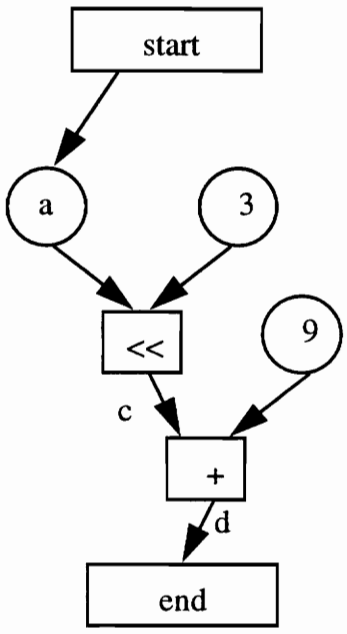
[1] The compiler used here was the GCC, version 2.6.0 provided by GNU.
[2] These are standard UNIX utilities.

$$\text{Speedup} = \frac{\text{Time taken by a conventional compiler to execute the HLL code}}{\text{Time taken by the DFG to execute on PRISM - 2}}$$

Two example HLL programs and their corresponding optimized DFGs, along with the execution time for both the cases, are presented. Figure 4.2.1 illustrates a HLL program and the corresponding DFG. The DFG representation required 52 CLBs for synthesis and 225 ns for execution. The HLL program was then compiled and executed on a Sun SPARC2 station. To give a feel for time, the program was run for 10^7 iterations. The execution time was found to be 12.65 s.

$$\text{Speedup} = (12.65) / (225 \times 10^{-9} \times 10^7) = 5.62$$

Thus, PRISM-2 was able to speed the program execution by almost five times. The above example HLL code was purely linear and hence the low speedup can be attributed to the absence of any inherent parallelism.




```
int test (a )
int a;
 {
 int c, d;
 c = a * 8;
 d = c + 9;
 return d;
 }
```

Figure 4.2.1. An example HLL program and the corresponding DFG.

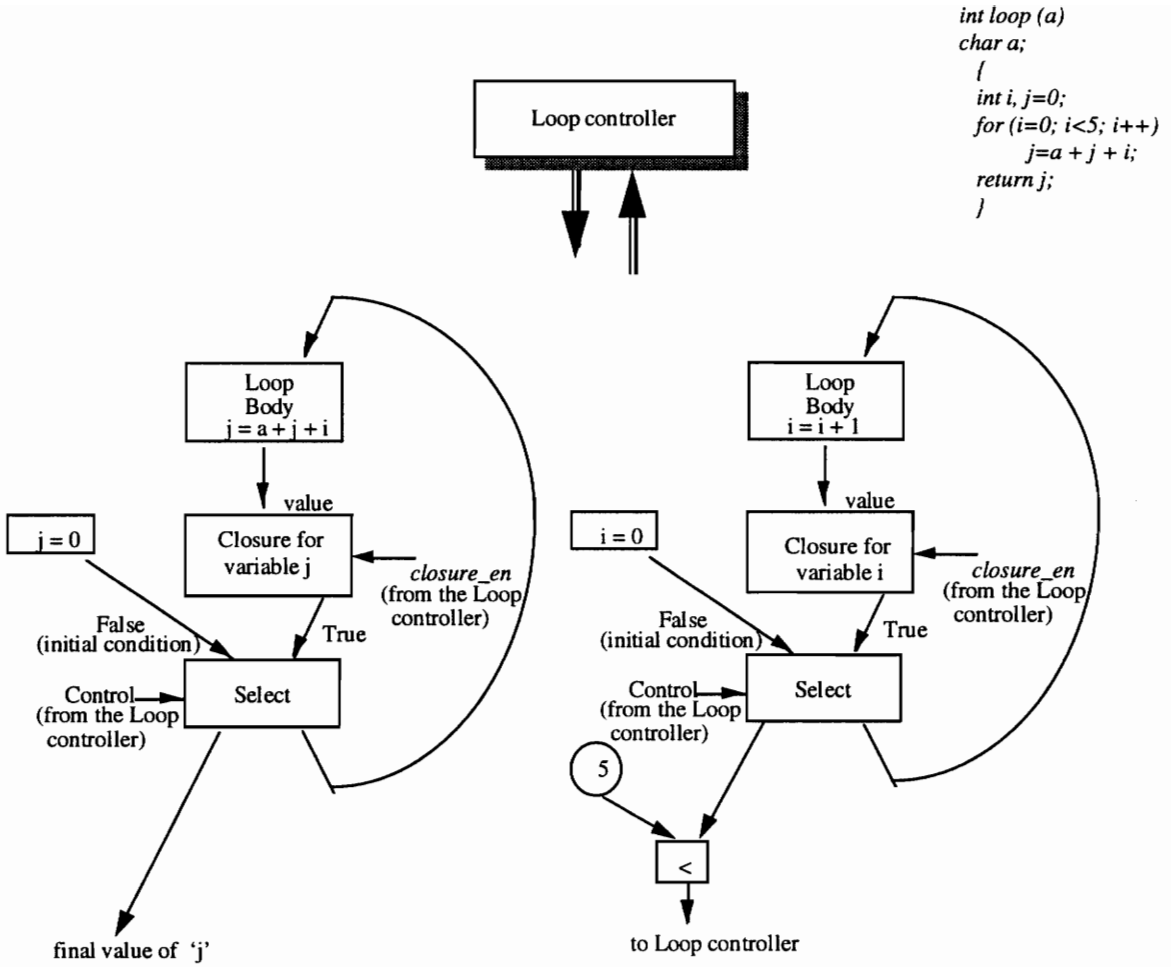


Figure 4.2.2. A HLL program and the corresponding DFG illustrating a loop.

Figure 4.2.2 illustrates a HLL program consisting of a loop. For simplicity only a high-level picture of the DFG is shown. The complete representation is shown in the Appendix. Even though loop optimizations are not covered, it would be worthwhile to see the speedup in this case. The HLL code when executed on the Sun SPARC 2

platform took 1.86 s for 10^6 iterations. The corresponding DFG took 34 CLBs and 72 ns for a single iteration. Hence,

$$\text{Speedup} = (1.86) / (72 \times 10^{-9} \times 10^6) = 5.1$$

It is to be noted that the above speedup was achieved even in the absence of any loop optimization.

Chapter 5

Future Directions

The construction of the PRISM-2 platform is one of the most ambitious and interesting areas of research at Virginia Tech. Most of the blocks of PRISM-2 are currently operational. The following blocks of Figure 2.2 need to implemented or improved upon:

1. Graph Reductions: Only linear optimizations are currently operational. The *loop optimization filter* needs to be implemented. Once operational, the output graph will truly be an optimized one, and suited for hardware synthesis.
2. Partitioning and Scheduling: This block is currently under investigation and will be operational soon. The rest of the section outlines future directions for the project. As it shall be seen, the future directions are mostly *technology dependent.*

The incorporation of field programmable gate arrays (FPGAs) in the architecture, makes PRISM-2 dependent on the advances in *FPGA technology*. As FPGAs get bigger and faster, increasingly complex functions shall become fully synthesizable. The execution time of these functions will fall dramatically. Currently, the XC4010 series of FPGAs in use, have 400 CLBs. The recently introduced XC4025 series has 1024 CLBs. The

maximum clock rate of a 16x16 adder for different *speed grade* FPGAs is shown in Table 5.1[1]. Table 5.1 shows an increase of almost 25% in the clock rates within the same family.

Table 5.1. Maximum clock rate for different speed grade FPGAs of the XC4010 family.

Component-Speed Grade	Maximum Clock Rate
XC4010-6	10.6 MHz
XC4010-5	13.1 MHz
XC4010-4	14.4 MHz

As the complexity of FPGA devices increase, high reconfiguration costs will decrease and they will be able to support *multitasking* and *context switching*[2], at least in part. The communication latency between the reconfigurable board and the general purpose one will decrease as faster bus architectures are introduced. This in turn will reduce the program execution time. With all the above mentioned features incorporated in the future FPGAs, reconfigurable platforms will truly come of age.

[1] This data is based on the *xdelay* tool provided by Xilinx. The component was changed in the .lca file and xdelay was run for every component.

[2] The area of multiple-context FPGAs is an interesting one. A new architecture called the Dynamically Programmed Gate Array (DPGA) is under investigation. Refer [26] for further information.

Bibliography

[1] John L. Hennessey and David A. Patterson. *Computer Architecture: A quantitative approach.* Morgan Kaufmann Publishers, San Mateo, California, 1990.

[2] Daniel Gajski, Nikil Dutt, Allen Wu and Steve Lin. *High-Level Synthesis: Introduction to chip and System Design*, Kluwer Academic Publishers, Boston, Massachusetts, 1992.

[3] Kai Hwang and Faye. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, 1984.

[4] Peter M. Athanas. *An adaptive Machine Architecture and Compiler for Dynamic Processor Reconfiguration.* Technical Report LEMS-101, Brown University, Providence, Rhode Island, 1992.

[5] G. Estrin, "Organization of Computer Systems: The Fixed-Plus Variable Computer," Proc. Western Joint Computer Conf., Am. Inst. Electrical Engineers, New York, 1960, pp. 33-40.

[6] T. G. Rauscher and A. K. Agarwala, "Dynamic Problem-Oriented Redefinition of Computer Architecture Via Microprogramming," *IEEE* Trans. Computers, Vol. C-27, No. 11, pp. 1,006-1,014, Nov. 1978.

[7] *The Programmable Gate Array Data Book*, Xilinx, San Jose, California, 1991.

[8] M. Gokhale et al., "Building and using a Highly Parallel Programmable Logic Array," *Computer*, Vol. 24, No. 1, pp. 81-89, Jan 1991.

[9] P. Bertin, D. Roncin, and J. Vullemin, "Introduction to Programmable Active Memories," Tech. Report No. 3, Digital Equipment Corporation-Paris Research Laboratory, June 1989.

[10] M. Gokhal et al., *SPLASH: A Reconfigurable Linear Logi Array*, Tech. Report SRC-TR-90-012, Supercomputer Research Center, Institute for Defense Analyses, Bowie, Maryland, 1990.

[11] D.E. Van den Bout et al., "Anyboard: An FPGA-Based, Reconfigurable System," *IEEE Design & Test of Computers*, Vol. 9, No.3, pp.21-30, Sept. 1992.

[12] D. Gajski. *Silicon Compilation*, Addison-Wesley, Reading , Massachusetts 1988.

[13] A. V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[14] G. De Micheli, *Synthesis and Optimizations of Digital Circuits*. McGraw-Hill Book Company, 1994.

[15] Robert A. Iannucci. *Parallel Machines: Parallel Machine Languages*. Kluwer Academic Publishers, Boston, Massachusetts, 1990.

[16] Micah Beck, Richard Johnson, and Keshav Pingali. "From Control Flow to DataFlow." *Journal of Parallel and Distributed Computing,* vol. 12, pp. 118-129, 1991.

[17] Richard Johnson, Wei Li and Keshav Pingali. "An Executable Representation of Distance and Direction." *Internet Posting.*

[18] W. B. Ackerman, "Data Flow Languages,"*IEEE Computer*, Vol. 15, No. 2, pp 19-25, 1982.

[19] Richard Johnson and Keshav Pingali. "Dependence-Based Program Analysis," *ACM SIGPLAN'93* PLDI, June 1993

[19] Alex Orailoglu, Daniel D. Gajski, Flow Graph Representation, *23rd IEEE Design Automation Conference*, pp. 503-509, 1986.

[20] J. R Armstrong. *Structured Logic Design with VHDL*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[21] L. F. Chao, A. LaPaugh, and E. H. M Sha, "Scheduling Cyclic Data-Flow Graphs by retiming with resource constraints," *ACM/IEEE Sixth International Workshop on High Level Synthesis*, Dana Point, California, November 1992.

[22] L. F. Chao and E. H. M Sha, "Retiming and Unfolding Data-Flow Graphs," *Proc. International Conference on Parallel Processing*, St. Charles, Illinois, August 1992.

[23] C-T. Hwang, J-H. Lee and Y-C. Hsu, "A Formal approach to the Scheduling Problem in High-Level Synthesis," *IEEE Transactions on CAD/ICAS*, Vol. CAD-10, No.4, pp.464-475, April 1991.

[24] R. Potasman, J. Lis, A.Nicolau and D.Gajski, "Percolation Based Scheduling," *DAC, Proceedings of the Design Automation Conference*, pp. 444-449, 1990.

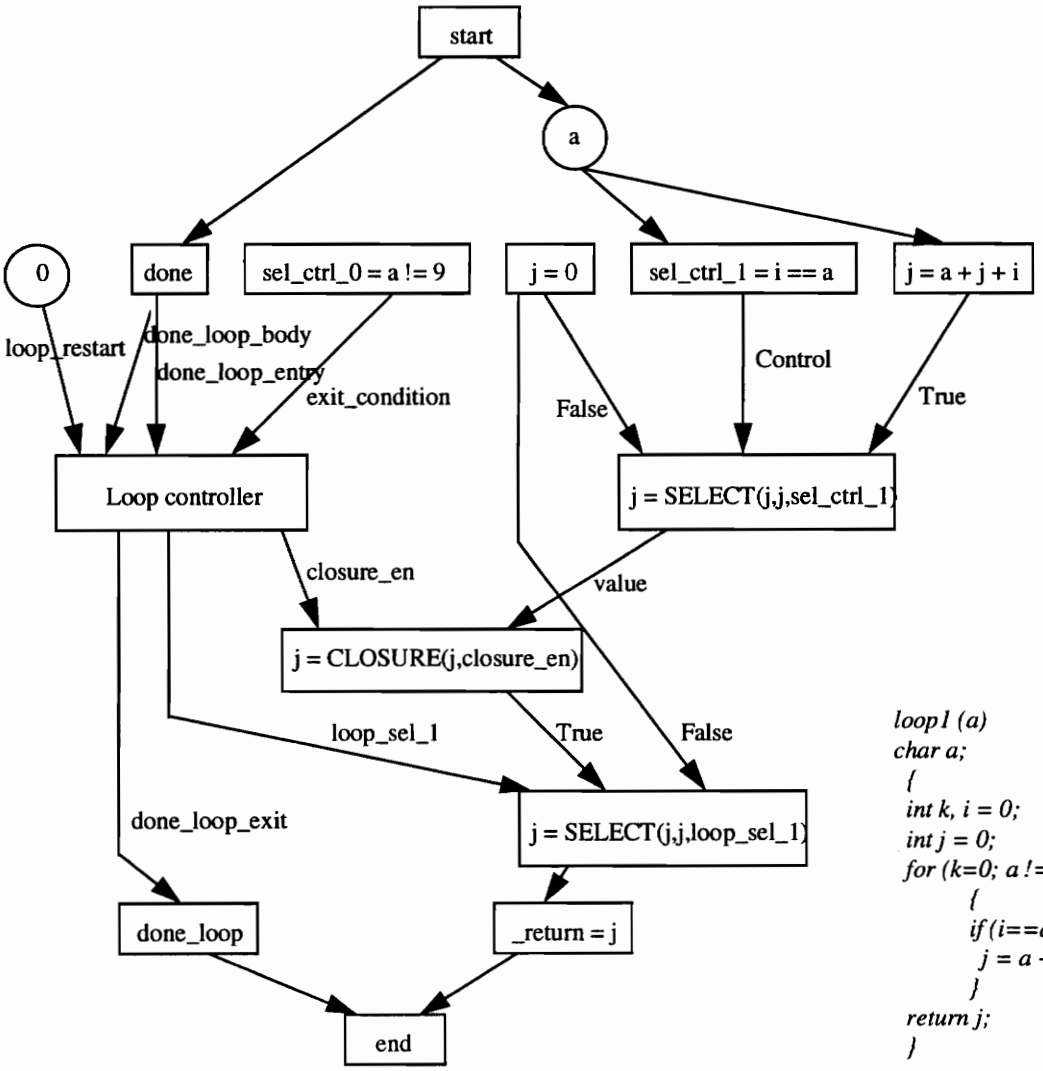
[25] R. Compasano, "From Behavior to Structure: High-Level Synthesis," *IEEE Design and Test of Computers*, October 1990.

[26] Andre DeHon, "DPGA- Coupled Microprocessors: Commodity ICs for the Early 21st Century," *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa Valley, California, April 1994.

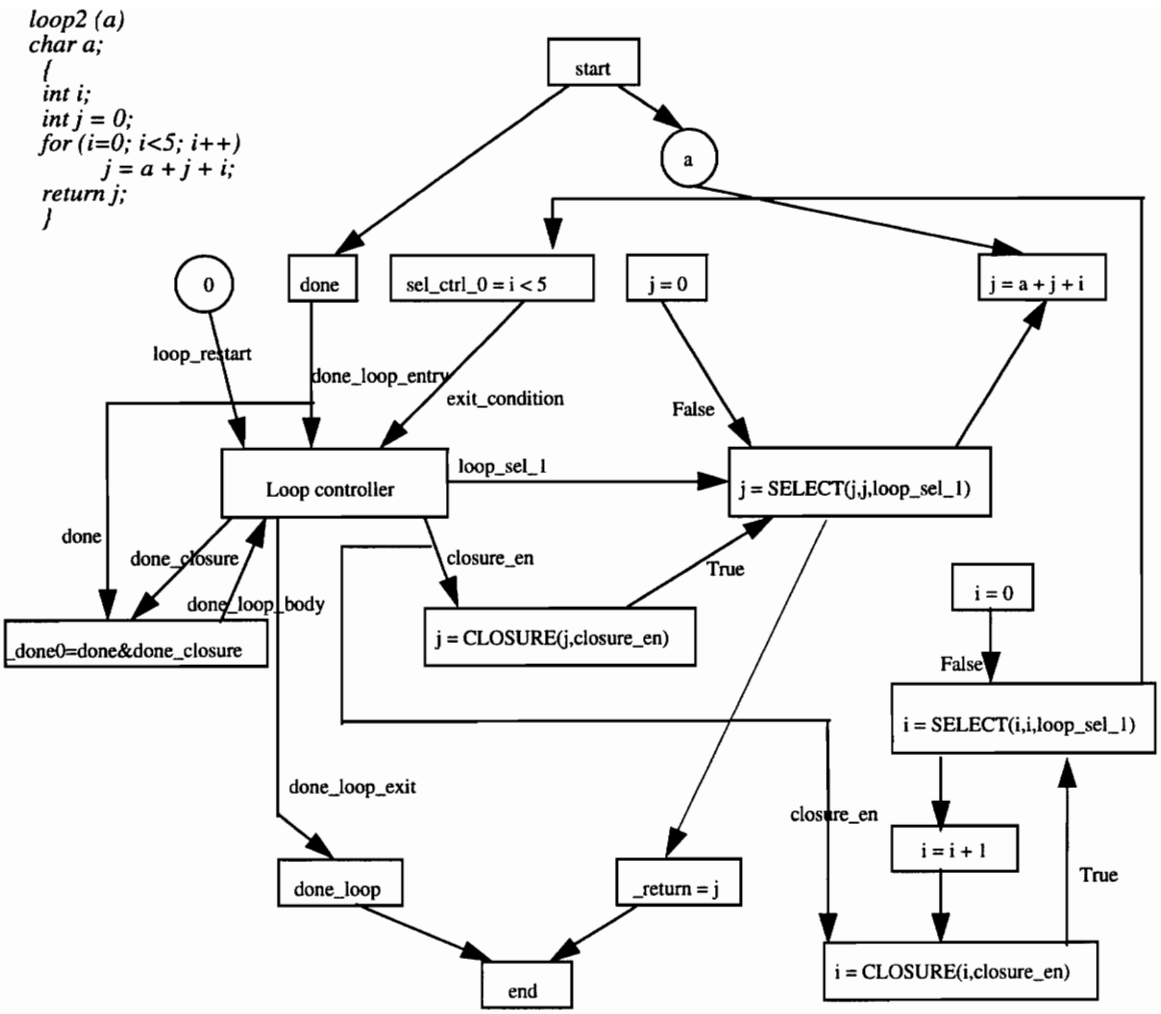
[27] M. S Lam and R. P Wilson, "Limits of Control Flow on Parallelism," *ACM/IEEE International Symposium on Computer Architecture*, May 1992.

Appendix

Example dataflow graphs




```
loop1 (a)
char a;
 {
 int k, i = 0;
 int j = 0;
 for (k=0; a != 9; k++)
        {
        if(i==a)
         j = a + j + i;
        }
 return j;
 }
```


loop2 (a)
char a;
{
int i;
int j = 0;
for (i=0; i<5; i++)
j = a + j + i;
return j;
}
start
a
done
sel_ctrl_0 = i < 5
j = 0
j = a + j + i
0
loop_restart
done_loop_entry
exit_condition
False
Loop controller
loop_sel_1
j = SELECT(j,j,loop_sel_1)
done
done_closure
done_loop_body
closure_en
True
i = 0
_done0=done&done_closure
j = CLOSURE(j,closure_en)
False
i = SELECT(i,i,loop_sel_1)
done_loop_exit
closure_en
i = i + 1
True
done_loop
_return = j
end
i = CLOSURE(i,closure_en)

Vita

Quaeed Motiwala was born in the city of Bombay, India on the 17th of October 1970. He went to the Jai Hind Institute, Bombay for his high school education. After completing his high school education, he received his Bachelors in Electronics Engineering (B. E) from the Vivekanand College of Engineering, Bombay in May 1992. To achieve his goal of a Master's, little realizing the sleepless nights that lay ahead of him, he decided to pursue his Master's in Electrical Engineering at Virginia Tech, Blacksburg, Virginia, and consequently came to Tech in Fall 1992. During his first semester at Virginia Tech he was fascinated by the VLSI Design course and decided to make chip design his career (although, manufacturing tortilla chips would have been easier). He received his Master's in September 1994.

His hobbies include swimming, cooking exotic food, and traveling to new places though, day-dreaming is one of his favorites. His favorite past time was bullying his younger sister who now studies in Bombay. He is currently a member of IEEE and was elected to the Honor Society of Phi Kappa Phi in August 1993.