View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Kentucky

University of Kentucky
UKnowledge

University of Kentucky Master's Theses Graduate School

2007

# NANOCONTROLLER PROGRAM OPTIMIZATION USING ITE DAGS

Sarojini Priyadarshini Rajachidambaram
*University of Kentucky*, rspriya@uky.edu

Right click to open a feedback form in a new tab to let us know how this document benefits you.

## Recommended Citation

Rajachidambaram, Sarojini Priyadarshini, "NANOCONTROLLER PROGRAM OPTIMIZATION USING ITE DAGS" (2007). *University of Kentucky Master's Theses*. 479.
https://uknowledge.uky.edu/gradschool_theses/479

This Thesis is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Master's Theses by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

ABSTRACT OF THESIS

# NANOCONTROLLER PROGRAM OPTIMIZATION USING ITE DAGS

Kentucky Architecture nanocontrollers employ a bit-serial SIMD-parallel hardware design to execute MIMD control programs. A MIMD program is transformed into equivalent SIMD code by a process called Meta-State Conversion (MSC), which makes heavy use of enable masking to distinguish which code should be executed by each processing element. Both the bit-serial operations and the enable masking imposed on them are expressed in terms of if-then-else (ITE) operations implemented by a 1-of-2 multiplexor, greatly simplifying the hardware. However, it takes a lot of ITEs to implement even a small program fragment.

Traditionally, bit-serial SIMD machines had been programmed by expanding a fixed bit-serial pattern for each word-level operation. Instead, nanocontrollers can make use of the fact that ITEs are equivalent to the operations in Binary Decision Diagrams (BDDs), and can apply BDD analysis to optimize the ITEs. This thesis proposes and experimentally evaluates a number of techniques for minimizing the complexity of the BDDs, primarily by manipulating normalization ordering constraints. The best method found is a new approach in which a simple set of optimization transformations is followed by normalization using an ordering determined by a Genetic Algorithm (GA).

Keywords: Nanocontrollers, BitC compiler, BDD Optimization, ITE DAGs, Kentucky Architecture

Sarojini Priyadarshini Rajachidambaram

08/22/2007

NANOCONTROLLER PROGRAM OPTIMIZATION USING ITE DAGS

By

Sarojini Priyadarshini Rajachidambaram

Dr. Henry G. Dietz

(Director of Thesis)

Dr. Yuming Zhang

(Director of Graduate Studies)

08/22/2007

# RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master's degrees and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the thesis in whole or in part requires also the consent of the Dean of the Graduate School of the University of Kentucky.

A library which borrows this thesis for use by its patrons is expected to secure the signature of each user.

| Name | Date |
| --- | --- |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

THESIS

Sarojini Priyadarshini Rajachidambaram

The Graduate School

University of Kentucky

2007

# NANOCONTROLLER PROGRAM OPTIMIZATION USING ITE DAGS

---

THESIS

---

A thesis submitted in partial fulfillment of the requirements of the degree of Master of Science in the College of Engineering at the University of Kentucky

By

Sarojini Priyadarshini Rajachidambaram

Lexington, Kentucky

Director: Dr. Henry G. Dietz, Department of Electrical and Computer Engineering

Lexington, Kentucky

2007

Copyright © Sarojini Priyadarshini Rajachidambaram 2007

MASTER'S THESIS RELEASE

I authorize the University of Kentucky Libraries to reproduce this thesis in whole or in part for purposes of research.

Signed: Sarojini Priyadarshini Rajachidambaram

Date: 08/22/2007

ACKNOWLEDGMENT

My heartfelt thanks to my academic advisor Dr. Hank Dietz , for introducing me to the area of compiler optimization and for being the guiding light for this thesis. Thanks to Dr. Bill Dieter for his insights and guidance through out my Masters program at the University of Kentucky. Thanks to Dr. Robert Heath for agreeing to serve in my committee. I would also like to acknowledge my fellow KAOS research group members for their inputs during my thesis writing process. Above all I would like to thank my family for motivating me and fueling my ambitions through out my life.

TABLE OF CONTENTS

ACKNOWLEDGEMENT . . . . . . . . . . . . . . . . . . . . . . . . . . . . iii

TABLE OF CONTENTS . . . . . . . . . . . . . . . . . . . . . . . . . . . iv

LIST OF FIGURES . . . . . . . . . . . . . . . . . . . . . . . . . . . . vii

LIST OF TABLES . . . . . . . . . . . . . . . . . . . . . . . . . . . . ix

1 CHAPTER 1: INTRODUCTION . . . . . . . . . . . . . . . . . . . . . . 1

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Contribution of Thesis . . . . . . . . . . . . . . . . . . . . . . 3

1.3 Organization of Thesis . . . . . . . . . . . . . . . . . . . . . . 3

2 CHAPTER 2: THE NANOCONTROLLER ARCHITECTURE . . . . . . . . 5

2.1 Example Nanocontroller Applications . . . . . . . . . . . . . . . . 5

2.2 Nanocontroller Requirements . . . . . . . . . . . . . . . . . . . . 6

2.2.1 Low Circuit Complexity . . . . . . . . . . . . . . . . . . . . 6

2.2.2 Predictable Real-time Execution Characteristics . . . . . . . . . . 6

2.2.3 Localized Input/Output . . . . . . . . . . . . . . . . . . . . 7

2.2.4 Ability to act as a Parallel Computer . . . . . . . . . . . . . . . 7

2.2.5 Independently Programmability . . . . . . . . . . . . . . . . . 7

2.2.6 Re-programmability . . . . . . . . . . . . . . . . . . . . . . 8

2.3 Basic Terminology . . . . . . . . . . . . . . . . . . . . . . . . 8

2.3.1 Single Instruction Stream Multiple Data Stream (SIMD) . . . . . 9

2.3.2 Multiple Instruction Stream Multiple Data Stream (MIMD) . . . . 9

2.3.3 Meta-State Conversion . . . . . . . . . . . . . . . . . . . . . 10

2.3.4 Common Sub-expression Induction . . . . . . . . . . . . . . . . 11

2.4 KITE Architecture . . . 12
2.4.1 The Control Unit . . . 12
2.4.2 The Instruction Sequencers . . . 13
2.4.3 The Nanoprocessor . . . 14
3 CHAPTER 3: DATA STRUCTURES OUTPUT BY THE BITC COMPILER . . . 16
3.1 Boolean Function Representation . . . 16
3.1.1 Binary Decision Diagram . . . 16
3.1.2 Karplus's ITE DAGs . . . 18
3.1.3 Kentucky Architecture ITE DAGs . . . 19
3.2 BitC Programming Language . . . 21
3.3 BitC Compilation Overview . . . 22
3.3.1 Word-Level to ITE Conversion . . . 24
3.3.2 Normalization . . . 26
4 CHAPTER 4: THE VARIABLE ORDERING PROBLEM . . . 30
4.1 Impact Of Variable Ordering . . . 30
4.2 Combating BDD Complexity Explosion In Nanocontrollers . . . 32
4.3 Variable Ordering Heuristics: Existing Work . . . 32
4.3.1 Algorithms based on variable exchange . . . 32
4.3.2 Genetic Algorithms . . . 33
5 CHAPTER 5: PRELIMINARY TECHNIQUES . . . 35
5.1 Original Bit Order . . . 35
5.2 Reverse Bit Ordering Technique . . . 35
5.2.1 Results . . . 36
5.3 Print Form Transformation . . . 36
5.3.1 Results . . . 38

6 CHAPTER 6: GA FOR FINDING OPTIMAL VARIABLE ORDER . . . . . 41
6.1 GA Details . . . . . 41
6.1.1 Genome . . . . . 41
6.1.2 Fitness . . . . . 42
6.1.3 Selection . . . . . 42
6.1.4 Mutation . . . . . 42
6.1.5 Crossover . . . . . 43
6.1.6 Steady-State Vs. Generational Genetic Algorithm . . . . . 43
6.2 Algorithm Description . . . . . 43
6.3 Experimental Procedure . . . . . 45
6.4 Results . . . . . 46

7 CHAPTER 7: BDD VARIABLE ORDERING GA WITH PFT AFTER NORMALIZATION . . . . . 48
7.1 Description . . . . . 48
7.2 Experimental Procedure . . . . . 49
7.3 Results . . . . . 50

8 CHAPTER 8: BDD VARIABLE ORDERING GA WITH PFT BEFORE NORMALIZATION . . . . . 53
8.1 Description . . . . . 53
8.2 Experimental Procedure . . . . . 53
8.3 Results . . . . . 54

9 CHAPTER 9: CONCLUSIONS AND FUTURE WORK . . . . . 57

10 CHAPTER 10: REFERENCES . . . . . 59

VITA . . . . . 62

LIST OF FIGURES

2.1 Representation of SIMD processing . . . . . . . . . . . . . . . . . . . . . 9
2.2 Representation of MIMD processing . . . . . . . . . . . . . . . . . . . . . 10
2.3 Illustration of Meta-State Conversion . . . . . . . . . . . . . . . . . . . . . 11
2.4 The KITE Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.5 Equivalence between a ITE representation and the KITE hardware . . . . 14

3.1 A general BDD representation . . . . . . . . . . . . . . . . . . . . . . . . 17
3.2 A BDD representing the Boolean function $a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$ . . . . 17
3.3 A general ITE DAG representation . . . . . . . . . . . . . . . . . . . . . 18
3.4 An ITE DAG representing the Boolean function $a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$ . 19
3.5 Representation of the BDD generated at the BitC compiler output . . . . 21
3.6 Block Diagram illustrating BitC compilation . . . . . . . . . . . . . . . . 23
3.7 Gate level representation of a full-adder circuit . . . . . . . . . . . . . . . 24
3.8 If-Then-Else DAG representing the full-adder program . . . . . . . . . . 25

4.1 Figure illustrating the difference in DAG complexity with difference in variable order while computing `int:3 a, b, c;c=a+b+c;` . . . . . 31

5.1 Figure illustrating the default bit order for variables . . . . . . . . . . . 35
5.2 Figure illustrating the reversed bit order for variables . . . . . . . . . . . 36
5.3 Scatter plot comparing the number of live ITEs in the final BDD using original bit order vs. reversed bit order . . . . . . . . . . . . . . . . . . . 37
5.4 Scatter plot comparing the number of nodes in the intermediate ITE DAG when using PFT as the optimization technique vs. the number of nodes in the intermediate BDD using normalization as the optimizing technique . . 39

5.5 Scatter plot comparing the number of nodes in the final ITE DAG when using PFT as the optimizing technique vs. number of nodes in the final BDD when using normalization as the optimizing technique . . . . . . . . 40

6.1 GA data structure diagram . . . . . . . . . . . . . . . . . . . . . . . . 45

6.2 Plot comparing the number of nodes in the final BDD normalized using GA variable ordering vs. default variable ordering . . . . . . . . . . . . . 47

6.3 Plot comparing the number of nodes in the final BDD normalized using GA variable ordering vs. reversed variable ordering . . . . . . . . . . . . 47

7.1 The data structures and data flow used in the algorithm for GA fitness calculation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

7.2 Scatter Plot final ITE DAG size when using GA for variable ordering with PFT after normalization vs. size of the final BDD when using our basic GA for variable ordering . . . . . . . . . . . . . . . . . . . . . . . . 51

8.1 The data structures and data flow used in the algorithm for GA fitness calculation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

8.2 Scatter plot comparing the size of final BDD using GA for variable ordering with PFT before normalization vs. our basic GA for variable ordering 55

LIST OF TABLES

3.1 Logic Operations and their ITE equivalents expressed using C's trinary operator syntax . . . . . . . . . . 24

5.1 The Print Form Transformation . . . . . . . . . . 38

# CHAPTER 1: INTRODUCTION

The new compiler optimization technology developed in this thesis is not as closely related to other compiler optimizations as it is to methods of logic design and analysis. To understand why this is so, it is useful to review the strange properties of nanocontrollers.

## 1.1 Motivation

Single-chip sensor arrays, fabricated by micro- or nano-scale processes, are now being used in a variety of fields. Various types of single-chip array output devices also are being constructed. All of these devices can function better when accompanied by their own intelligent, potentially feedback-based, controller. One could attempt to control such nanodevices using a conventional microcontroller [KRD07], but sending signals to/from a single controller becomes infeasible as the number of devices needing control becomes large.

It is difficult to make intelligent control small enough to fit on-chip with the tiny devices they control. This reduction in controller circuit complexity can be achieved by using SIMD "nanoprocessing" elements in the controller. One such solution is to use a "Kentucky Architecture" nanocontroller programmed by an If-Then-Else (ITE) compilation model [DAG04]. The KITE (Kentucky ITE) architecture is a bit-serial multiplexor-based SIMD architecture with as few as tens of one-bit local registers. The nanocontroller would have no on-chip storage except these registers in order to minimize circuit complexity. So the one-bit local registers must fulfill all the on-chip storage requirements by storing live ITEs, temporaries, network and I/O information, etc. An external PROM acts as the program memory, cost-effectively supporting very large control programs. This less complex hardware comes at the expense of increased expectations on the compiler with respect to

issues like register allocation and the complexity of input program that can be handled by the nanocontroller.

The compiler for a nanocontroller converts an input control program in a high-level language (Section 3.2), into form that can be understood by the KITE hardware. Each operation in the control program is transformed from a word-level representation into bit-level Boolean operation sequences represented by Directed Acyclic Graphs (DAGs) of If-Then-Else (ITE) operators. Each basic block in the input program typically generates an ITE DAG containing hundreds or even thousands of ITEs. Because each ITE requires a constant amount of time to execute, run time complexity is proportional to number of nodes in the ITE DAG, and minimizing this number is the key to achieving good performance. The number of nodes in the ITE DAGs also impacts compilation speed and difficulty of register allocation; manipulating ITE DAG complexity also can improve these properties. The complexity of the ITE DAGs can be managed using techniques borrowed from the logic circuit design and verification community, such as Binary Decision Diagrams (BDD) [Bry92].

Variable ordering has been found to influence the complexity of normalized DAG-based data structures like the BDDs. BDD node counts have been found to vary from linear to exponential in the number of input variables depending on the variable order used for their generation because the order of Boolean variable manipulation influences the amount of factorization of a Boolean function. In a nanocontroller, an exponential size ITE DAG representation of the bit-serial control program increases the response time of the nanocontroller for the target application, requires a large off-chip memory for storage, and complicates the register allocation portion of the compiler. For some Boolean functions, the number of ITEs that must be analyzed exceeds compiler limitations, so keeping the number of intermediate ITEs small can be just as important as minimizing the final number of ITEs. Thus, techniques for BDD variable ordering have the potential to significantly improve processing of ITE DAGs for nanocontrollers.

However, finding the optimal variable order to minimize the complexity of a BDD is an NP-complete problem[BW96] – minimizing the ITE DAGs described in this thesis also is NP-complete. Several heuristics have been devised [DG97] [LB05] [HS01] [Ric93] to find the optimal variable order for BDDs, and this thesis was able to build upon these ideas for minimization of nanocontroller ITE DAGs.

### 1.2 Contribution of Thesis

This thesis introduces techniques for minimizing the size of the target DAG output by BitC compiler, using a combination of heuristic search techniques and standard Boolean normalization techniques and transformations. Techniques used in the circuit minimization realm have been applied for program minimization. Three genetic algorithms have been devised to find the optimal variable order, which minimizes the BDD-equivalent ITE DAGs output by the BitC compiler.

### 1.3 Organization of Thesis

Chapter 2 sets the stage for this thesis, by introducing some nanocontroller applications, requirements and the KITE architecture designed to meet the requirements of nanoscale devices. Chapter 3 introduces the data structures used in KITE Architecture and differentiates it from BDDs and other ITE DAG data structures; this thesis sometimes uses the shorthand of referring to a nanocontroller ITE DAG with BDD properties as simply a BDD. Chapter 4 provides the background on the compiler which outputs nanocontroller programs. Chapter 5 introduces the BDD variable ordering problem and how it is related to the compilation for nanocontroller. The next chapter carries details on some preliminary techniques that motivated us towards designing some of our GA-based heuristics. Chapter 7 discusses the initial GA-based variable order optimization implemented in the BitC compiler. Chapter 8 and 9 discuss some advanced GA-based searches for optimal variable

orders. Chapter 10 presents the conclusions and future work for this thesis.

# CHAPTER 2: THE NANOCONTROLLER ARCHITECTURE

This chapter discusses some nanocontroller applications, the hardware and software requirements of the nanocontrollers, and the KITE architecture which meets the discussed requirements.

## 2.1 Example Nanocontroller Applications

A few applications with which a nanocontroller can be used are:

- Loss of quality in the image rendered by an image sensor in a digital camera is due to the application of the same gain and integration time settings to all pixels in the sensor. By using a nanocontroller to control each pixel, the pixels can be corrected for defects independently thereby increasing the dynamic range and lowering the noise in the image output.

- Nanosensors used in chemical and biological applications are made up of arrays of carbon nanotubes whose electrical and quantum mechanical properties are used to measure the level of chemical and biological toxins. Properties are measured by watching how resistance of a clump of nanotubes changes over time as a chemical passes through the walls of the nanotubes. Measuring thousands or even millions of these resistances by routing the weak analog signals off-chip for processing simply is not feasible. Use of a nanocontroller under each sensor array allows the correction of sensor defects in software in addition to allowing direct digital readout of sensor data.

- Digital Light Processing (DLP) based projectors use an array of tiny pivoted micro-mirrors arranged on a semiconductor chip for image rendering. The tilt of each

micro mirror determines if that pixel is dark or bright; a pixel is bright if its mirror deflects light into the projection lens. Gray scale is obtained by linear Pulse Width Modulation (PWM). However, human eyes respond logarithmically to light, so the fact that mirrors can only tilt about 1,000 times per second yields very few visible gradations between black and white. A nanocontroller at each micro mirror could provide much finer timing control of the PWM to better approximate exponential brightness, while simultaneously simplifying the off-chip control logic.

## 2.2 Nanocontroller Requirements

A nanocontroller should meet the requirements outlined in the ensuing subsections in order to be suitable for use in the applications discussed in the previous section.

### 2.2.1 Low Circuit Complexity

The nanocontrollers should have a circuit complexity low enough to be paired with nanoscale devices. Using low temperature nanotechnology fabrication methods, the nanocontroller array might be fabricated first followed by the nanodevice array on top of their nanocontrollers. For a circuit complexity comparable to the physical size of the nanofabricated devices, the controller should consist of no more than hundreds of transistors.

### 2.2.2 Predictable Real-time Execution Characteristics

In order to be able to control a sensing device in real-time, the nanocontroller should have predictable execution timing characteristics. The amount of timing precision required varies depending on the application of the device which the nanocontroller controls. With the small physical dimensions of the nanocontroller contributing small time constants, a real-time response time no worse than a microsecond is desired.

### 2.2.3 Localized Input/Output

Since each nanocontroller is envisioned to be fabricated on the same substrate as the controlled devices, it is necessary to accommodate the logic for analog and digital I/O operations in the nanocontroller. While digital I/O can be accomplished through the use of registers as I/O devices, analog I/O would require a method for Analog-to-Digital and Digital-to-Analog Conversion (ADC and DAC). To prevent the analog I/O circuitry from affecting the low circuit complexity of the nanocontroller, analog input would be implemented in software by recording the amount of time taken for a digital threshold voltage to be crossed when charging a capacitor and analog output would be implemented using Pulse-Width Modulation software to drive a RC-filter circuit. Implementing analog I/O using software, allows precision to be traded off for sample speed unlike conventional ADC/DAC units which fix the precision of conversion. This type of analog I/O is possible only in a fast enough processor that also has a predictable execution time.

### 2.2.4 Ability to act as a Parallel Computer

A typical chemical sensor array has thousands or millions of nanosensing devices on a single chip that need to act together as a system to sense and report the levels of particular compounds in its environment. This activity requires coordination in the actions of all the nanosensors, to reduce all their inputs to a single-meaning that would be reported by the chemical sensor. So all the nanocontrollers on a chip must have the ability to act together as a parallel computer system.

### 2.2.5 Independently Programmability

Nanotechnology-based devices exhibit a lot of process variations compared to relatively larger-scale devices, mainly contributed by variations in their molecular-level structure.

These variations among the sensors lead to varying levels of device non-linearities in adjacent nanosensors, hence requiring different constants/ algorithms for normalizing their output values. If each nanocontroller is independently programmable, then the control and sensing algorithms can take different constants or code paths depending on the state of the sensor with which it interacts. Independent programmability of the nanocontrollers, also lets a nanocontroller minimize or prevent the impact of faulty devices in the sensor array on the measured output by allowing it to take a different code path. Providing independent programmability to support such algorithms requires a programming environment that can support MIMD execution (see Section 2.3.2).

### 2.2.6 Re-programmability

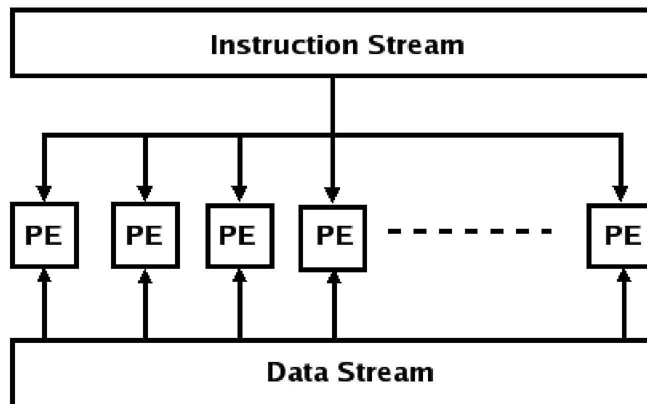
After months or years of use, nanosensors can develop faulty elements which will require a change or upgrade in their control program. This means that the nanocontroller programs must be able to be changed by reprogramming the memory. With the reprogramming occurring infrequently, it is acceptable to perform expensive compile-time transformations which can improve the efficiency of program execution in the nanoprocessor and hence the real-time response of the nanosensor.

## 2.3 Basic Terminology

The requirements stated in Section 2.2 cannot be realized using any existing micro-controller architectures and compilers. A new architecture and a compilation model is necessary. The nanocontroller architecture is introduced in this chapter and for a better understanding of the architectural concepts the following terms are introduced.

### 2.3.1 Single Instruction Stream Multiple Data Stream (SIMD)

SIMD is a data parallel processing model where many processing elements perform the same operation on different streams of data. SIMD processors have a single control unit which controls many processing elements. Instruction storage and decode is controlled by the control unit and so the individual processing elements do not require an instruction memory. The SIMD processing model has a simple hardware because the decoding, addressing and sequencing operations are all taken care of by a central control unit and hence no additional complexity is added to the PE's. SIMD processing is preferred in nanocontrollers due to its simple hardware.




(a)

Figure 2.1: Representation of SIMD processing

### 2.3.2 Multiple Instruction Stream Multiple Data Stream (MIMD)

MIMD is a control parallel processing model where many processing elements perform different operations on different streams of data. Each processing element takes different execution paths depending on the Instruction stream processed by it. The control and sensing algorithms for the nanocontrollers may require different constants or code path depending on the state of the sensor interacting with it. So MIMD has the independent

programmability feature required for the nanocontrollers. This processing model requires a hardware which is complex compared to SIMD due to the overhead of replicating the logic for instruction decode, addressing and sequencing for each processing element.

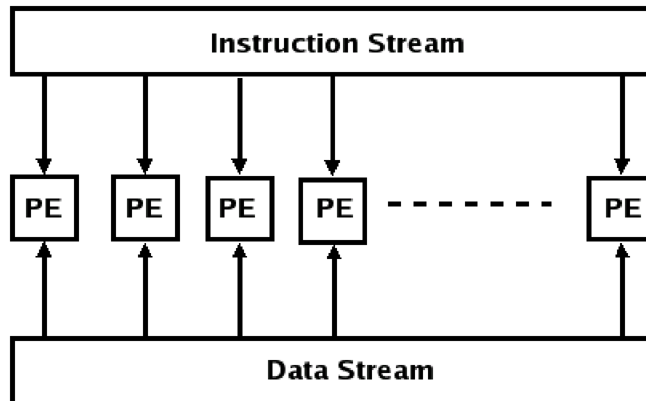



Figure 2.2: Representation of MIMD processing

### 2.3.3 Meta-State Conversion[1]

In the nanocontrollers, the simplicity of SIMD processors and the independent programmability of MIMD processors are desired at the same time. While there are several techniques available for MIMD emulation on SIMD hardware [NH90] [DC93] [San94], they all require that each PE must have a copy of the MIMD program in their local memory – which adds a lot of hardware to a traditional SIMD processor.

Meta-State Conversion [DK93] is a compiler technique which removes the overhead of storing the control program in the local memory of each processing element and hence favours our model of computation. This technique considers the set of processor states at a particular time as a single meta-state. Using static scheduling techniques [HA91], MSC converts a MIMD program into a SIMD-executable finite automaton based on meta-states as shown in Figure 2.3. The next meta-state to transition to is decided based on the global OR of votes from all participant processors. The generated meta-state automaton is held

[1]Although Meta-State Conversion is a compiler technique, it is discussed in this chapter to enable better understanding of the KITE control unit.

by the SIMD control unit, thereby removing the necessity of a separate instruction memory for each processing element.

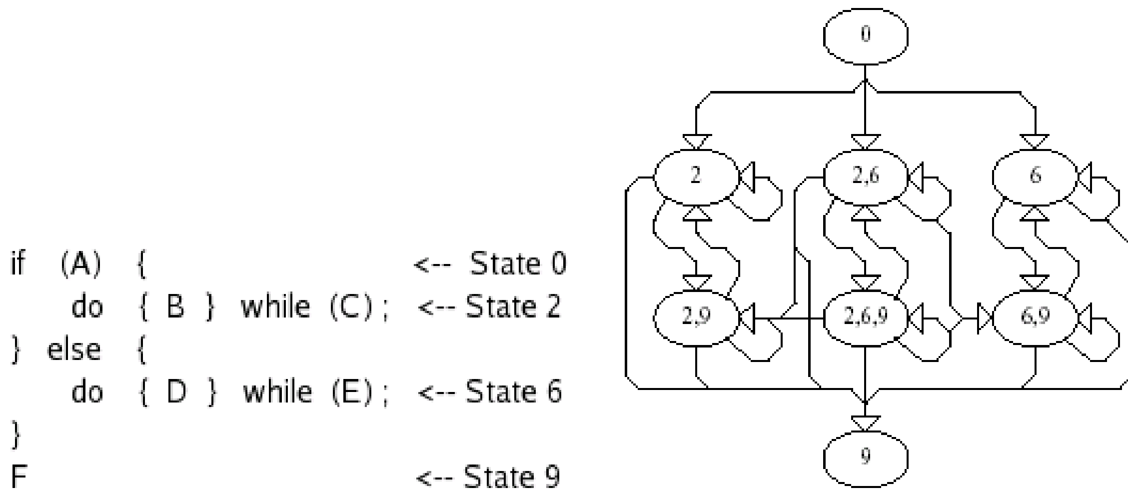



Figure 2.3: Illustration of Meta-State Conversion

### 2.3.4 Common Sub-expression Induction[2]

After MSC, many meta-states in the meta-state graph contain more than one MIMD state. These MIMD states contain many instructions that in true MIMD execution would have been executed in parallel. Execution of these instructions on SIMD hardware serializes the instructions, with each PE enabled only for the MIMD code it would have executed. (Nanocontrollers simulate disable of PEs by masking operations.) Efficiency of execution can be improved by factoring the operations that are common between different MIMD states, which allows them to be executed in parallel by the SIMD PE's. Common Sub-expression Induction (CSI) [Die92] is the technique which develops a code schedule for the SIMD PE's by factoring common operations between different threads in a meta-state.

[2]Although Common Sub-expression Induction is a compiler technique, it is discussed in this chapter for continuity.

## 2.4 KITE Architecture

KITE (Kentucky If Then Else) is a single-bit architecture designed to cater the needs of intelligently controlled nanosensing and MEMS devices. ITE (If-Then-Else) is the only instruction in the KITE Instruction Set Architecture. A control program written in a high level language is converted into a meta-state automaton by the BitC compiler. The code schedule developed by Common Sub-expression Induction after Meta-State Conversion is used to execute the code in the meta-state automaton in the SIMD nanoprocessing elements. Each meta-state in the automaton is composed of ITE DAGs representing the MIMD states composing that state and ends in k-way branches to k possible meta-states. The next meta-state to transition to, is determined using a Global OR (GOR) of votes from all participant processors. The Kentucky Architecture differs from traditional SIMD in that it implements control by selection and thus the control unit is not a uniprocessor but provides VLIW style multi-way branch support. The envisioned KITE architecture has three component modules:

### 2.4.1 The Control Unit

The meta-state automaton is stored in an off-chip memory like a PROM or EEPROM, interfaced by address (A) and data (D) buses. A computer host loads the meta-state program into the off-chip memory. Unlike a typical SIMD control unit which fetches one instruction at a time, the KITE control unit prefetches a compressed representation of the next meta-state based on a Global OR of votes from all participant processors. Thus a KITE control unit controls the program memory interface and not the processors. After prefetch, the controller would perform decompression, branch prediction and instruction cache management treating each basic block as a single unit. Thus the control unit can prefetch basic blocks at a slower clock (C0) rate while broadcasting the partially decoded instructions (SITEs) at a faster rate (C1).

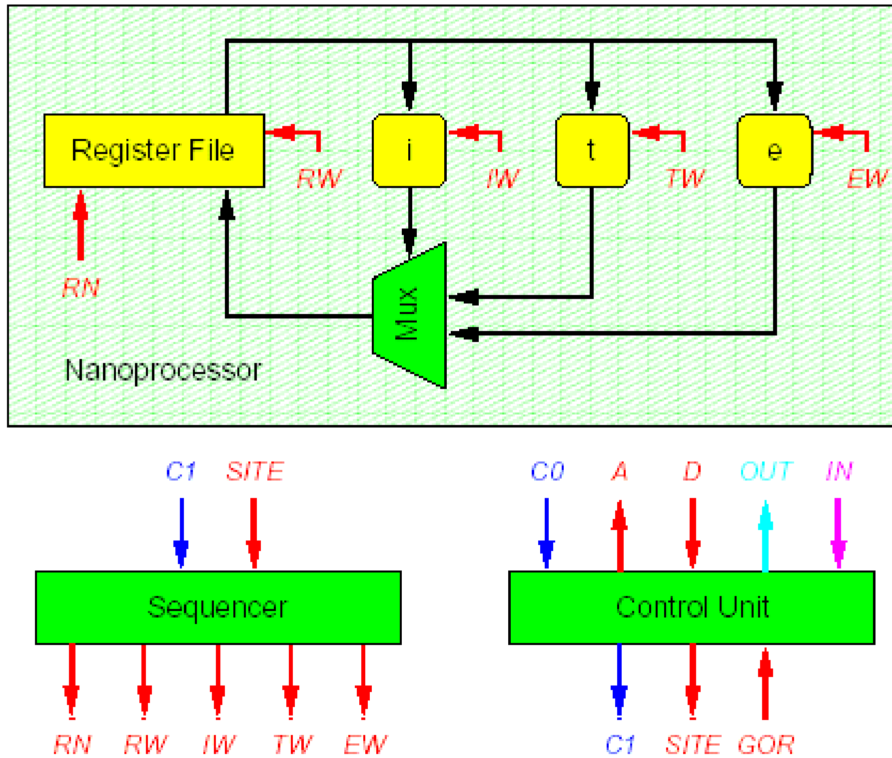



Figure 2.4: The KITE Architecture

### 2.4.2 The Instruction Sequencers

Broadcast of decoded signals in a huge fan-out device is a clock-speed limiting factor. The purpose of a sequencer is to prevent the instruction broadcast delay from affecting the nanoprocessor execution speed. The SITE representation of an instruction generates four consecutive clock cycles worth of control information for the nanoprocessors. Thus the control line outputs of the sequencer (RN, RW, IW, TW, and EW) effectively clock each nanoprocessor. So the input clock to a sequencer can be as much as four times slower than the nanoprocessor clock, thereby effectively hiding the instruction broadcast delay from higher level hardware.

### 2.4.3 The Nanoprocessor

A nanoprocessor is a simple piece of hardware which consists of a few single-bit registers, a register-number decoder, and a 2-to-1 multiplexor. Since the operation of a 2-to-1 multiplexor is analogous to the software If-Then-Else instruction, it can process the ITE instructions and hence acts as the ALU for the nanoprocessor. The three inputs to the multiplexor (if, then and else), are stored in three single bit registers (i, t, and e) . If the multiplexor's select input (***if*** part of the instruction) is true then the ***then*** bit is latched, otherwise the ***else*** bit is latched to the output. This value is stored in another register number. The equivalence between the if-then-else operational model and KITE hardware is shown in Figure 2.5.

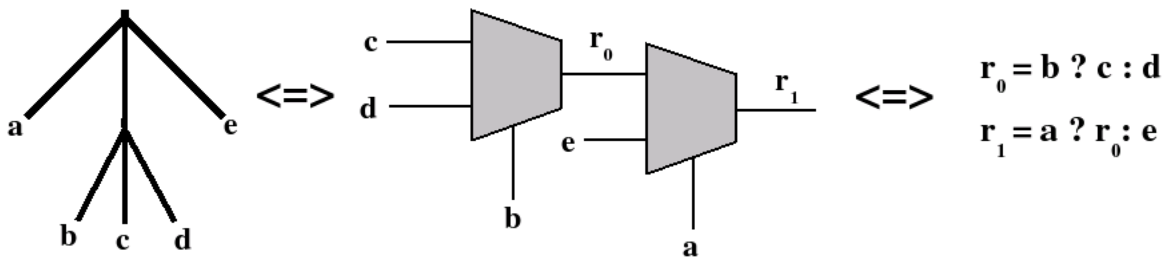



Figure 2.5: Equivalence between a ITE representation and the KITE hardware

The registers of a nanoprocessor hold program variables, constants, network connections, and I/O device interfaces. The registers 0 and 1 are hard-coded to represent constants 0 and 1. The SITE representation of an instruction is four register numbers (S, I, T, and E) consisting of the register **S** to store into, a register **I** for ***If*** bit, a register **T** for ***then*** bit, and a register **E** for ***else*** bit. A sequencer converts this information into a four-cycle sequence. For example, a SITE (5, 6, 7, 8) would cause the sequence:

1. RN=6, RW=0, IW=1, TW=0, EW=0
2. RN=7, RW=0, IW=0, TW=1, EW=0
3. RN=8, RW=0, IW=0, TW=0, EW=1

4. RN=5, RW=1, IW=0, TW=0, EW=0

to be broadcast from the sequencer.

# CHAPTER 3: DATA STRUCTURES OUTPUT BY THE BITC COMPILER

Sections 2.3.3 and 2.3.4 already introduced some enabling compiler technologies for MIMD programmability on SIMD hardware. For converting a controller program written in a high-level language to a form executable on the bit-serial nanocontroller hardware, a number of techniques from multi-level logic minimization are used. The data structures used for this purpose are introduced in this section.

## 3.1 Boolean Function Representation

Boolean functions can be represented using Truth Tables, Karnaugh maps, canonical Sum-Of-Products form, Binary Decision Diagrams (BDD), If-Then-Else DAGs etc. In the case of Truth Tables, Karnaugh Maps and SOP forms, any $n$argument function requires a representation of complexity $2^n$. Using any of these representations to manipulate a Boolean function involving many variables would either cause the application to run out of memory or take exponential amount of time in the worst case. Binary Decision Diagrams and If-Then-Else DAGs have a canonical representation which prevents them from having an exponential complexity for commonly encountered functions. In the following sections describe BDDs, ITE DAGs and a variant of these structures used in the BitC compiler.

### 3.1.1 Binary Decision Diagram

**DEFINITION:** *A Binary Decision Diagram is a rooted binary directed acyclic graph in which the terminal nodes are two leaves* TRUE *and* FALSE *and each non-terminal node is labeled with a Boolean variable, with its two out-edges leading to a high child and a low child which could be a nonterminal node or a leaf.*

A Binary Decision Diagram (BDD) is a data structure useful for representing and manipu-

lating a Boolean function. BDDs find extensive application in the fields of symbolic model checking, test generation, logic synthesis, fault simulation etc. In a BDD, the **if-** part is always a simple variable, whereas the **then-** and the **else-** parts can be recursively defined in terms of other variables. In general, a BDD is represented as shown in Figure 3.1, where v is a trivial variable and x is a non-trivial node.

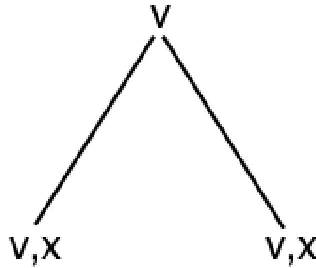



Figure 3.1: A general BDD representation

The normal form for BDD, does not allow negated nodes in the synthesized representation. Negation in the nodes is represented using an ITE of the form `if` (a) `then` 0 `else` 1. A BDD representing the Boolean function $a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$ is shown in Figure 3.2.

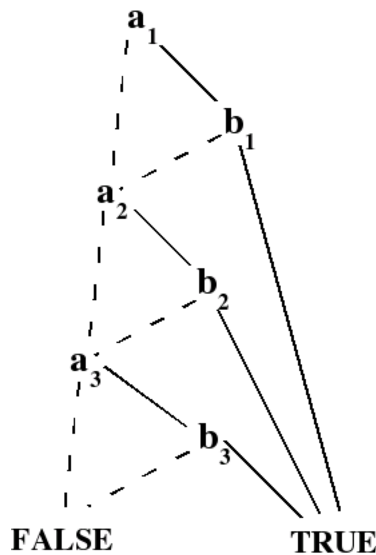



Figure 3.2: A BDD representing the Boolean function $a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$

### 3.1.2 Karplus's ITE DAGs

**DEFINITION:** *An If-Then-Else (ITE) DAG as defined by Karplus is a ternary directed acyclic graph with its leaves labeled with* FALSE *or a literal, and each internal node has three edges pointing to the if-, then- and else- parts. Any of the if-, then-, and else- parts may be specified as a complemented reference, incorporating negation without introducing another node; for example,* TRUE is the reference ¬FALSE.

If-Then-Else(ITE) DAG is a data structure closely related to the BDD representation of a Boolean function, such that a BDD with a two-cut can be trivially transformed into an If-Then-Else triple representing the same Boolean expression. The ***if-*** part corresponds to the part of the BDD above the cut and the ***then-*** and the ***else-*** parts correspond to the portions below the cut[Kar88b]. The If-Then-Else DAGs allow the ***if-*** part to be recursively defined, which allows the ITE representation shown in Figure 3.3 to have increased sub-expression sharing when compared to BDDs.

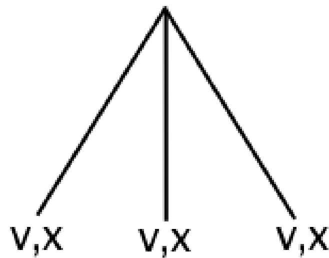



Figure 3.3: A general ITE DAG representation

Thus, the If-Then-Else representation increases opportunities for optimization. Rules for normalizing If-Then-Else DAGs allow negated nodes in the representation, which removes the need for nodes of the form if(a) then 0 else 1. In this way, negated nodes further reduce the complexity of If-Then-Else DAGs.

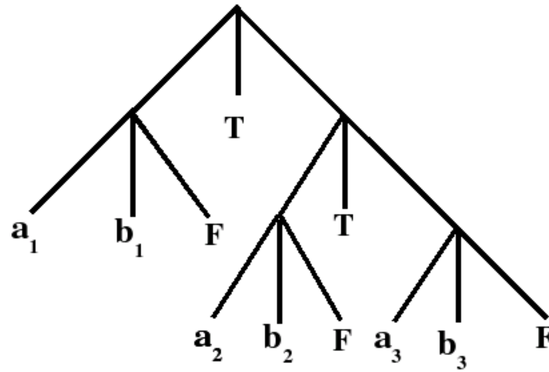



Figure 3.4: An ITE DAG representing the Boolean function $a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$

### 3.1.3 Kentucky Architecture ITE DAGs

**DEFINITION:** *A Kentucky Architecture If-Then-Else (*ITE*) DAG is a ternary directed acyclic graph with its leaves labeled with* TRUE, FALSE *or a literal, and each internal node has three edges pointing to the if-, then- and else- parts. The if- part can be a variable reference or a non-terminal node depending on the type of optimization algorithm used. Unlike Karplus ITE DAGs, complemented references are not allowed, thus requiring use of ITEs with the then-part* FALSE *and the else-part* TRUE.

The Kentucky Architecture ITE structure is a direct result of the use of SITE (Store If Then Else) as the only instruction implemented by nanocontroller hardware; for example, there are not complemented references because the hardware does not have any such capability. However, SITEs differ from ITEs in even more significant ways. SITEs are executed in a strictly sequential complete ordering, not the partial order given by a DAG. Further, the place to hold the result of each ITE is explicitly given in each SITE; that is the store- part of each operation. Many different ITE results typically are stored into the same register at different times in the sequence of SITE executions, whereas ITEs and BDDs are considered as each placing their result on a unique wire. Sophisticated register allocation algorithms are used to mark SITEs with their register assignments. The key insight was that by temporarily omitting the store- part and using a DAG to express the family of allowable SITE

orderings, SITE code can be modeled using an ITE form that can be optimized using techniques derived from those normally operating on BDDs or Karplus-style ITE DAGs. In fact, the store- parts are actually kept as a list of separate operations at the end of each Kentucky Architecture ITE DAG, and then later applied when generating the final SITE program sequence.

The Kentucky Architecture ITEs generated by the BitC compiler always have the general structure shown in Figure 3.3. However, the representation can be more closely related to either a Karplus If-Then-Else DAG or a BDD, depending on the point at which one examines the internal form and which optimization algorithms have been applied in the BitC compiler. When the BDD normalization algorithm is used, then the ITE DAG is equivalent to a BDD, and this thesis will often use the shorthand of referring to such an ITE DAG as a BDD. Before or without such normalization, the ITE DAGs used are essentially a subset of Karplus's If-Then-Else DAG structures in which no references are complemented. The thesis will generally refer to a Kentucky Architecture ITE DAG that might or might not be equivalent to an BDD as simply an ITE DAG. Earlier, the practicality of directly using Karplus's ITE DAG structure and normal form was evaluated, and the BitC compiler still offers that analysis as a command-line option, but the failure of that model to associate cost with negation made it a less effective way to optimize SITEs.

Compared to BDDs or Karplus's ITE DAGs, the diagrams of the data structures output by the BitC compiler are inverted. The inverted DAG representation, as shown in Figure 3.5, places registers at the top of the graphic. More precisely, the DAG diagrams used correspond to a level-order schedule of the ITE operations. A level-order schedule is commonly used to show the maximum parallelism available in performing a DAG; all ITE nodes on the same horizontal level could be executed simultaneously with sufficient hardware. The level of each node is one below the lowest level of a node upon whose result this node depends. Although the nanocontroller SITE programs do not use this type of parallelism in execution, the level order schedule is used to construct the meta-state automaton which

enables the MIMD program execution on SIMD hardware.

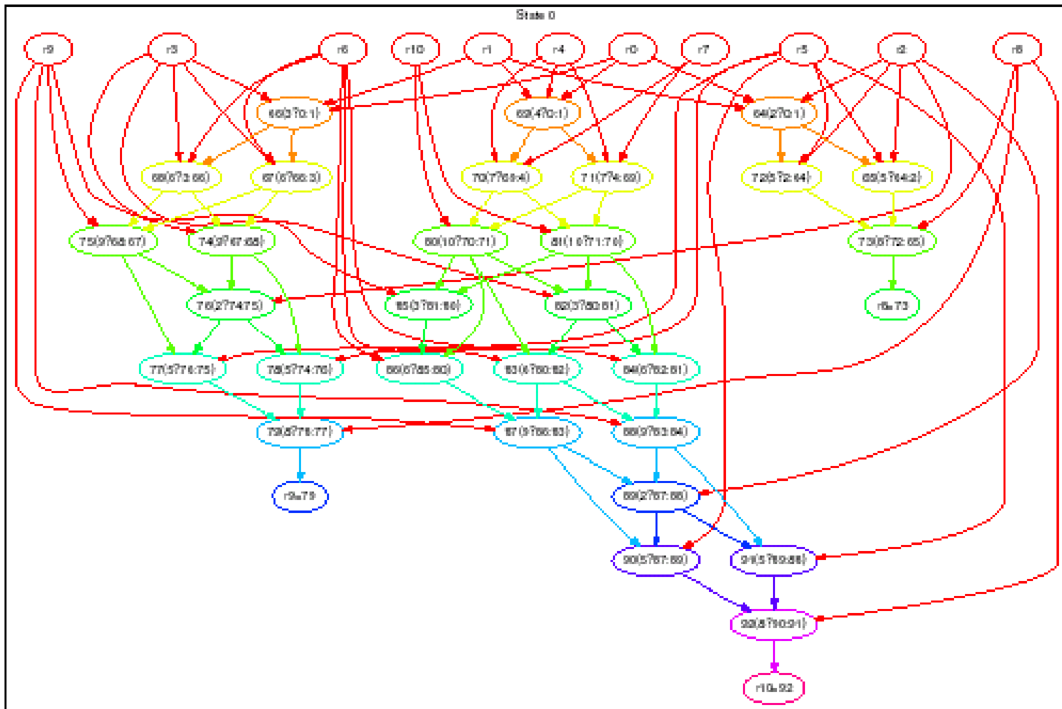


Figure 3.5: Representation of the BDD generated at the BitC compiler output

The BitC Compiler

This chapter discusses the features of the compiler which outputs the control program for the nanocontroller and the normalizing rules employed in the optimizing phases of the BitC compiler.

## 3.2 BitC Programming Language

BitC is the programming language for nanocontrollers. BitC extends the bit field syntax from the C language to allow explicit declaration of bit precision for variables. For example the declaration `int:2 a,b;`, declares variables `a` and `b` to be 2 bits each. As already stated, the registers 0 and 1 are reserved for constants 0 and 1, so the 4 bits in the variables get assigned to registers 2, 3, 4 and 5 respectively in the order of declaration. Standard C operators, conditional, and looping constructs are supported in BitC; function

calls are not supported because C-style recursion is not useful given the limited data memory of a the nanoprocessor. I/O operations are handled through application-specific registers, which are reserved prior to declaration of ordinary variables. For example, `int:1 adc@5;` reserves a single bit register starting at register 5. BitC is primarily a sequential programming language, but also supports parallel programming features like barrier synchronization, real-time operations, and inter-processor communication.

### 3.3 BitC Compilation Overview

During compilation, an input program written in BitC undergoes the phases of lexical analysis and parsing, during which the syntax and semantics of the input program are checked for correctness. After this, the input variables and program constants are loaded into the appropriate data structures in a format suitable for further manipulation. The BitC compiler then transforms the word-level operations, contained in an input expression written in BitC, into bit-level Boolean operations represented using If-Then-Else operators. These bit-level operations are then made to undergo ITE normalizing conditions and/or transformations specified in [Kar88b]. The normalized ITE expressions from all programs are then logically merged into a SPMD (Single Program Multiple Data) program, which then undergoes Meta-State Conversion to produce a guarded SIMD code. To improve the efficiency of execution of the guarded SIMD code on SIMD hardware, the common sub-expressions are factored out between the different MIMD states constituting a meta-state. During this phase the compiler performs CSI, in which it reuses the normalizations and/or transformations employed in the optimizing phase. The ITEs resulting after CSI are converted into SITEs which specify the registers in which the result of every ITE operation should be stored. To prevent maxlive from exceeding the number of available on-chip registers, a novel Register Allocation [ADR05] technique is applied before final code generation.

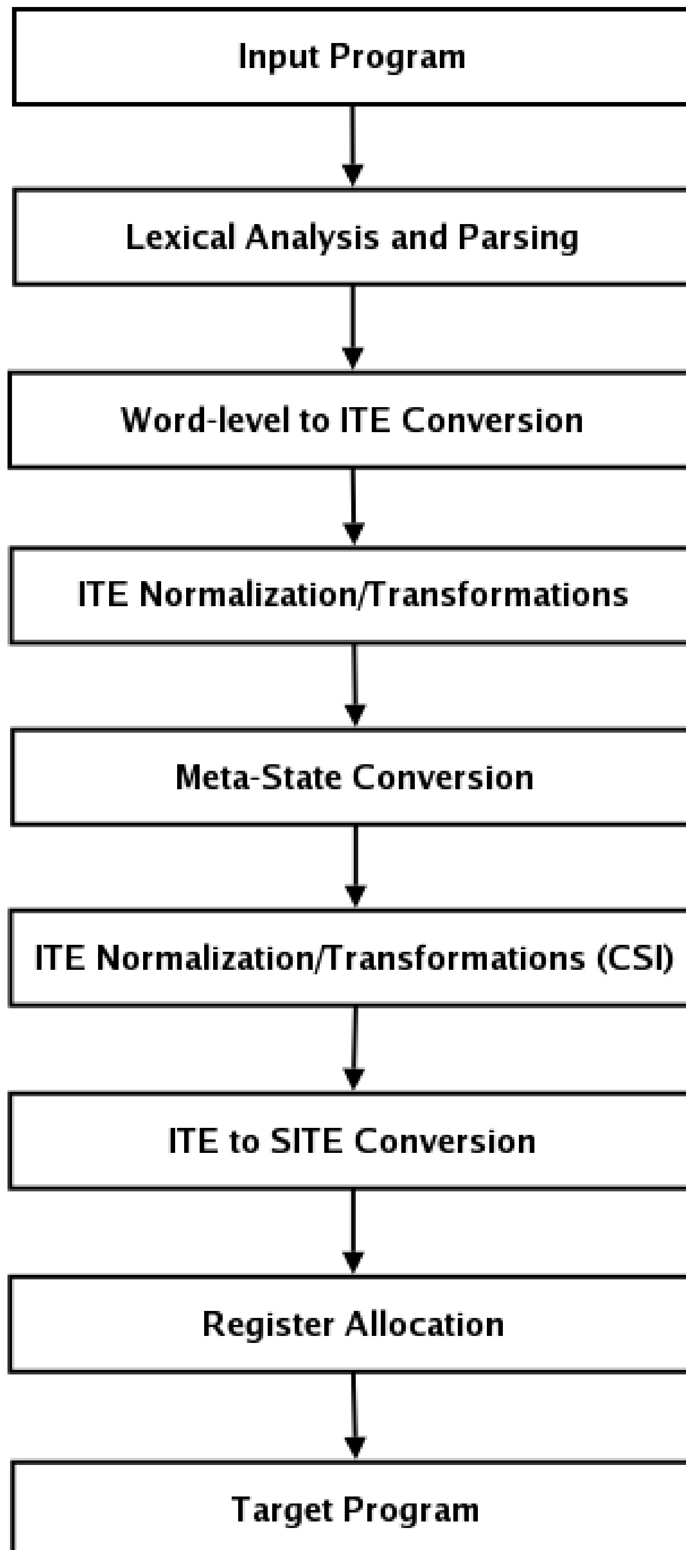



Figure 3.6: Block Diagram illustrating BitC compilation

### 3.3.1 Word-Level to ITE Conversion

The BitC compiler converts a word-level operation into its equivalent bit-level operation as shown in the following example. Consider the word-level program, `int:2 a, b, c; c=a+b;`. The add operation in this program can be expressed using the individual bits in each word value as:

$$\{c_1, c_0\} \;\leftarrow\; \{a_1, a_0\} + \;\{b_1, b_0\}$$

Implementing the above program using logic gates would lead to a circuit of the form shown in Figure 3.7.

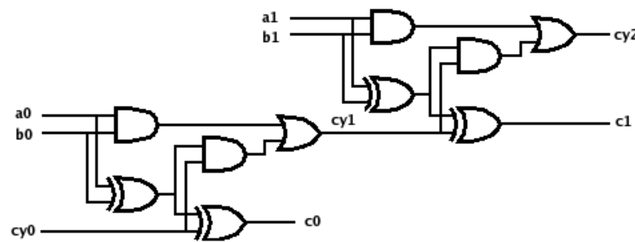



Figure 3.7: Gate level representation of a full-adder circuit

Common logic operations can be represented using the If-Then-Else operator as shown in Table 3.1.

Table 3.1: Logic Operations and their ITE equivalents expressed using C's trinary operator syntax

| Logic Operation | Equivalent ITE structure |
|---|---|
| (x AND y) | (x ? y : 0) |
| (x OR y) | (x ? 1 : y) |
| (NOT x) | (x ? 0 : 1) |
| (x XOR y) | (x ? (y ? 0 : 1) : y) |
| ((NOT x) ? y : z) | (x ? z : y) |

The logic operations involved in a two bit full-adder shown in Figure 3.7 can be expressed using the If-Then-Else operator shown in Table 3.1 as shown in the Algorithm 1.

| | | |
|---|---|---|
| $cy_0$ | $= 0$ | |
| $x_0$ | $=$ EXOR( $a_0$ , $b_0$ ) | $= ( a_0 \ ? \ ( b_0 \ ? \ 0 : 1 ) : b_0 )$ |
| $n_0$ | $=$ AND( $a_0$ , $b_0$ ) | $= ( a_0 \ ? \ b_0 : 0 )$ |
| $c_0$ | $=$ EXOR( $x_0$ , $cy_0$ ) | $= ( x_0 \ ? \ ( cy_0 \ ? \ 0 : 1) : cy_0)$ |
| $cy_1$ | $=$ OR( $n_0$ , AND( $x_0$ , $cy_0$ )) | $= ( n_0 \ ? \ 1 : ( x_0 \ ? \ cy_0 : 0 ))$ |
| $x_1$ | $=$ EXOR( $a_1$ , $b_1$ ) | $= ( a_1 \ ? \ ( b_1 \ ? \ 0 : 1 ) : b_1 )$ |
| $n_1$ | $=$ AND( $a_1$ , $b_1$ ) | $= ( a_1 \ ? \ b_1 : 0 )$ |
| $c_1$ | $=$ EXOR( $x_1$ , $cy_1$ ) | $= ( x_1 \ ? \ (cy_1 \ ? \ 0 : 1 ) : cy_1 )$ |
| $cy_2$ | $=$ OR( $n_1$ , AND ($x_1$ , $cy_1$ )) | $= ( n_1 \ ? \ 1 : ( x_1 \ ? \ cy_1 : 0 ))$ |

Algorithm 1: Algorithm for full-adder

The BDD generated by the BitC compiler after imposing the normalizing conditions is shown in Figure 3.7.

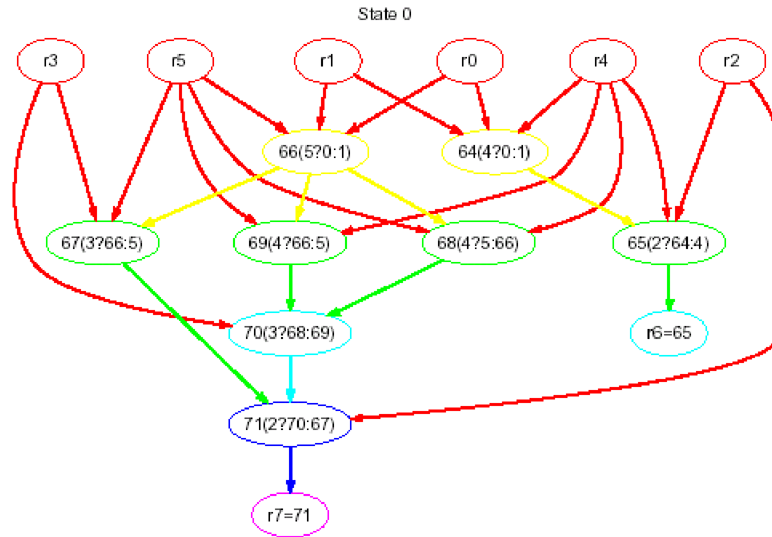



Figure 3.8: If-Then-Else DAG representing the full-adder program

The output generated by the BitC compiler after compiling the shown example program is `30 ITEs Created, 10 Kept`, where the number of ITEs created relates to the number nodes in the intermediate DAG and the number of ITEs kept denotes the number of nodes in the final DAG i.e the number of live ITEs at the end of computation. In the KITE architecture the number of ITEs left alive at the end of compilation signifies the number of ITE operations that need to be performed by the KITE processor.

### 3.3.2 Normalization

Normalized form or Canonical form of a Boolean function is any standard representation in which all logically equivalent expressions have an identical representation. The technique of Boolean expression normalization involves imposing restrictions on the order of variable manipulation and some rules that identify and factor equivalent sub-expressions.

In the BitC compiler, after the word-level operations are converted to their equivalent bit-level ITE forms they are subject to BDD normalizing techniques and/or ITE DAG transformations from [Kar88a]. Usually, BDD normalization is carried out for making tests like equivalence and satisfiability be just an equality check on the pointers, i.e. have unitary cost. In the BitC compiler, the primary goal of applying the normalization rules is to have a BDD representation which has a small number of nodes at its output.

#### BDD Normal Form

The canonical form for Binary Decision Diagrams was originally proposed by Bryant [Bry86]. The Bryant Normal Form imposed two restrictions on the nodes of a BDD. The first of the restrictions requires the atom in each node of the BDD to be earlier in order than its high and low children. The second restriction requires each node of a BDD to represent non-equivalent expressions.

Karplus [Kar88a] introduced a strong canonical form version of Bryant's Normal Form in which different nodes in a BDD represented different expressions rather than just non-equivalent expressions as originally implemented. Karplus also introduced the use of an If-Then-Else Operator for manipulating BDDs instead of the standard Boolean Operators. His work also listed a standard set of simplification and recursion stopping conditions which can be applied when recursively simplifying a BDD represented using an If-Then-Else Operator. Algorithms 2 and 3 list these simplifying conditions and recursion stopping

conditions with the *if-* part represented by *a*, the *then-* part represented by *b* and the *else-* part being represented by *c* in an If-Then-Else operator.

---

Algorithm 2: Special Cases for If-Then-Else simplification [Kar88a] of `if` (a) `then` (b) `else` (c)
If *a* = TRUE, return *b*.
If *a* = FALSE, return *c*.
If *b* = *c*, return *b*.
If *b* = TRUE and *c* = FALSE, return *a*.
If *c* = TRUE and *b* = FALSE, return ¬*a*.

---

Algorithm 3: Recursion stopping conditions for simplifying BDDs and ITE DAGs [Kar88a] represented using ITE operators of the format `if` (a) `then` (b) `else` (c)
If *a* = *b*, then replace *b* with TRUE.
If *a* = *c*, then replace *c* with FALSE.
If *a* = ¬*b*, then replace *b* with FALSE.
If *a* = ¬*c*, then replace *c* with TRUE.

---

The BitC compiler applies the stated normalizing restrictions, simplifying conditions and recursion stopping conditions to the Boolean equivalent of the word-level program during the code optimization phase. The function which accepts the *if-* , *then-* and *else-* representations of the word-level operations and subjects them to normalizing rules is called *mkite*, which is described in the next section.

### The *mkite* Function

The *mkite* function accepts the bit-level ITE equivalents of the word-level operations and subjects them to the normalizing rules recursively. Consider the case of two-bit addition demonstrated in Section 3.3.1. The corresponding ITE representation would be:

$$c_0 \leftarrow (a_0 \,?\, (b_0 \,?\, 0 : 1) : b_0)$$

$$c_1 \leftarrow ((a_1 \,?\, (b_1 \,?\, 0 : 1) : b_1) \,?\, ((a_0 \,?\, b_0 : 0) \,?\, 0 : 1) : (a_0 \,?\, b_0 : 0))$$

The bits $c_0$ and $c_1$ are computed by calling *mkite* function for each ITE tuple shown in the above representation. The *mkite* functions enforces the normalization conditions from Bryant Normal form on the atoms of the ITE. A *trivial* atom in an ITE is an atom that is a part of the variables in the input expression. A *nontrivial* atom is the atom that has a recursive representation using the previously generated ITEs.

A call to *mkite* for generating ITE representation for the bit $c_0$ would be of the form,

$$\texttt{mkite}(a_0,\texttt{mkite}(b_0,0,1),b_0)$$

While the *mkite* function can apply BDD normalizing conditions and/or ITE DAG transformations, we discuss the *mkite* function which applies BDD normalizing rules here to lead us into some of the preliminary techniques discussed in Chapter 5. The techniques in which the *mkite* function uses ITE DAG transformations instead of normalizations, uses combinations of ITE DAG transformations and BDD normalizations etc are discussed in subsequent sections.

The *mkite* function (Algorithm 4) performs a number of standard optimizations on the If, Then and Else parts passed to it as inputs to convert the final BDD into a canonical representation. First it applies a standard set of simplifications that check for and eliminate redundant logic. Then the algorithm applies a set of recursion stopping conditions borrowed from the If-Then-Else DAG minimization rules proposed by Kevin Karplus [Kar88a]. Although the ITE DAGs here do not implement negated references, the processing does recognize the (more complex) cases involving negation using *eqnot*. Then the normalization routine chooses the smallest node among the 3 nodes that are given to it to constitute the *if*-part, with the *then*- and the *else*- parts determined by recursive DAG walk. So, the *if*- parts are restricted to trivial atoms where as the *then*- and the *else*- parts can have a non-trivial node when necessary.

*A* hash table (*uniq*) accepts ITEs from the *mkite* routine, creates new entries for unique ITEs and returns a pointer to an existing record whenever the ITE already exists. The nor-

malizing restrictions are such that the variable order determines the amount of factorization in the ITE DAG generated by the ITE (target code) generator and hence its size.

---

Algorithm 4: The *mkite* function

```
int mkite(register int a,
register int b,
register int c)
{
register int am, bm, cm, m, left, right;
/* Simplifications */
if (a == b) b = ITE1;
if (a == c) c = ITE0;
if (eqnot(a, b)) b = ITE0;
if (eqnot(a, c)) c = ITE1;
/* Stopping conditions */
if (a == ITE1) return(b);
if (a == ITE0) return(c);
if (b == c) return(b);
if ((b == ITE1) && (c == ITE0)) return(a);
/* Recursive normalization */
am = (ITEISVAR(a) ? a : ites[a].a);
bm = (ITEISVAR(b) ? b : ites[b].a);
cm = (ITEISVAR(c) ? c : ites[c].a);
m = ((am > bm) ? am : bm);
m = ((m > cm) ? m : cm);
left = mkite(splitleft(a,m),
splitleft(b,m),
splitleft(c,m));
right = mkite(splitright(a,m),
splitright(b,m),
splitright(c,m));
if ((m != a) || (left != b) || (right != c))
return(mkite(m, left, right));
return(uniq(m, left, right));
}
```

---

The optimizations are recursively performed because the normalizing conditions could potentially expose ITE forms that could be factored by one of the initial conditions.

# CHAPTER 4: THE VARIABLE ORDERING PROBLEM

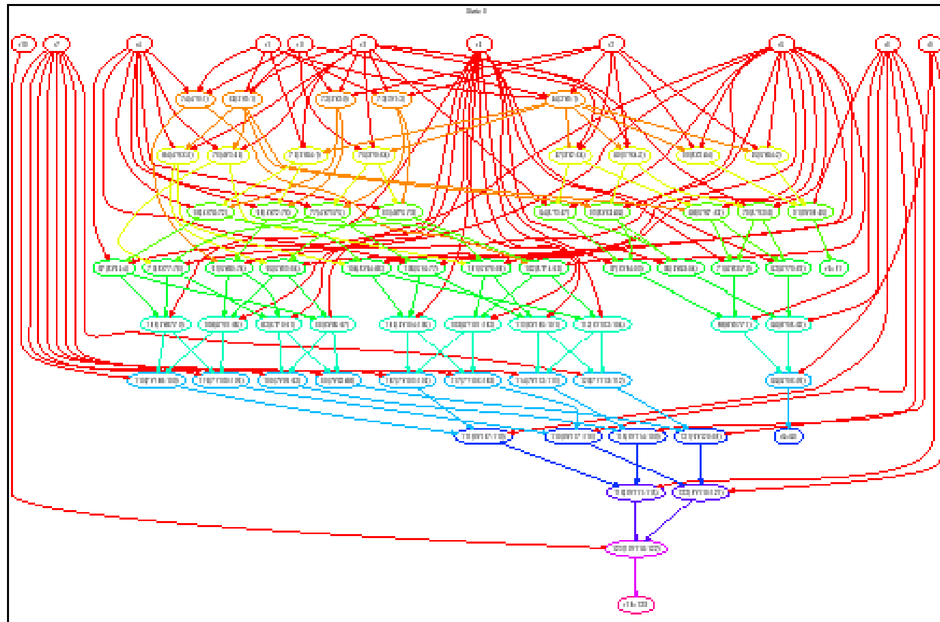
This chapter explains the impact of the BDD variable ordering problem on nanocontroller compilation and discusses some existing work addressing the BDD variable ordering problem.

## 4.1 Impact Of Variable Ordering

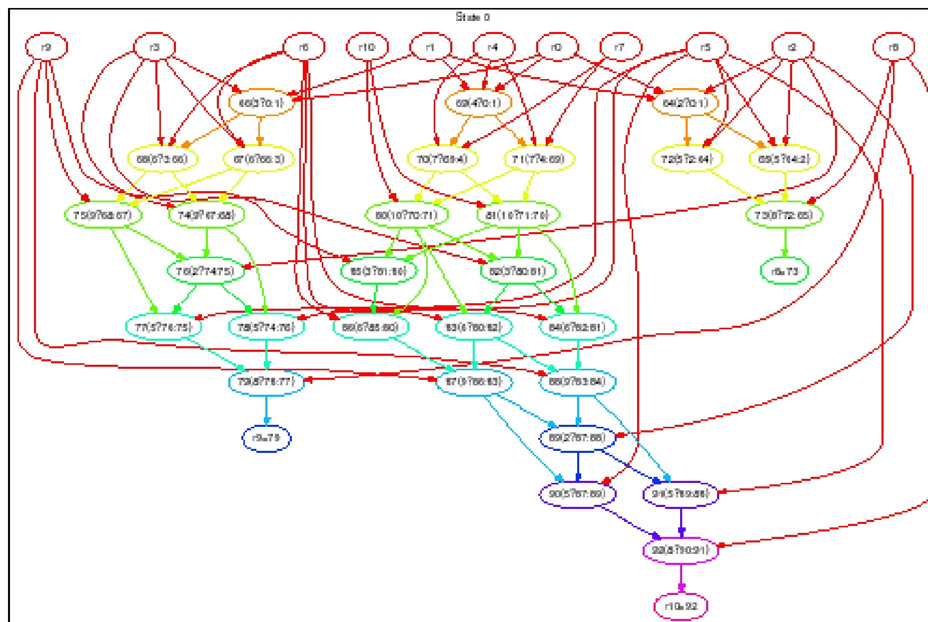
The order of variables in a BDD can favorably or adversely impact BDD complexity in terms of the number of nodes and the amount of memory required to develop the representation. The difference between BDD complexity with good and bad orderings can be exponentially large; in practice, the difference might not be exponentially large, but ranges from a few percent to about a factor of two. For example, using a good variable ordering vs. using a bad variable ordering in the BitC compiler produced the two ITE DAGs shown in Figure 4.1. With the word-level to bit-level transformations inherently generating large basic blocks even for small programs, this complexity difference is very significant. The size of the DAG determines the amount of time required to compute the result of a Boolean function in hardware, which influences the response time of the nanocontroller. The amount of memory required to develop a DAG representation impacts the complexity of control algorithms that can be used with the nanocontroller.

ITE DAG complexity also has an indirect impact on register allocation. The KITE Architecture provides a limited number of registers and no other memory on the nanocontroller chip. Thus, the on-chip registers are the only means of storage in a nanoprocessor. These registers store the live and temporary ITEs during the computation of a Boolean function in addition to storing control and state information. This keeps the processor complexity low but the pressure is now on the compiler to reduce the number of DAG nodes that require

storage in the registers and to perform efficient register allocation for the nanocontroller to be functional.



(a) default variable order




(b) reversed variable order

Figure 4.1: Figure illustrating the difference in DAG complexity with difference in variable order while computing `int:3 a, b, c;c=a+b+c;`

## 4.2 Combating BDD Complexity Explosion In Nanocontrollers

Finding a variable order that produces a BDD with the least possible number of nodes can help the nanocontroller function with a fast response time, accuracy and to perform complex control operations. It also helps the BitC compiler to develop representations for controller programs without the computer host running out of memory space. There are several heuristics available for variable ordering as will be seen in the following section. The techniques implemented in this thesis are based on:

1. Applying GA based heuristics to find the variable order that ensures the most compact final DAG.
2. Applying Code Factoring Transformations which expose possibilities for factorization of ITE tuples.
3. Combinations of the above.

## 4.3 Variable Ordering Heuristics: Existing Work

There are several existing algorithms designed to find the optimal BDD variable order for minimizing completely specified Boolean functions. A few algorithms which are relevant to this work are reviewed here.

### 4.3.1 Algorithms based on variable exchange

There are two main algorithms based on exchange of neighboring variables in a BDD. These algorithms are based on the observations of Fujita *et al* [FMK91], Ishiura *et al* [N.I91], and Rudell [Ric93] that an adjacent variable swap in a BDD has a predictable complexity.

#### Window permutation

Window permutation algorithm starts with choosing a window of size $k$ which is used to scan all permutations among the $k$ adjacent variables starting at a particular level in the DAG. Permutations inside a window are done using adjacent variable swap and the DAG size is recorded every time. The best permutation of variables is restored at the end of the search.

#### Sifting

Sifting starts with a sorted variable order in which the Boolean variables are arranged in a descending order based on the number of nodes at each level of the DAG. In this technique each variable is moved up and down in the order using adjacent variable swaps and the size of the DAG is recorded at each position. Finally the variables are moved to their locally optimal position which is selected based on the best DAG size recorded during this search. The worst case complexity of sifting for variable ordering over $n$ variables is $O(n^2)$, which is typically controlled by terminating the search in a particular direction when the DAG size grows to twice its original size. Many genetic algorithms for variable ordering (see Section 4.3.2) use sifting on each population member, after applying genetic operators, to find the new local minima.

### 4.3.2 Genetic Algorithms

The various Genetic Algorithms (GA) found in the literature differ primarily in their recombination operators or their fitness metric. The GA approach suggested by Drechsler *et al* [Dre96] uses partially matched crossover (PMX) and mutation as the genetic operators. The PMX operator was found to completely modify the ordering which often lead to explosive increases in DAG complexity. A later work by Drechsler *et al* [DG97] suggested

the use of Inversion and Sifting as genetic operators with a selection probability of 0.5 for each operator. When Inversion was selected, it was followed by a SIFT operation to obtain a new local minimum for the search.

A work by Wolfgang *et al* [LB05] suggests the use of alternating crossover with sifting as a hybridization technique in the genetic algorithm for variable ordering. The fitness for this GA is calculated based on a distance graph, in which the nodes are variable orders and the edges are labelled with the minimum number of swaps necessary to transform one order to another.

All existing work tries to find the optimal variable order that minimizes a BDD while still maintaining a canonical form. Our objective differs from the above in that we use heuristics to find the best variable order that minimizes the number of nodes in a BDD but we care less about canonicity of the final DAG.

# CHAPTER 5: PRELIMINARY TECHNIQUES

This chapter discusses some of the initial techniques the results from which led us to design some GA-based heuristics.

## 5.1 Original Bit Order

In the first version BitC compiler, the bits in the variables declared in the input program were ordered with the lower bits getting lower order number compared to the higher bits. This variable ordering is shown in Figure 5.1. As explained in the previous section the bit order of a variable influences the walk order of the BDD during normalization.

int:2 a,b,c;

| $a_0$ | $a_1$ | $b_0$ | $b_1$ | $c_0$ | $c_1$ |
|---|---|---|---|---|---|
| 2 | 3 | 4 | 5 | 6 | 7 |

**Bit Order :** $a_0 < a_1 < b_0 < b_1 < c_0 < c_1$

Figure 5.1: Figure illustrating the default bit order for variables

## 5.2 Reverse Bit Ordering Technique

Intuitively, because many of the operations performed on multi-bit data involve carry, it seems reasonable to reorder the bits according to how early they would appear in a carry chain involving all variables. This is the ordering that we refer to as reversed, because higher bit positions are given lower-numbered positions in the order (i.e., they appear closer to the top of the ITE DAGs as drawn in this thesis). For example, the default order shown in Figure 5.1 becomes the order given is Figure 5.2.

int:2 a,b,c;

| $a_0$ | $a_1$ | $b_0$ | $b_1$ | $c_0$ | $c_1$ |
|---|---|---|---|---|---|
| 5 | 2 | 6 | 3 | 7 | 4 |

**Bit Order :** $a_1 < b_1 < c_1 < a_0 < b_0 < c_0$

Figure 5.2: Figure illustrating the reversed bit order for variables

### 5.2.1 Results

A perl script was used to generate 2,000 random test cases which involved operators like `+`, `-`, `*`, `/`, `^`, `&`, `|`, `%`, `<<`, `>>`, and `%*` (which means multiply with result same precision as operands). A maximum of four variables of up to 5-bit precision each were used, with a maximum of 3 operators per basic block. All declared variables within each block had the same bit precision. The Figure 5.3 shows the number of live ITEs constituting the final DAG after applying a reversed bit priority to the Boolean variables input to the normalization routine.

The plot shows that normalizing the BDD with reversed variable order provides an average 5.72% reduction in the number of nodes as compared to using the default bit order. Clearly, the reversed bit order is a worthwhile improvement upon the default order, because imposing this order has virtually no additional compiler overhead – the reversed order would be a better default order.

## 5.3 Print Form Transformation

Another optimization that can be used in the BitC compiler is not a normalization order constraint at all, but a more conventional compiler transformation that is based on pattern matching and does not result in a normalized form. Print Form Transformation (PFT) is a

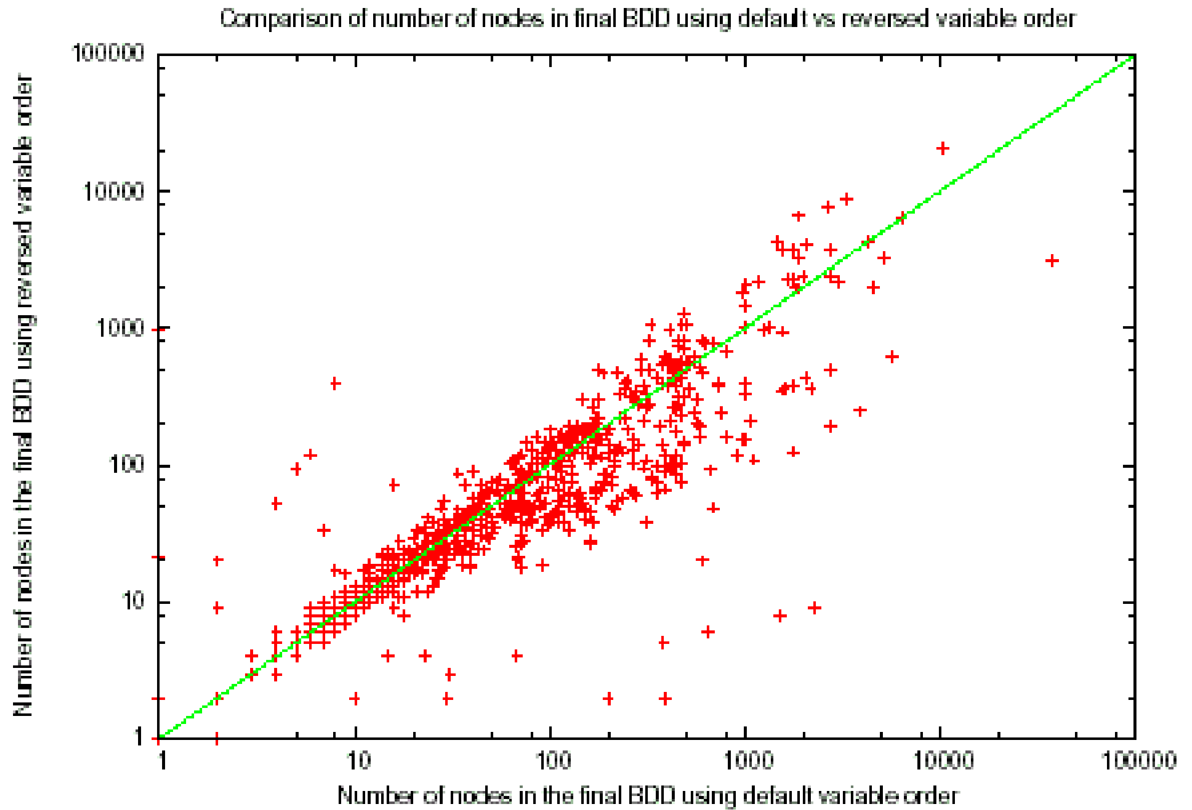



Figure 5.3: Scatter plot comparing the number of live ITEs in the final BDD using original bit order vs. reversed bit order

method suggested by Kevin Karplus [Kar88a] to make If-Then-Else DAGs easier to read when printed. The normal form of an If-Then-Else DAG is required to have all nodes of the if- part be earlier in order than all nodes of the then- and the else- parts. The print form transformations listed in Table 5.1 are a set of patterns by which nodes in an If-Then-Else DAG can be rewritten to increase the number of times the constants TRUE and FALSE are referenced. By this rearrangement of the If-Then-Else DAG, which is a fairly complex and recursive compiler algorithm in itself, the atoms of the if- part are made disjoint from the atoms of the then and the else part. The final DAG loses the ordering constraint used for normalization and hence is non-canonical. In fact, even changing the order of application of the individual PFT rewrites can produce different result ITE DAGs.

In the KITE Architecture, the constants FALSE and TRUE are hardwired as registers zero

and one respectively. Increasing the number of times TRUE and FALSE appear in the DAG increases the number of times a hardwired register is being referenced, thus tending to reduce the number of references to temporary values in registers (which is closely related to minimizing maxlive). Applying this technique to the optimizing portion of the BitC compiler generates If-Then-Else DAGs which have a smaller intermediate DAG size and a larger final DAG size when compared to the BDDs obtained from Bryant Normal Form. Note that the output produced by this technique is an If-Then-Else DAG that is not equivalent to a BDD, as it does not meet the basic structural requirement that the if- part of a BDD node always must be trivial.

Table 5.1: The Print Form Transformation

| Existing Expression | Transformed Expression |
|---|---|
| **Transformations involving 3 subDAGs** | |
| (*if* a *then* (*if* $b_a$ *then* $b_b$ *else* FALSE) *else* $b_a$) | (*if* (*if* a *then* $b_b$ *else* TRUE) *then* $b_a$ *else* FALSE) |
| (*if* a *then* (*if* $b_a$ *then* $b_b$ *else* $b_b$) *else* $b_a$) | (*if* (*if* a *then* $b_b$ *else* TRUE) *then* $b_a$ *else* $b_a$) |
| (*if* a *then* (*if* $b_a$ *then* $b_b$b *else* TRUE) *else* $b_a$) | (*if* (*if* a *then* $b_b$ *else* TRUE) *then* $b_a$ *else* TRUE) |
| (*if* a *then* (*if* $b_a$ *then* $b_b$ *else* $b_b$) *else* $b_a$) | (*if* (*if* a *then* $b_b$ *else* TRUE) *then* $b_a$ *else* $b_a$) |
| (*if* a *then* $c_a$ *else* (*if* $c_a$ *then* $c_b$ *else* FALSE)) | (*if* (*if* a *then* $c_b$ *else* TRUE) *then* $c_a$ *else* FALSE) |
| (*if* a *then* $c_a$ *else* (*if* $c_a$ *then* $c_b$ *else* $c_b$)) | (*if* (*if* a *then* $c_b$ *else* TRUE) *then* $c_a$ *else* $c_a$) |
| (*if* a *then* $c_a$ *else* (*if* $c_a$ *then* $c_b$ *else* TRUE)) | (*if* (*if* a *then* $c_b$ *else* TRUE) *then* $c_a$ *else* TRUE) |
| (*if* a *then* $c_a$ *else* (*if* $c_a$ *then* $c_b$ *else* $c_b$)) | (*if* (*if* a *then* $c_b$ *else* TRUE) *then* $c_a$ *else* $c_a$) |
| **Transformations involving 4 subDAGs** | |
| (*if* a *then* (*if* $b_a$ *then* $b_b$ *else* $b_c$) *else* $b_c$) | (*if* (*if* a *then* $b_a$ *else* FALSE) *then* $b_b$ *else* $b_c$) |
| (*if* a *then* (*if* $b_a$ *then* $b_b$ *else* $b_c$) *else* $b_b$) | (*if* (*if* a *then* $b_a$ *else* TRUE) *then* $b_b$ *else* $b_c$) |
| (*if* a *then* $c_b$ *else* (*if* $c_a$ *then* $c_b$ *else* $c_c$)) | (*if* (*if* a *then* $c_a$ *else* TRUE) *then* $c_b$ *else* $c_c$) |
| (*if* a *then* $c_c$ *else* (*if* $c_a$ *then* $c_b$ *else* $c_c$)) | (*if* (*if* a *then* $c_a$ *else* FALSE) *then* $c_b$ *else* $c_c$) |
| **Transformations applied only if result is simpler** | |
| (*if* a *then* (*if* $b_a$ *then* $b_b$ *else* $b_c$) *else* (*if* $b_a$ *then* $c_b$ *else* $c_c$)) | (*if* $b_a$ *then* (*if* a *then* $b_b$ *else* $c_b$) *else* (*if* a *then* $b_c$ *else* $c_c$)) |
| (*if* a *then* (*if* $b_a$ *then* $b_b$ *else* $b_c$) *else* (*if* $b_a$ *then* $c_b$ *else* $c_c$)) | (*if* $b_a$ *then* (*if* a *then* $b_b$ *else* $c_c$) *else* (*if* a *then* $b_c$ *else* $c_b$)) |

### 5.3.1 Results

A perl program was used to generate 2,000 random test programs and the results were plotted – the same 2,000 test cases that were used to evaluate the default and reversed

ordered normalizations. The results showed that applying PFT decreased the number of temporary ITEs created during the optimization phase when compared to the normalization techniques, but at the expense of an increased the number of live ITEs in the final DAG. Figure 5.4 shows a marked decrease in the size of the intermediate DAG when applying PFT to the Boolean functions. Figure 5.5 shows the increase in the size of final DAG when applying PFT.

Generating a smaller intermediate DAG is our secondary goal, as small intermediate DAGs can prevent the compiler from running out of memory when trying to generate the BDD for a Boolean function. Thus, although PFT is less effective than normalization, it may be a viable method for handling program fragments for which the normalization processing required infeasibly many intermediate nodes.

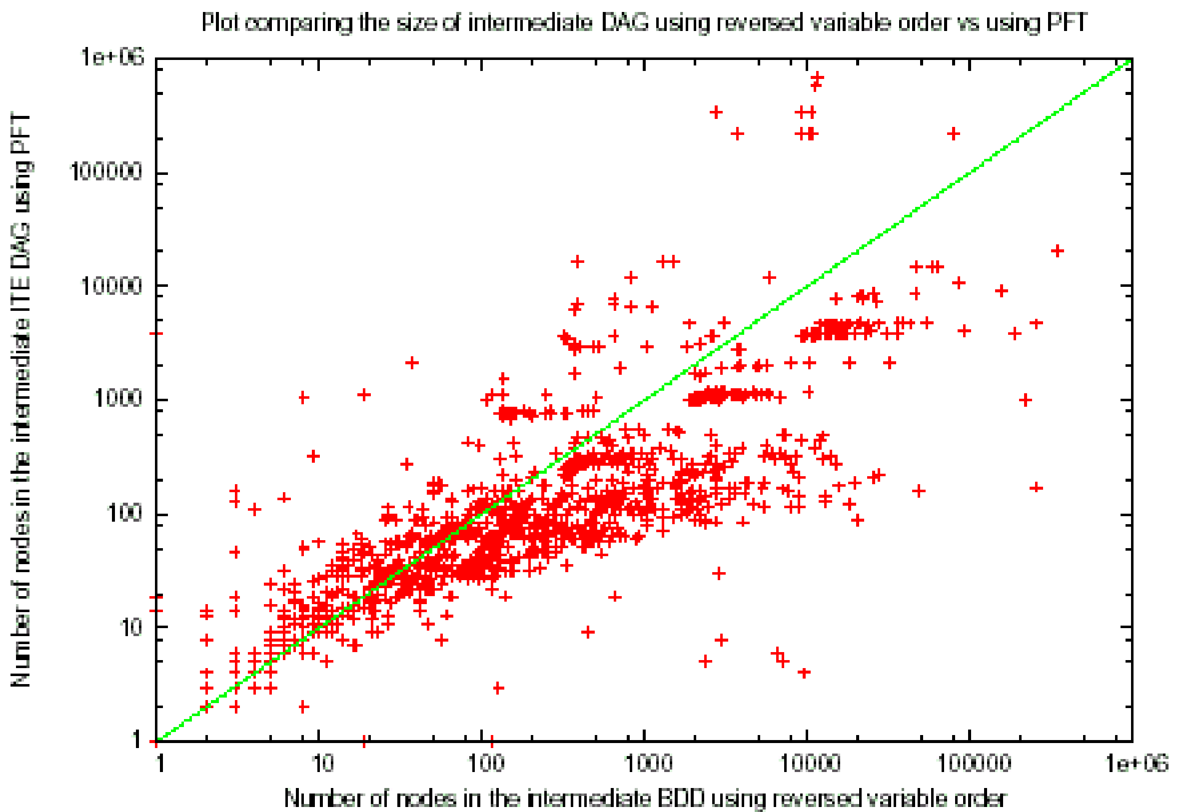



Figure 5.4: Scatter plot comparing the number of nodes in the intermediate ITE DAG when using PFT as the optimization technique vs. the number of nodes in the intermediate BDD using normalization as the optimizing technique

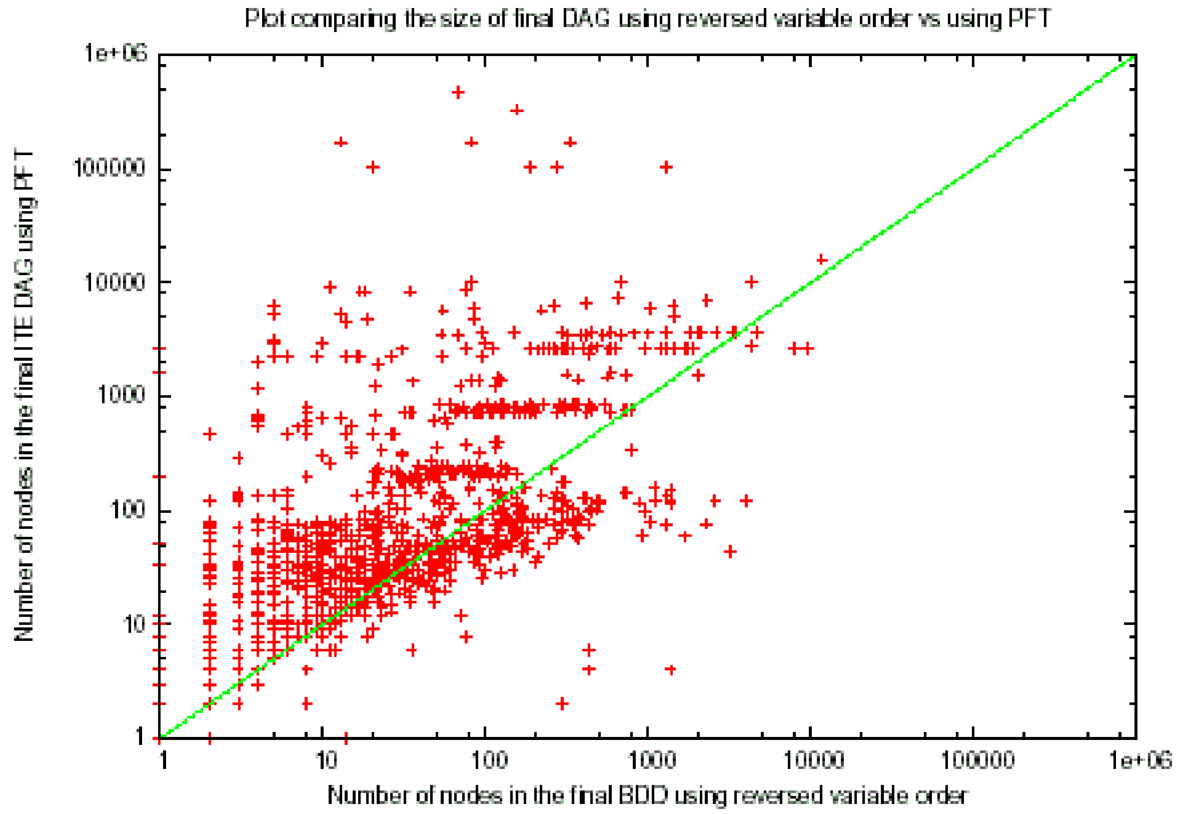



Figure 5.5: Scatter plot comparing the number of nodes in the final ITE DAG when using PFT as the optimizing technique vs. number of nodes in the final BDD when using normalization as the optimizing technique

# CHAPTER 6: GA FOR FINDING OPTIMAL VARIABLE ORDER

The Genetic Algorithm (GA) is an adaptive heuristic search technique based on the concepts of evolution and natural selection. During code optimization in the BitC compiler, a genetic algorithm is used to search for the best possible variable order in terms of minimizing the number of number of live If-Then-Else statements in the generated code.

## 6.1 GA Details

The various GA parameters used when finding an optimal variable order for the BDD are listed in this section.

### 6.1.1 Genome

The genome describes a population member that participates in the evolutionary process. Each genome encodes the problem specific data in it. This data gets transformed through a number of generations to produce a solution to the problem. Although the phenotype for this problem is a variable ordering, the genotype used is slightly more abstract. The genome for our problem is a string of integers that, when sorted into increasing order, rearrange the variables into the order of the corresponding phenotype. A genome is permitted to have duplicate integer values within it, with the resulting phenotype order determined by how the sorting procedure breaks ties. The initial population consists of randomly-generated genomes. The GA searches for a solution in this subset of variable orders using the evolutionary techniques of mutation and crossover.

### 6.1.2 Fitness

Fitness is a measure of goodness of a genome in a population of genomes. This measured fitness of a genome decides its evolutionary path. For the variable ordering problem, fitness is measured in terms of the number of live ITEs left at the end of compilation. The fittest genome results in the least number of nodes in the final DAG produced by normalization using the corresponding phenotype order.

### 6.1.3 Selection

Selection is a mechanism that decides which individuals get propagated to the subsequent generations. This GA uses tournament-based selection of genomes. Tournament-based selection increases a fit individual's probability of survival. Individuals are partially sorted by conducting tournaments with other genomes in the population. The most fit individuals in the current generation tend to move toward the top of the population, while the less fit individuals move toward the bottom of the population. The bottom portion of the population is replaced by new individuals created by using the genetic operators of mutation and crossover, thus making survival probability higher for the most fit individuals. The organization of the population in this way offers the additional benefit that only the bottom portion of the population data structures will hold new individuals and they may be evaluated in a sequence that has inherently good cache locality.

### 6.1.4 Mutation

Mutation is a genetic operator that helps add diversity to a pool of genomes, thus helping to prevent the population from getting stuck at a local minimum. For the bit-ordering problem, *random multiple-point* mutation has been employed. Random mutation replaces randomly selected bit positions (integer values) from a parent genome with new randomly

generated values.

### 6.1.5 Crossover

Crossover is the genetic operator that creates new genomes by combining characteristics from two parents that are better in fitness than the replaced genome. For the bit-ordering problem, *multiple point cross-over* has been employed where a less fit genome gets replaced by another genome that is a result of crossing-over better fit parents at multiple points. The amount of crossover and mutation depends on the user-specified cross over and mutation rates.

### 6.1.6 Steady-State Vs. Generational Genetic Algorithm

GAs can be either "steady state" or "generational" in nature. Generational GAs create an entire population at a time whereas steady state GAs replace a single individual at at a time. There are benefits to both approaches. The approach used here is essentially a generational GA, but has many of the benefits of steady state GAs in that it does not re-evaluate population members that survive from one generation to another.

## 6.2 Algorithm Description

The GA for variable ordering starts with generating a preset number of random integer strings to represent the variable orders in the initial population. Word-level operations in the input program are translated into their bit-level equivalents and expressed using the *If-Then-Else* Operator. These bit-level ITE operations are subject to the normalization algorithm using the variable order encoded in every population member. The fitness of a variable order is measured using the number of nodes in the final BDD it generates after the normalization algorithm. A small number of nodes in the final BDD represents a variable

order with high fitness and hence a high probability of survival.

The population at the end of a generation is partially sorted based on the fitness of each genome; the better fit genomes are sorted towards the top of the population and less fit genomes are towards the bottom. The less fit genomes are tweaked through the processes of mutation and crossover to produce children which replace the parents in the gene pool. Mutation is carried out by random replacement of bit order of a less fit population member. Crossover is carried out by multiple point crossover between two population members that are higher up in the sorted order to replace a less fit member. Now this new set of genomes that are slightly better than the previous generation forms the population of the next generation. This procedure is repeated until a preset number of generations are reached.

---

1. Initialize a set random integer strings representing variable orders as the initial individuals in the population.
2. For a given word-level program generate the corresponding final ITE representation using the variable orders in the different population members.
3. Evaluate the fitness of each individual in this generation by using the number of ITEs in the generated code as the metric.
4. Store the fitness value of the fittest variable order string of this generation.
5. Partially sort the variable orders based on their fitness.
6. Apply random multiple point crossover between two randomly selected population members from the top of the sorted order and replace the variable orders which are at the bottom of the sorted order.
7. Apply random multiple point mutation on the individuals which are at the bottom of the sorted order.
8. Evaluate the fitness of this new generation.
9. Update the value for best fit individual.
10. Repeat from Step 2 until specified number of generations are reached.

Algorithm 5: Genetic Algorithm for Variable Ordering

---

### 6.3 Experimental Procedure

At the start of the GA, the bit-level operations expressed using *If-then-Else* operators are recorded in a table. These recorded ITEs together represent a non-canonical ITE DAG with each table entry corresponding to individual nodes of the ITE DAG. These nodes of the non-canonical ITE DAG are subject sequentially to the normalization routine using the variable orders in the current generation of the GA. The normalization routine converts each node of the initial ITE DAG into a BDD and stores the BDD in another table.

When ITEs are converted to BDDs, a few nodes in the original ITEs may get factored out to reference an existing node and a few other nodes may generate new BDD nodes. When sequentially normalizing the ITE entries in the first table, the future nodes which use a prior node in their if- , then- or else- parts need to be redirected to refer their BDD representation. For this purpose, a translation table is required between the recorded ITE operations in the first table and the table which stores the final BDD. The Figure 6.1 demonstrates the queuing and translation of ITEs occurring during the GA.

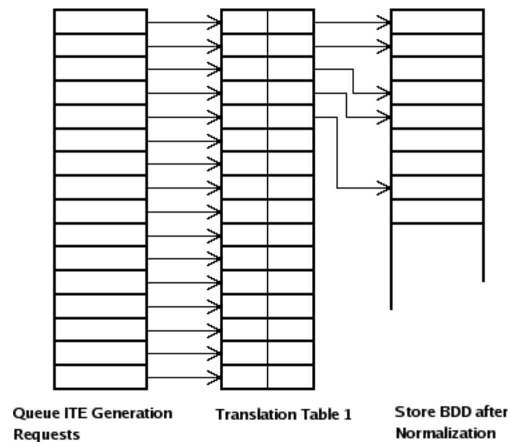



Figure 6.1: GA data structure diagram

Wolfgang *et al* suggested that a GA which uses alternating crossover and sifting as the hybridization technique performed better than other types of crossover techniques like order crossover, cycle crossover, partially matched crossover and inversion. Implementation of

alternating crossover in the GA used during BitC compilation did not show any uniform trend in the results.

## 6.4 Results

The GA was run over a population of 120 individuals for 30 generations with the probability of mutation set at 0.16 and the probability of crossover at 0.36. The results plotted in Figure 6.2 and Figure 6.3 are from 1,000 randomly generated test programs. Only 1,000 cases were tested because the GA runs fairly slowly, and 1,000 cases seemed to sufficiently clearly show the performance trends.

Each plot compares the number of nodes in the final BDD generated using the GA variable order to the number of nodes generated with default and reversed variable orders. When compared to the original bit order the GA shows a significant decrease in the size of the final DAG as seen in Figure 6.2. When compared to the reversed bit order the GA still shows some improvement in the size of the final DAG as shown by Figure 6.3. Thus, the GA ordering consistently yielded a modest improvement over the fixed orderings, and is a better method if compile time allows.

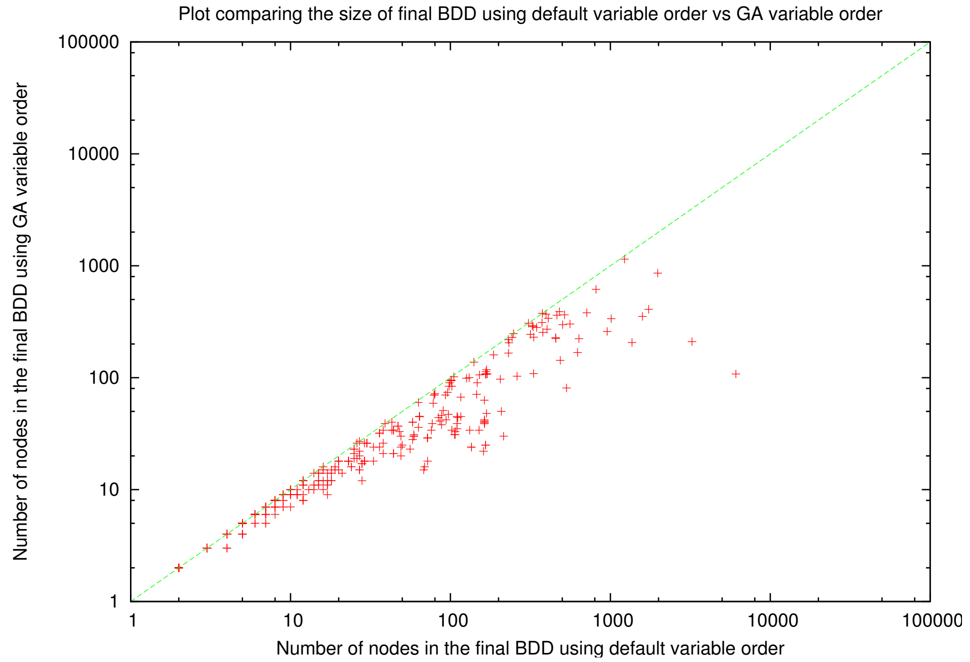



Figure 6.2: Plot comparing the number of nodes in the final BDD normalized using GA variable ordering vs. default variable ordering

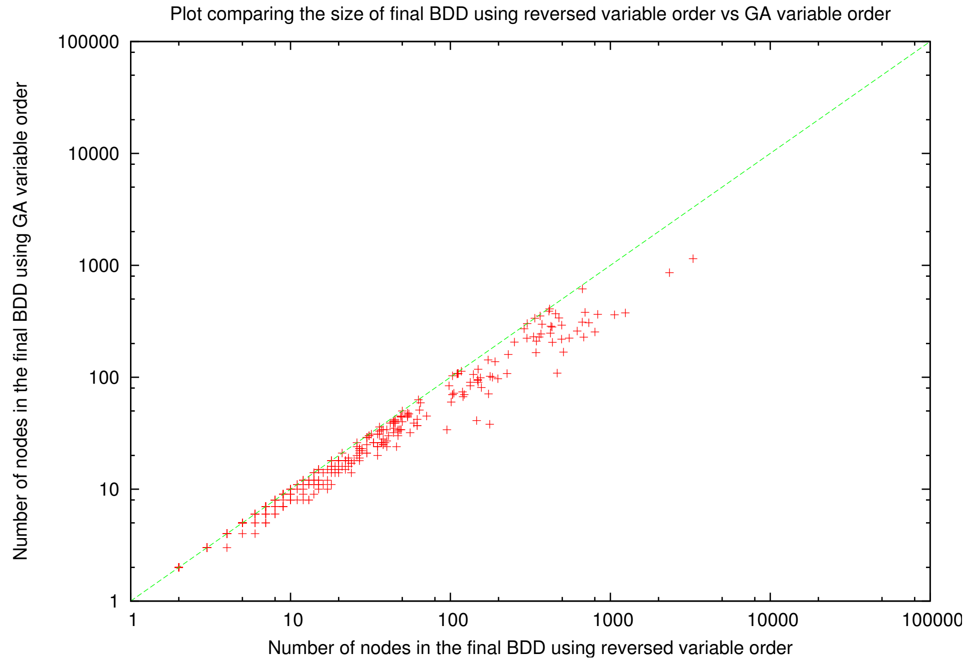



Figure 6.3: Plot comparing the number of nodes in the final BDD normalized using GA variable ordering vs. reversed variable ordering

# CHAPTER 7: BDD VARIABLE ORDERING GA WITH PFT AFTER NORMALIZATION

From the results in the Figure 5.3 and Figure 5.4 it can be inferred that a genetic algorithm for variable ordering can create huge intermediate BDDs in the process of creating a final minimized logic. In contrast, PFT has a smaller intermediate DAG size though it results in a final ITE DAG that is significantly larger. During our testing, very large intermediate BDDs were created for arithmetic operations which involved >15 bits and caused the GA to run out of memory space. These observations led us to try a combination of these techniques that could potentially optimize both the number of ITEs created (size of the intermediate BDD) and the number of ITEs kept (size of the final BDD).

## 7.1 Description

While the normalization routine builds BDDs which are in canonical form the PFT converts some BDDs to If-Then-Else DAGs on recognizing identities. Applying normalization after print form transformation in the optimizing portions of the compiler can help reduce the size of intermediate DAGs due to additional factoring happening in the if part due to BDD to If-Then-Else DAG conversion. This conversion can provide more opportunities for factoring which can lead to smaller final DAGs. By introducing the normalization routine followed by print form transformation be a part of a GA search for optimal variable order, many possibilities for optimization are opened up. Algorithm 4 provides the details on this genetic algorithm.

## 7.2 Experimental Procedure

The bit-level ITE equivalents of the word-level operations in the input program to the compiler, are recorded in a table. The nodes of this non-canonical ITE DAG are played back in a sequence to the normalization routine. The generated canonical BDD node(s) are stored in a hash table. The BDD nodes are immediately subject to PFT to recognize any nodes which can be transformed to ITE DAGs by factorization. The non-canonical final ITE DAG is stored in another hash table. Two translation tables are required by this algorithm, one for redirecting node references between the initial ITE DAG and the normalized BDD and another between the normalized BDD and the final non-canonical ITE . The data structures and data flow used in this optimizing part of the compiler is shown in Figure 7.1 .

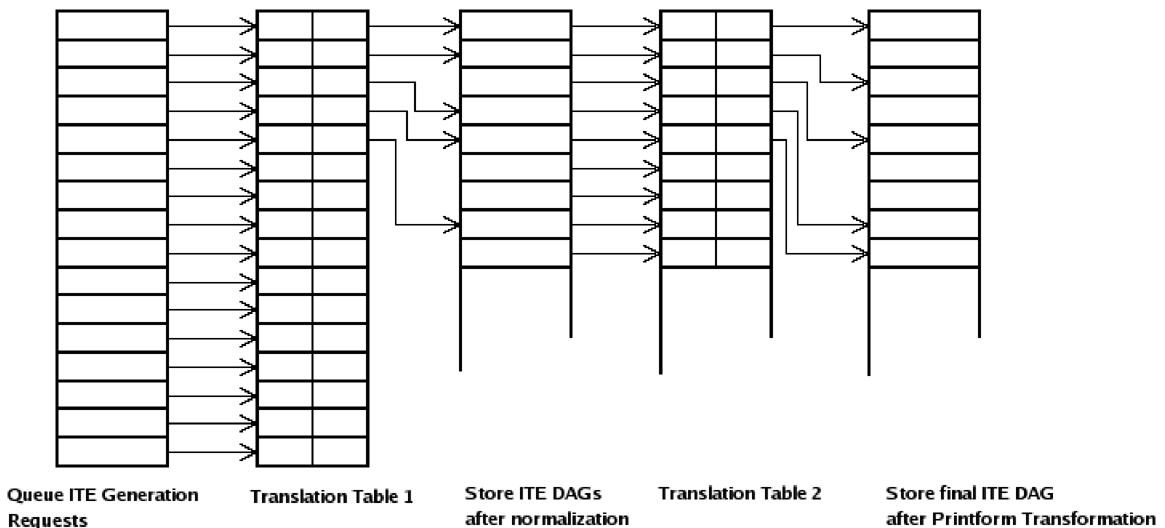



Figure 7.1: The data structures and data flow used in the algorithm for GA fitness calculation

### 7.3 Results

The use of PFT after normalization has not proved to be advantageous in the number of live ITEs or the number of intermediate ITEs. This method leaves many DAGs which are larger in size than the BDDs generated by the GA in chapter 6. It produces some reduction in the number of ITEs but the number of instances of this decrease in DAG size is very limited compared to the number of instances in which the DAG size increases in comparison with our first GA.

In summary, this technique was not effective and probably should never be used in BitC – a surprising result given that Karplus envisioned PFT would be used after normalization in essentially this way. The poor performance might be due to the differences between Karplus's ITE DAGs and Kentucky ITE DAGs, but that determination is outside of the scope of this thesis.

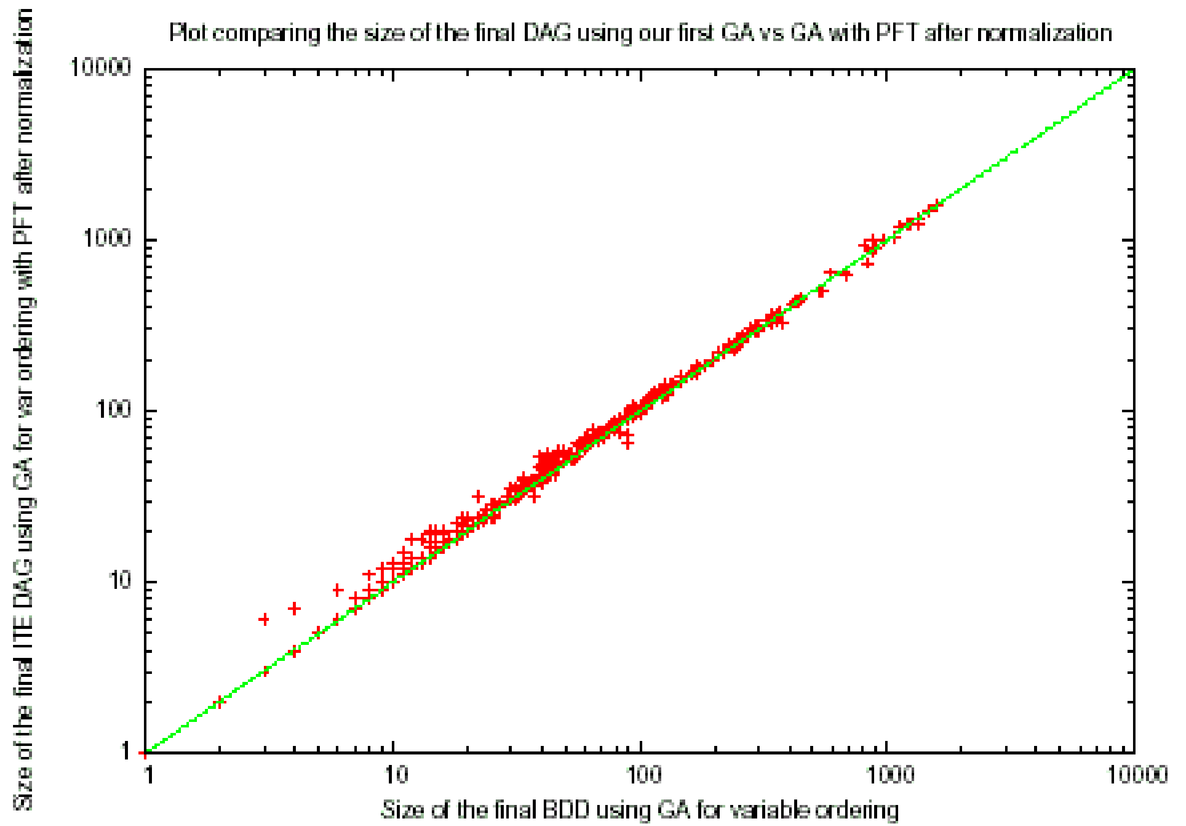



Figure 7.2: Scatter Plot final ITE DAG size when using GA for variable ordering with PFT after normalization vs. size of the final BDD when using our basic GA for variable ordering

1. Initialize a set random integer strings representing variable orders as the initial individuals in the population.

2. For a given word-level program generate the bit-level ITE representation by issuing calls to the *mkite* routine.

3. Queue the requests for generating ITEs in a table.

4. Evaluate the fitness of each individual in this generation by going through steps 5 through 8.

5. Subject the queued ITEs to the normalization algorithm. Use a translation table to translate between the queued ITEs and the newly generated normal ITEs.

6. Store the normalized ITEs in a table.

7. Subject the normal ITEs to the print form transformation. Use a translation table to translate between the normal ITEs and the newly generated ITEs.

8. Store the number of ITEs in the final DAG in a table to serve as the fitness metric.

9. Partially sort individuals based on their fitness.

10. Apply random multiple point crossover between two randomly selected population members from the top of the sorted order and replace the variable orders which are at the bottom of the sorted order.

11. Apply random multiple point mutation on the individuals which are at the bottom of the sorted order.

12. Update the value for best fit individual

13. Repeat from Step 3 until specified number of generations are reached

Algorithm 6: Genetic Algorithm for variable ordering with ITE normalization carried out before PFT

# CHAPTER 8: BDD VARIABLE ORDERING GA WITH PFT BEFORE NORMALIZATION

Since the results from the previous section showed that applying PFT after normalization is disadvantageous the next step was to apply PFT before normalization.

## 8.1 Description

The assumption behind this technique is that by first converting a function to ITE DAGs, which are meant to introduce a lot of factorization and then applying normalizing rules which convert the ITE DAGs to BDDs, the normalization stage would have a fewer ITEs to start with. When normalization starts with a fewer calls to *mkite* then it is reasonable to expect the technique to have reduction in the number of live ITEs in the generated code. The natural expectation is that the final normalized form would have the same complexity in either case. Algorithm 5 provides the details on this genetic algorithm.

## 8.2 Experimental Procedure

The recorded ITEs equivalents of the word-level input program are first subject to Print-Form transformation and are later subject to normalization using technique. The initial non-canonical DAG is further factorized using PFT, which generates a small ITE DAG to start with for the normalization algorithm when compared to the size of the DAG available to the normalization algorithm in the GA described in chapter 6. This highly factored non-canonical ITE DAG gets converted to a canonical BDD after normalization. The data structures and data flow involved in this procedure is depicted in Figure 8.1.

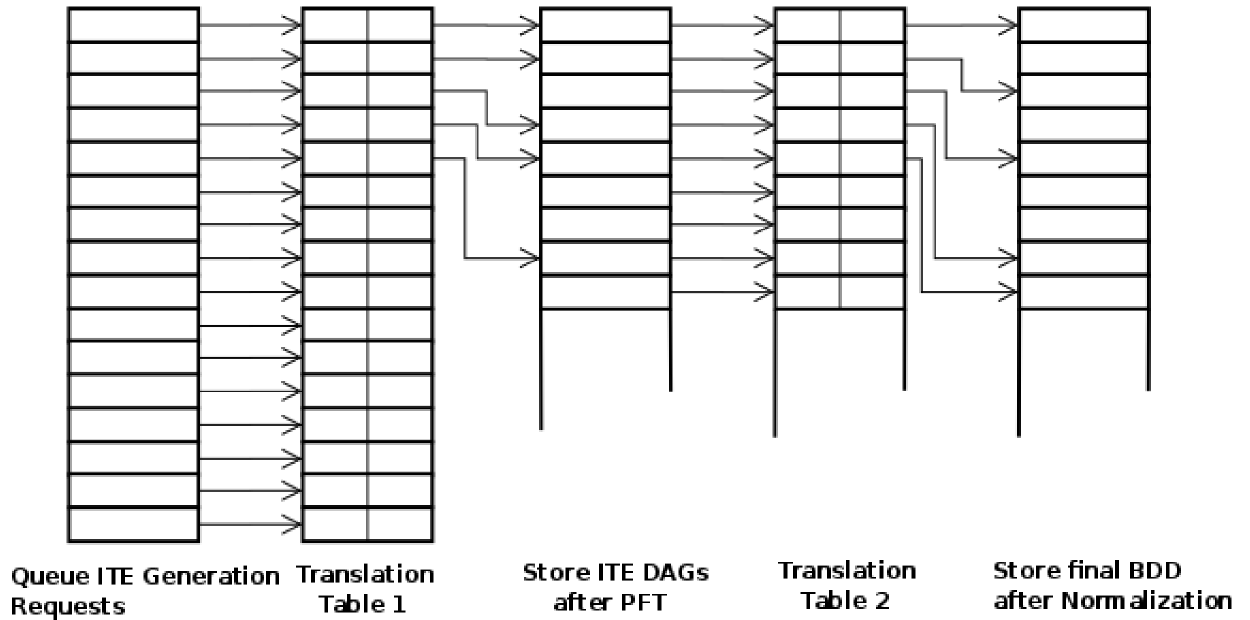



Figure 8.1: The data structures and data flow used in the algorithm for GA fitness calculation

## 8.3 Results

The results from this technique compared against our first GA are shown in Figure 8.2. This plot shows that number of nodes in the final ITE DAG is *lower* than our first GA technique. In most cases this technique produces a savings of 1 or 2 nodes in the final DAG compared to the first GA technique.

This admittedly modest reduction is none-the-less surprising because the GA normalization will produce the same result if the same variable ordering is used – clearly, the same orderings were not used. It would appear that applying PFT beforehand somehow made the GA search more effective, presumably by generating a more effective trajectory of metrics. In other words, the PFT-processed input tended to have a better correlation between incremental improvement of the metric and movement toward the global optimum order. The plot can be seen to show a very few exceptional cases in which the non-PFT input yielded a better result.

Given the cost of performing the PFT transformations, the additional benefit probably

comes at too great a compile time cost to be worthwhile for many BitC compilations. However, it does produce a measurable improvement when time allows, about 0.115%, and it also opens the unexpected possibility that future work might find other transformations to apply before normalization so that the GA normalization search is even more effective.

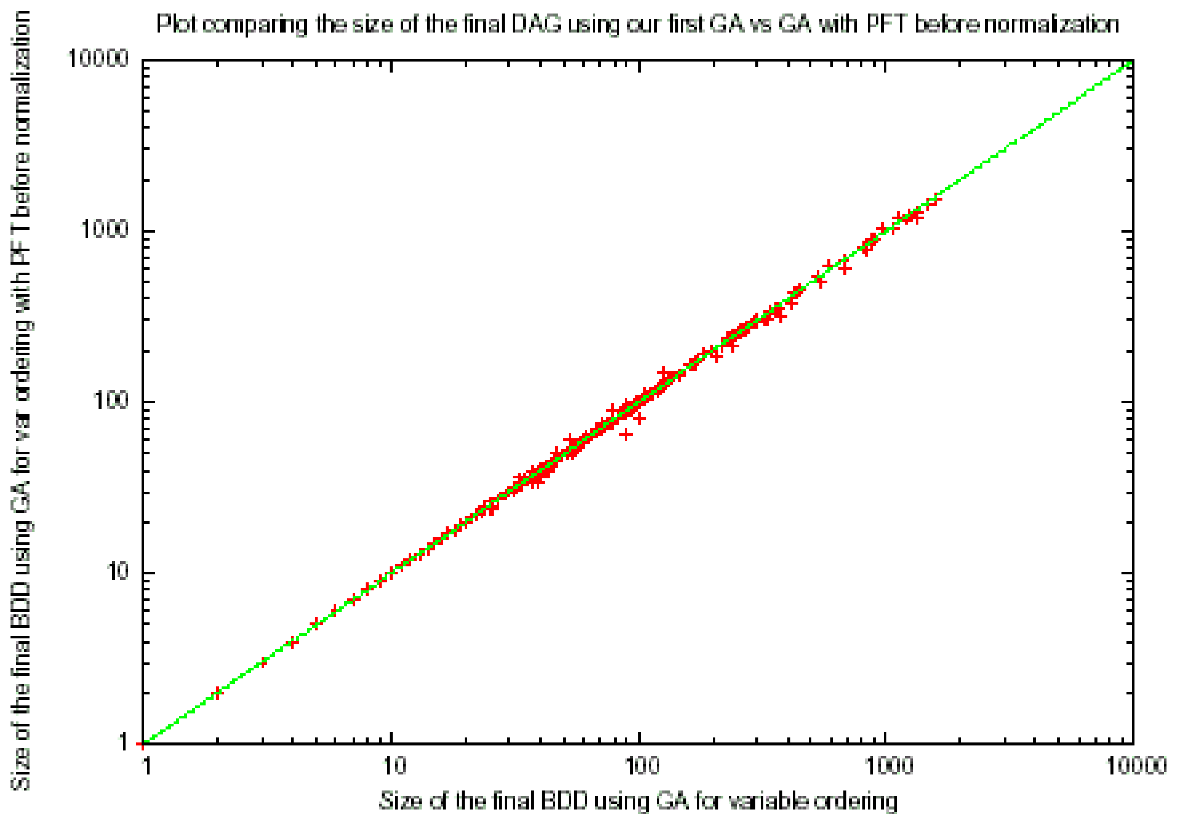



Figure 8.2: Scatter plot comparing the size of final BDD using GA for variable ordering with PFT before normalization vs. our basic GA for variable ordering

1. Initialize a set random integer strings representing variable orders as the initial individuals in the population.
2. For a given word-level program generate the bit-level ITE representation by issuing calls to the *mkite* routine.
3. Queue the requests for generating ITEs in a table
4. Evaluate the fitness of each individual in this generation by going through steps 5 through 8.
5. Subject the queued ITEs to the print form transformation algorithm. Use a translation table to translate between the queued ITEs and the newly generated non-canonical ITEs.
6. Store the non-canonical ITEs in a table.
7. Subject the non-canonical ITEs to the normalization algorithm. Use a translation table to translate between the non-canonical ITE nodes and the newly generated BDD nodes.
8. Store the number of ITEs in the final DAG in a table to serve as the fitness metric.
9. Partially sort individuals based on their fitness.
10. Apply random multiple point crossover between two randomly selected population members from the top of the sorted order and replace the variable orders which are at the bottom of the sorted order.
11. Apply random multiple point mutation on the individuals which are at the bottom of the sorted order.
12. Update the value for best fit individual
13. Repeat from Step 3 until specified number of generations are reached

Algorithm 7: Genetic Algorithm for variable ordering with ITE normalization carried out after PFT

# CHAPTER 9: CONCLUSIONS AND FUTURE WORK

Out of the three GA based heuristic techniques designed in this thesis for BDD minimization in the BitC compiler, the third GA, is found to produce the best results. This fitness metric used in this GA measures the number of BDD nodes resulting from factoring non-canonical ITE DAGs using PFT followed by normalizing the DAGs to convert them into BDD. Thus this thesis has introduced a new heuristic for minimizing the Kentucky Architecture DAGs. The techniques introduced in this thesis are not only useful for minimizing Kentucky Architecture DAGs but can also be directly applied to solving circuit minimization problems using BDDs and Karplus's ITE DAGs.

Future work on the research specific to this thesis could be directed towards:

1. The techniques developed for this thesis can be applied to standard BDD packages [BRB90] to find out how they compare against existing techniques for variable ordering like GAs, sifting, simulated annealing, scatter search etc. The techniques can also be tested on commercial BDD benchmarks like `LGsynth93` to compare against other methods.
2. The effect of the nanocontroller variable ordering algorithms on the nanocontroller GA-based register allocation routine would be an interesting direction to investigate.
3. Improving the GA runtime by terminating the search in a particular direction based on some DAG growth size criterion.

Future work on the nanocontrollers in general would be:

1. To develop a hardware for the nanocontrollers using conventional CMOS fabrication methods.

## CHAPTER 10: REFERENCES

[ADR05] Shashi Deepa Arcot, Henry G. Dietz, and Sarojini Priyadarshini Rajachidambaram. Manipulating MAXLIVE for Spill-Free Register Allocation. In *Languages and Compilers for Parallel Computing*, volume 4339 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2005.

[BRB90] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient Implementation of a BDD Package. In *Design Automation Conference*, pages 40–45, 1990.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[Bry92] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BW96] Beate Bollig and Ingo Wegener. Improving the Variable Ordering of OBDDs Is NP-Complete. *IEEE Trans. Computers*, 45(9):993–1002, 1996.

[DAG04] Henry G. Dietz, Shashi D. Arcot, and Sujana Gorantla. Much Ado about Almost Nothing: Compilation for Nanocontrollers. In *Languages and Compilers for Parallel Computing, (16th LCPC'03)*, volume 2958 of *Lecture Notes in Computer Science (LNCS)*, pages 466–480. Springer-Verlag (New York), October 2003, Revised Papers 2004.

[DC93] H. G. Dietz and W. E. Cohen. A Control-Parallel Programming Model Implemented On SIMD Hardware. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science (LNCS), pages 311–325. Springer-Verlag (New York), 1993.

[DG97] R. Drechsler and N. Gockel. Minimization of bdds by evolutionary algorithms, 1997.

[Die92] Henry G. Dietz. Common Subexpression Induction. In *ICPP (2)*, pages 174–182, 1992.

[DK93] Henry G. Dietz and G. Krishnamurthy. Meta-State Conversion. In *ICPP*, pages 47–56, 1993.

[Dre96] B. Gockel N. Drechsler, R. Becker. Genetic algorithm for Variable Ordering of OBDDs. In *IEE Proceedings - Computers and Digital Techniques*, pages 364–368, 1996.

[FMK91] Masahiro Fujita, Yusuke Matsunaga, and Taeko Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 50–54, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[HA91] M.T.O'Keefe H.G.Dietz and A.Zaafrani. An Introduction to Static Scheduling for MIMD Architectures. In *Advances in Languages and Compilers for Parallel Processing*, pages 452–444. The MIT Press, Cambridge, Massachusetts, 1991.

[HS01] William N. N. Hung and Xiaoyu Song. BDD Variable Ordering by Scatter Search. In *ICCD*, pages 368–373, 2001.

[Kar88a] Kevin Karplus. REPRESENTING BOOLEAN FUNCTIONS WITH IF-THEN-ELSE DAGs. Technical Report UCSC-CRL-88-28, 1988.

[Kar88b] Kevin Karplus. USING IF-THEN-ELSE DAGs FOR MULTI-LEVEL LOGIC MINIMIZATION. Technical Report UCSC-CRL-88-29, 1988.

[KRD07] N. Calander M. Willander K. Risveden, J.F. Ponten and B. Danielsson. The region ion sensitive field effect transistor, a novel bioelectronic nanosensor. *Biosensors and Bioelectronics*, 22(12):3105–3112, June 2007.

[LB05] Wolfgang Lenders and Christel Baier. Genetic Algorithms for the Variable Ordering Problem of Binary Decision Diagrams. In *FOGA*, pages 1–20, 2005.

[NH90] M. Nilsson and H.Tanaka. MIMD Execution by SIMD Computers. In *Journal of Information Processing, Information Processing Society of Japan, vol. 13, no. 1*, pages 58–61, 1990.

[N.I91] N.Ishiura,H.Sawada,S.Yajima. Minimization of Binary Decision Diagrams based on Exchanges of Variables. In *IEEE International Conference on Computer-Aided Design*, pages 472–475, 1991.

[Ric93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *IEEE International Conference on Computer-Aided Design*, pages 42–47, 1993.

[San94] P. Sanders. Emulating MIMD Behavior on SIMD Machines. In *International Conference on Massively Parallel Processing, Delft.* Elsevier, 1994.

VITA

Sarojini Priyadarshini Rajachidambaram was born in Thirunelveli, a town in Southern India on July 8, 1983. Majority of her schooling was in Chennai, India. She received her undergraduate degree in Electronics and Communication Engineering from the University of Madras in Summer 2004. In the Fall of 2004, she joined the graduate program in Electrical and Computer Engineering at the University of Kentucky. She has been a member of the KAOS research group in the ECE department since the early days of her graduate study. Ms. Rajachidambaram has worked as a CAD Engineering intern with Cypress Semiconductor Corporation. The author is now a full-time CAD Engineer with Cypress.