Ph.D Thesis

An Error-Localization, Validation and Optimization Tool for Embedded Code Augmentation: an Architecture Oriented Approach

Submitted to

Cochin University of Science and Technology

In partial fulfillment of the requirements for the award of the degree of

Doctor of Philosophy

by

MARIAMMA CHACKO

Under the guidance of

Dr. K. POULOSE JACOB

DEPARTMENT OF COMPUTER SCIENCE

FACULTY OF TECHNOLOGY

COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY

COCHIN, INDIA 682 022

August 2011

An Error-Localization, Validation and Optimization Tool for Embedded Code Augmentation: an Architecture Oriented Approach

Ph.D Thesis in the field of *Embedded Systems*

Author

MARIAMMA CHACKO
Department of Computer Science
Cochin University of Science and Technology
Cochin,
Kerala,
India 682 022
Email: mariamma@cusat.ac.in

Research Advisor

Dr. K. POULOSE JACOB
Professor
Department of Computer Science
Cochin University of Science and Technology
Cochin,
Kerala,
India 682 022
Email: kpj@cusat.ac.in

August 2011

"Yea, though I walk through the valley of the shadow of death, I will fear no evil: for thou art with me; thy rod and thy staff they comfort me".

Holy Bible, Psalms: 23:4

Lovingly dedicated to my parents,
Husband
and daughters

CERTIFICATE

This is to certify that this thesis entitled *"**An Error-Localization, Validation and Optimization Tool for Embedded Code Augmentation: an Architecture Oriented Approach**"* is a bona fide record of the research work carried out by **Mrs. Mariamma Chacko** under my supervision in the Department of Computer Science, Cochin University of Science and Technology. The results presented in this thesis or parts of it have not been presented for the award of any other degree(s).

Dr. K. Poulose Jacob

(Supervising Guide)
Professor
Department of Computer Science
Cochin University of Science and Technology

Cochin 22
22-08-2011

DECLARATION

I hereby declare that the work presented in this thesis entitled "***An Error-Localization, Validation and Optimization Tool for Embedded Code Augmentation: an Architecture Oriented Approach***" is based on the original research work carried out by me under the supervision of **Dr. K. Poulose Jacob**, Professor, in the Department of Computer Science, Cochin University of Science and Technology. The results presented in this thesis or parts of it have not been presented for the award of any other degree.

MARIAMMA CHACKO

Cochin 22
22-08-2011

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to my research guide Dr. K. Poulose Jacob, Professor & Head of the Department, Department of Computer Science, Cochin University of Science and Technology for his guidance, support and timely advice. I could not have completed the thesis without his encouragement and valuable suggestions.

My heartfelt debt and thanks goes to my teachers and former heads of the department of Electronics Prof. (Dr) K. G. Nair, Prof. (Dr) C. S. Sridhar and Prof. (Dr) K. G. Balakrishnan, for their advice and encouragement during the past years.

I would like to acknowledge the support provided by all my colleagues in the department of Ship Technology.

I thank the entire library, technical and administrative staff of the department of Computer Science as well as the department of Ship Technology for their co-operation and support.

My special thanks to the doctors who have treated me during my adverse health conditions and to my friend Mrs. Santha Roy for her love and companionship during these years.

Let me also remember at this moment the co-operation provided by my daughters during these years.

I would like to thank all of the people who have helped, encouraged and supported me directly or indirectly during the period of my research work.

Finally, and most of all, I would like to thank my husband James for his love, support, guidance and understanding without whom completion of this dissertation would not have been possible.

ABSTRACT

An Error-Localization, Validation and Optimization Tool for Embedded Code Augmentation: an Architecture Oriented Approach

Embedded systems are usually designed for a single or a specified set of tasks. This specificity means the system design as well as its hardware/software development can be highly optimized. Embedded software must meet the requirements such as high reliability operation on resource-constrained platforms, real time constraints and rapid development. This necessitates the adoption of static machine codes analysis tools running on a host machine for the validation and optimization of embedded system codes, which can help meet all of these goals. This could significantly augment the software quality and is still a challenging field.

This dissertation contributes to an architecture oriented code validation, error localization and optimization technique assisting the embedded system designer in software debugging, to make it more effective at early detection of software bugs that are otherwise hard to detect, using the static analysis of machine codes. The focus of this work is to develop methods that automatically localize faults as well as optimize the code and thus improve the debugging process as well as quality of the code.

Validation is done with the help of rules of inferences formulated for the target processor. The rules govern the occurrence of illegitimate/out of place instructions and code sequences for executing the computational and integrated

peripheral functions. The stipulated rules are encoded in propositional logic formulae and their compliance is tested individually in all possible execution paths of the application programs. An incorrect sequence of machine code pattern is identified using slicing techniques on the control flow graph generated from the machine code.

An algorithm to assist the compiler to eliminate the redundant bank switching codes and decide on optimum data allocation to banked memory resulting in minimum number of bank switching codes in embedded system software is proposed. A relation matrix and a state transition diagram formed for the active memory bank state transition corresponding to each bank selection instruction is used for the detection of redundant codes. Instances of code redundancy based on the stipulated rules for the target processor are identified.

This validation and optimization tool can be integrated to the system development environment. It is a novel approach independent of compiler/assembler, applicable to a wide range of processors once appropriate rules are formulated. Program states are identified mainly with machine code pattern, which drastically reduces the state space creation contributing to an improved state-of-the-art model checking. Though the technique described is general, the implementation is architecture oriented, and hence the feasibility study is conducted on PIC16F87X microcontrollers. The proposed tool will be very useful in steering novices towards correct use of difficult microcontroller features in developing embedded systems.

Contents

Abstract xiii
List of Tables xix
List of Figures xxi
Abbreviations xxiii

Chapter 1 INTRODUCTION 1-15

1.1 Background and Motivation 2
1.1.1 Reliable Software 3
1.1.2 Redundant Codes 4
1.1.3 Software Constraints 5
1.2 Embedded System Development 6
1.2.1 Real Time Systems 9
1.3 Validation and Optimization Techniques 10
1.4 Programmable System on Chip (PSoC) 12
1.5 Thesis Roadmap 13
1.6 Summary 15

Chapter 2 REVIEW OF DEBUGGING AND OPTIMIZATION TECHNOLOGIES.................... 17-69

2.1 Embedded System Constraints 18
2.2 Software Development Tools 21
2.2.1Embedded Software 23
2.2.1.1 Assemblers and Compilers 25
2.2.1.2 Dependable Software 28
2.2.1.3 Emerging Technologies 30
2.2.2 Fault Localization Techniques 30
2.2.2.1 Source Level Debugger 34
2.2.2.2 Program Slicing 36
2.2.2.3 Static Analysis Tools 39
2.2.2.4 Static Analysis of Executables 42

2.2.2.5 Static Analysis of Embedded Software 47
2.2.2.6 Dynamic Analysis 48
2.2.3 Debugging Systems and Tools 50
2.2.3.1 Testing on Host Machine 52
2.2.3.2 Simulator 53
2.2.3.3 Oscilloscopes and Logic Analyzers 53
2.2.3.4 In-Circuit Emulators 54
2.2.3.5 On-Chip Debuggers 55
2.3 Hardware and Software Integration 56
2.4 Control Flow Checking 58
2.5 Optimization 60
2.5.1General Optimizations 62
2.5.2 Processor Specific Optimizations 64
2.5.3 Interprocedural Optimizations 65
2.5.4 Profile-Guided Optimizations 66
2.5.5 Optimization of Bank Switching Instructions 66
2.6 Summary 69
Chapter 3 METHODOLOGY.. **71-78**
3.1 Program Partitioning 71
3.2 Rule Formation and Codification 73
3.3 Validation and Fault Localization 74
3.4 Optimization 75
3.5 System Realization 76
3.6 The Development Support Systems 77
3.7 Summary 78
Chapter 4 CODE VALIDATION AND ERROR LOCALIZATION... **79-120**
4.1Validation Technique 81

4.1.1Background 82
4.1.2 Applicability in RISC Architectures 87
4.1.3 Control Flow Graph Construction 87
4.1.4 Codification of Rules 90
4.1.5 Analysis Technique 92
4.2 Tool Chain 96
4.3 Feasibility Study on PIC16F87X MCU 97
4.4. Code Validation and Error Detection 99
4.4.1Fault Localization 99
4.4.2 Fault Diagnosis 109
4.4.2.1 Discrepancy in the Opcodes or Operands 109
4.4.2.2 Illegal Opcodes 110
4.4.2.3 Missed Instructions 112
4.4.2.4 A Deadlock 116
4.4.3 Error Correction 117
4.5 Results and Discussions 118
4.6 Summary 120
Chapter 5 **CODE OPTIMIZATION** **121-150**
5.1. Motivation and Approach 123
5.2. Detection of Redundant Bank Switching Codes 126
5.2.1Relation Matrix Formulation 128
5.2.2 Realization 132
5.2.3 Tool Evaluation 136
5.3 Optimization Technique 141
5.3.1 Variable Partitioning 141
5.3.2 Optimum Memory Bank Allocation 143
5.4 Redundant I/O port Configuration 147

5.5 Redundant ADC Channel Selection 148
5.6 Software Realization 149
5.7 Summary. 150

Chapter 6 CONCLUSIONS 151-159

6.1Contributions 151
6.2 Highlights of the Work 153
6.3 Merits and Demerits 154
6.4 New Research Directions 158
6.5 Summary 158

REFERENCES 161-175

LIST OF PUBLICATIONS 177-178

INDEX 179-182

APPENDIX- A A1- A4

LIST OF TABLES

Table 4.1	A-clusters for instructions MOV C, E; ADD B & POP B	83
Table 4.2	Examples of (a) B-Cluster and & (b) C- Cluster	84
Table 4.3	A sample assembly language program used to describe the partitioning concepts and analysis of the proposed code validation technique	94
Table 4.4	(a) to (i) List of governing rules formed for the PIC16F87X microcontrollers	101
Table 4.5	Code sequence governing rule 1 of Table 4.4(c)	113
Table 4.6	Code sequence governing rule 2 of table 4.4(a)	114
Table 4.7	A delay program and its erroneous version resulting in a deadlock	117
Table 5.1	Bank switching instructions and their symbols	129
Table 5.2	Relation matrix formation with PAMB and bank switching instructions	130
Table 5.3	Results of the analysis	140
Table A.1	Evaluation of programs developed using different compilers/assemblers	A3

LIST OF FIGURES

Fig. 1.1	Possible stages in the development process for the program of a simple embedded system project.	7
Fig. 2.1	The architectural differences between (a) general purpose microprocessor system and (b) a microcontroller (single chip).	19
Fig. 2.2	Activities that involve testing, debugging, verification/ validation in a typical software development process.	24
Fig.2.3	Construction of the syntax tree from the binary code of a program.	46
Fig.2.4	A typical optimization sequence in an advanced compiler.	62
Fig. 3.1	A sample design developed for simulation using PROTEUS VSM for a traffic signaling application based on PIC16F877 microcontroller.	77
Fig. 4.1	CFG abstraction details for the sample program given in Table 4.3. (a) shows the program graph, (b) shows the formation of subprograms **1**, **2** and **3** by eliminating the incoming arcs of the merge nodes 6 and 8 whereas (c) shows the CFG.	94
Fig. 4.2	The tool chain used for the proposed validation technique of embedded system machine codes.	96
Fig. 4.3	Screen shots for the analysis and reporting of violation of sleep mode operation of ADC if any. (a) the rule is validated when the antecedent (0x0063immediately follows0x151F) of the formula is found true at addresses 24h and 25h preceded by the consequent (0x179F and 0x171F) satisfied at addresses 1Bh and 1Ch. (b) reports the violation of the rule as the antecedent is true at locations 4Bh and 4Ch without satisfying its consequent.	108

Fig. 4.4	Screenshots for the results of the analysis for TRISB register configuring for programs developed in high level languages. (a) in the erroneous program a warning is generated of the use of portB as output port without corresponding trisb setting at location 3Fh. (b)in the correct program the rule is validated as the antecedent and its consequent are satisfied at locations 3Eh and 3Bh respectively.	115
Fig. 4.5	The directed graph representations for the delay routines given in Table 4.7 where each node numbered in bold represents an instruction and arrows represent the control flow between instructions. (a) shows the digraph for the correct version and (b) shows the same for the incorrect version.	117
Fig. 5.1	State transition diagram showing the bank switching scheme.	130
Fig. 5.2	Flowchart explains the identification and pruning of redundant MBSWC in the machine code sequence of a program.	133
Fig. 5.3	CFG of the sample program for the analysis.	137
Fig. 5.4	Screen shot of the developed MC_CODE ANALYZER v1.02 for the sample program.	138
Fig 5.5	Screen shot of the developed MC_CODE ANALYZER v3.00 for the sample program.	138
Fig. 5.6	CFG of the sample program with the worst case data allocation scheme.	145
Fig 5.7	The number of redundant bank switching instructions reported in the 64 data allocation schemes of the program.	147
Fig. 5.8	Various steps realized in software for the code optimization.	149

ABBREVIATIONS

ADC	:	Analog to Digital Converter
AMB	:	Active Memory Bank
ASIC	:	Application Specific Integrated Circuit
BDM	:	Background Debug Mode
CASE	:	Computer Aided Software Engineering
CFC	:	Control Flow Checking
CFE	:	Control Flow Errors
CFG	:	Control Flow Graph
CISC	:	Complex Instruction Set Computer
CPU	:	Central Processing Unit
DSP	:	Digital Signal Processor
EEPROM	:	Electrically Erasable Programmable Read Only Memory
FPGA	:	Field Programmable Gate Array
ICE	:	In-Circuit Emulators
IDE	:	Integrated Development Environment
IPO	:	Interprocedural Optimization
JTAG	:	Joint Test Action Group
MBSD	:	Model-Based Software Debugging
MCU	:	Microcontroller Unit
NVP	:	N-version programming
OCD	:	On-Chip Debuggers
OCG	:	Omniscient Code Generation
PAMB	:	Previously Activated Memory Bank
PDG	:	Program Dependence Graph
PELAS	:	Program Error–Locating Assistant System
PGO	:	Profile-Guided Optimizations

PSoC	:	Programmable System on Chip
RAM	:	Random Access Memory
RB	:	Recovery Blocks
RISC	:	Reduced Instruction Set Computers
ROM	:	Read Only Memory
RTOS	:	Real Time Operating System
SDG	:	System Dependence Graph
SFR	:	Special Function Register
SIHFD	:	Software Implemented Hardware Fault Detection
SIMD	:	Singe Instruction Multiple Data
SIS	:	Signatured Instruction Streams
SoC	:	System on Chip
STAD	:	System for Testing and Debugging
VHDL	:	VHSIC Hardware Description Language
VHSIC	:	Very High Speed Integrated Circuit
VLIW	:	Very Large Instruction Word
VSA	:	Value-Set Analysis
WET	:	Whole Execution Trace

1

INTRODUCTION

1.1 Background and Motivation 2
- Reliable Software
- Redundant Codes
- Software Constraints

1.2 Embedded System Development 6
- Real Time Systems

1.3 Validation and Optimization Techniques....................... 10
1.4 Programmable System on Chip (PSoC) 12
1.5 Thesis Roadmap... 13
1.6 Summary... 15

These days embedded systems are everywhere, appearing in places like the home, office, industry, transport, communication, automobile, robotics and in safety-critical applications such as military, medical and nuclear systems where human lives are at stake [1, 2, 3]. An embedded system can be defined as: *A system whose principal function is not computational, but has embedded software and computer hardware, which makes it a system dedicated for an application(s) or specific part of an application or product or part of a larger system* [4, 5]. It is often a complex mix of external stimuli and system responses, controlled by one or more processors and dedicated hardware [6].

Testing and debugging of embedded software remains a black art, with only ad hoc methods and techniques available. Tool availability dictates the quality of a testing process. The implications of software failure are much more severe in

embedded systems than in desktop systems [6]. The lockout of a PC for sometime may result in loss of certain files or results of some application program, whereas suspending a time critical task controlled by an embedded system could be disastrous. Embedded systems are dedicated to specific tasks which means that design engineers can optimize it for high performance and reliability because the range of tasks the device must perform is well bounded. Due to strict timing constraints owing to real time concerns, the code optimization problem is more complex than for general purpose systems. It is desirable to have automated debugging, code validation and optimization methods which utilize the vast power of host machines available today to generate efficient machine codes. An efficient compiler can provide compact code, without having to learn the intricacies of the device architecture. This makes these devices more accessible to engineers with limited programming experience who are increasingly using MCUs in their product designs [7, 8].

1.1 Background and Motivation

Most of today's technological application utilizes embedded processors as a part of their infrastructure. It is common to select a processor based on its performance and to rely on the compiler to deliver this performance. This is particularly true of high-performance RISC (Reduced Instruction Set computer) based devices. Often performance is found to be hindered by the constraints of available debugging technology [7]. Developing programs for these systems in assembly language will take more coding time, as it is less flexible, than in a higher-level language. But developers prefer assembly language modules for critical real time applications requiring stringent timing and code size. The reliability and short time-to-market requirements of embedded systems are much better met by using high level language compilers. Even though code optimization

is integrated with some of the compilers, they cannot eliminate code redundancy in many cases. Typically, a developer would guess what the problem is and try to gain visibility on the suspect variables or code segments by adding debugging statements, assertions, and breakpoints into the program. This trial and error process can be time consuming for long running programs. Moreover, a developer's intuitions may not necessarily be dependable especially if the errors are caused by his own misconceptions in the first place. [9].

1.1.1 Reliable Software

The development of error-free software for complex real time systems is an achievable goal within the reach of current software development technology. There are various approaches for developing highly dependable software through software fault tolerance techniques that uses diversity as the main ingredient [10, 11]. Static bug detection methods attempt to analyze a program for possible bugs without running it. Static tools can verify that a program is correct for all inputs, whereas dynamic tools can only find errors triggered by input test cases [9]. The notion of static program slicing was first proposed by Mark Weiser as a debugging aid [12]. In-lining of assembly code in high level language is a characteristic for embedded system software development to enable direct access to the device's hardware. Static analysis on machine code rather than source code eliminates the requirement of knowledge of the semantics of high level language. Several techniques have proposed to obtain information from executables by means of static analysis [13, 14, 15]. In the existing static bug detection methods, program verification is indecisive in general, and has only been applied successfully to small programs. Furthermore, static tools often require manual specification. Though dynamic program slicing is useful in debugging programs, the size of dynamic-dependence graphs can be very large and thus it is not possible to keep them in

memory for realistic program runs [16]. All these techniques for developing dependable software cause software overheads to the system.

1.1.2 Redundant Codes

Most of the embedded control systems are designed around a microcontroller unit which integrates on-chip program memory for storing and executing application code, data memory (RAM), various peripherals and I/O ports. Due to their architectural features there are various possibilities of introducing redundant codes by the programmer/compiler. The integration of processor cores and memory in the same chip effects a reduction in the chip count, leading to cost effective solutions. Typical examples of optional memory modules integrated with the processor on the same chip are: Instruction Cache, Data Cache, and on-chip SRAM. Many MCUs have banked memories that cannot be addressed simultaneously. Bank switching is a technique that increases the program and data memory in microcontrollers without extending the address buses [17]. A bank-sensitive program statement requires the appropriate bank to be made active prior to its execution. Use of macros simplifies the program development by managing memory resources of the target processors [18]. But, when they are used without care there is a possibility of introducing unnecessary bank select instructions which make the program too large for the device's program memory. Advanced compilers are utilizing algorithms for optimization technique to minimize the overhead of bank switching, but do not guarantee the optimal placement of bank selection instructions [17]. Generating efficient memory access code for bank switched architectures is still a challenging research problem. I/O port direction switching too may cause redundancy.

1.1.3 Software Constraints

The increasing complexity of embedded systems and the increasing need for development standards in building safety-critical systems are driving development groups to use more systematic processes [19]. In embedded applications, the cost and the short time-to-market are the leading issues [20]. It is highly desirable to develop an easy method, which will take the burden away from the software engineer by automating the error detection, identification and location steps [21] resulting in improved quality while shortening design cycles [22]. Embedded software must meet conflicting requirements such as being developed rapidly, running on resource-constrained platforms and being highly reliable. Static program analysis can help meet all of these goals [23]. Static analysis is important as these systems are used in safety critical applications and can be hard to upgrade once deployed; it is useful to detect software bugs early [6, 24, 25]. Some of the techniques proposed to find bugs in software automatically [26, 27, 28] require sophisticated program analysis. In this context a software tool to assist programmers to develop the application programs for the embedded controllers in assembly language as well as in high level language with more efficiency would be of great use.

The present state-of-the-art technology in system development uses tools like in circuit debuggers and loaders so that the compiled code can be transferred to the system and tested in real time. The integration of a code validation and optimization tool will easily fit into such a development environment for error free and efficient program development. To the best of author’s knowledge, the related literature is limited to a method for statically guaranteeing stack safety of interrupt-driven embedded software based on context-sensitive dataflow analysis of object code [24], model checking of microcontroller assembly programs [29], static

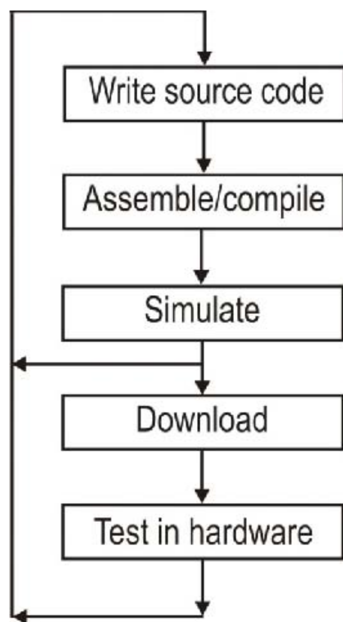
analysis on embedded assembly code to validate DSP software [30] and code optimization [17, 31, 32, 33, 34].

1.2 Embedded System Development

Embedded systems require specialized tools and methods to be efficiently designed. The various phases in a design cycle include system specification and design, hardware/software design and debug, prototype debug and system test. The specific toolset necessary depends on the nature of the project to a certain extent. At a minimum, a good cross compiler and good debugging support are needed. In many situations, facilities such as in-circuit emulators (ICE), simulators, and so on are needed. The traits that separate embedded software from applications software are [6]:

- Embedded software must run reliably without crashing for long periods of time.
- Embedded software is often used in applications in which human lives are at stake.
- Embedded systems are often so cost-sensitive that the software has little or no margin for inefficiencies of any kind.
- Embedded software must often compensate for problems with the embedded hardware.
- Real-world events are usually asynchronous and nondeterministic, making simulation tests difficult and unreliable.
- A company can be sued if their code fails.

The possible stages in the development process for the program of a simple embedded system project are similar to those of a desktop/personal computer. The programmer writes the program, called the source code, in high level/assembler


Write source code
Assemble/compile
Simulate
Download
Test in hardware

necessary to write a program in Assembler, assemble it, simulate it and then download it to a target system using a programmer. The latter must be built or bought, or designed in to the target system. With certain IDEs like MPLAB, software tools can be bought and then integrated, both from Microchip and from other suppliers. This includes alternatives to what MPLAB already offers – e.g. Assemblers or simulators, as well as tools which offer much greater development power, like C compilers or emulator drivers.

In times past, the process of downloading the program to a microcontroller always used to require the IC carrying the memory (whether a stand-alone device or memory in a microcontroller) to be placed in a programmer. This was linked to a desktop computer for the process to be carried out. As memory technology has improved, it has become increasingly easy to design the necessary programming circuitry into the target system. This means that many microcontrollers can now be programmed in situ, i.e. within the target system. Most of the modern microcontrollers are equipped with on-chip program memory using Flash technology.

If a systematic approach to test and realization is followed the first step is to ensure the correct power supply, proper running of oscillator, correct status of the Reset pin and a properly downloaded program. Once these fundamental conditions have been satisfied, a further set applies if the system is to run continuously and achieve a moderate level of functionality. These include plausible circuit and program designs, correct hardware assembly and all the peripherals being configured appropriate to the situation. As the conditions indicated are met, the system should progress to a stage of optimization. Now it shows a good level of functionality, although still imperfect in some areas. From here it is likely that further tests must be accompanied by ongoing incremental design development,

which may lie in either hardware or the program. Finally, one expects to see a system functioning to the full anticipated level of performance [35].

In this procedure our work contributes to the early validation and optimization of embedded software by conducting a static analysis on the machine code. This takes care of any error in the instruction sequence including the testing of configuration of peripherals to see that it is appropriate to the situation resulting in some sort of validation and optimization as well as the elimination of redundant instructions.

1.2.1 Real Time Systems

A system is said to be *real-time* if the total correctness of an operation depends not only upon its logical correctness, but also upon the time in which it is performed. Many embedded system must meet *real-time constraints.* A real-time system must react to stimuli from the controlled object (or the operator) within the time interval dictated by the environment. A hard real-time system guarantees that critical tasks complete on time. The application may be considered to have failed if it does not complete its function within the allotted time span. Examples of hard real-time systems include components of pacemakers, anti-lock brakes and aircraft control systems. In firm real-time systems infrequent deadline misses are tolerable, but may degrade the system's quality of service. The usefulness of a result is zero after its deadline. In soft real-time systems violation of constraints results in degraded quality, but the system can continue to operate. If the task should take, for example, 4.5ms but takes, on average, 6.3ms, then perhaps the inkjet printer will print two pages per minute instead of the design goal of three pages per minute [6, 36].

1.3 Validation and Optimization Techniques

The most crucial step in embedded system design is the integration of hardware and software [7]. Software validation involves many activities that take place throughout the lifecycle of software development. A substantial portion of the validation process is software testing, which is the development of test procedures and the generation and execution of test cases. Validation confirms that the architecture is correct and the system is performing optimally. Target level testing occurs extremely late in the development lifecycle and only a small window is allocated for hardware/software integration testing [37]. The most difficult errors to reveal and locate are found extremely late in the testing process, making them even more costly to repair.

System integration requires special tools to manage the complexity: tools that (mostly) reside on the development platform but that allow the programmer to debug a program running on the target system [6]. At a minimum these tools must:

- Include a debug kernel for controlling the processor during code development
- Support a convenient means to replace the code image on the target
- Provide non-intrusive, real-time monitoring of execution on the target.

The process of integrating embedded software and hardware is an exercise in debugging [6]. The integration phase really has three dimensions to it: Hardware, software, and real-time. The hardware can operate as designed, the software can run as written and debugged, but the product as a whole can still fail because of real-time issues. Emulators are the premier tools for HW/SW integration. An emulator's close coupling of run control, memory substitution, and trace facilities generates a synergism that significantly increases the power of each component.

Optimization is very important for embedded systems, due to limited system-on-a-chip memory sizes, real-time constraints of embedded applications, and the need to minimize power consumption of mobile devices [38]. The compilation process starts with source code analysis and source level optimization. Standard optimizations techniques, such as constant folding, common subexpression elimination, or jump optimization [39, 40] need only a minimum of machine-specific information. These are also performed at the *intermediate representation* (IR) level, where complex source code constructs have already been split into a simple form, such as three-address code. In case of multimedia applications mapped to VLIW processors, *loop unrolling*, where loop iterations are duplicated, resulting in larger basic blocks and thereby in a higher potential for parallelization of instructions during scheduling, is a very effective means of code optimization [38]. Function inlining is a well-known technique, in which function calls are replaced by copies of function bodies, so as to reduce the calling overhead. These optimizations come at the price of an increased code size. When the machine independent IR statements are mapped to assembly instructions, all machine-specific features, such as special-purpose registers, complex instruction patterns, and inter-instruction constraints need to be taken into account. This is what makes efficient code generation for embedded processors generally difficult.

More advanced approaches use a dedicated optimization phase for partitioning the program variables between the dual memory banks which are accessible in parallel in such a way, that potential parallelism is maximized [41, 42]. In embedded processors having partitioned memory architecture, where the memory banks cannot be accessed in parallel minimal placement of bank switching instructions results in code optimization [17, 31, 32, 33, 34]. Post pass optimizers usually work on the assembly language or machine code level which takes the executable output by an "optimizing" compiler and optimize it even further. For

embedded systems high code quality is much more important than high compilation speed.

Debugging software is an inevitable and arduous task. The responsibility of the fault diagnosing is to delve deeply into the bug and determine the root cause of the malfunction. Embedded systems provide the additional challenges of limited visibility of the system through a small number of inputs and outputs. Today's debugging methodologies for embedded systems can be inadequate for overcoming this problem with a low cost and flexible solution. The capability of automatic detection, identification and location of an extensible set of logic errors adds intelligence to the debugger [21, 43].

1.4 Programmable System on Chip (PSoC)

FPGA (Field Programmable Gate Array) and ASICs (Application Specific Integrated Circuits) are the modern revolutions in embedded-systems design since processors and associated peripherals can be integrated to a single chip [44, 45, 46]. ASICs are also the technology of the SoC (System on Chip) revolution that is still being sorted out today. Until recently, designers have been limited to the choice of microprocessor versus microcontroller. Now, at least for mass-market products, it might make sense to consider a system-on-a-chip (SOC) implementation, either using a standard part or using a semi-custom design compiled from licensed intellectual property. Today, it's common for a customer to completely design an application-specific embedded system containing multiple CPU elements and multiple peripheral devices on a single silicon die. Individual elements are designed in the form of "synthesizable" VHSIC (very high speed integrated circuit) Hardware Description Language (VHDL) or Verilog codes [44, 46]. Engineers connect these modules with custom interconnect logic, creating a chip that contains

the entire design. Unlike an ASIC, an FPGA can be reprogrammed without a high silicon development charge.

1.5 Thesis Roadmap

This Thesis deals with the constraints in the existing tools in embedded software development and some solutions. A code validation, fault localization and optimization tool and its applications in RISC (Reduced Instruction Set Computer) microcontrollers, to assist in efficient software development is described. This is achieved through the static analysis of machine codes by applying the rules and algorithms formulated. An algorithm which helps to eliminate the redundant bank switching instructions in partitioned memory architectures is also presented.

The thesis explores the various debugging and optimization technologies available for embedded systems in chapter two. The static and dynamic analysis techniques for localizing errors in a program are discussed. Static analysis of executables, various program slicing methods and their usefulness in debugging are examined. The role of simulation in providing a useful environment for software testing is considered. The scope of oscilloscopes and logic analyzers in debugging embedded software are limited due to the inaccessibility of buses of microcontrollers. In Circuit Emulators as a powerful technique for testing both hardware and software are examined. On chip debug supports like BDM (Background Debug Mode) and JTAG (Joint Test Action Group) standard interface protocols are briefed. Verification and Validation- the two important components of integrating hardware and software are explained. Fault tolerance through control flow checking for dependable embedded system development and various optimization techniques provided by advanced compilers are also discussed.

Chapter three introduces the methodology adopted by the static machine code analyzer for the architecture oriented validation, fault localization and optimization of embedded software. The concepts behind the program partitioning and formation as well as codification of rules are briefed. Analysis of machine code resulting in validation and optimization is presented. Programming in Visual Basic and development support systems are discussed.

Chapter four describes an approach towards code validation of RISC microcontrollers, at the level of machine instruction stream. Formulation and codification of rules governing the occurrence of illegal instructions and code sequences for executing the CPU/Integrated peripheral functions is explained. Development of a prototype based on PIC 16F87X microcontrollers is discussed. Retrieval of machine code from *Intel hex* file and the construction of CFG (Control Flow Graph) from the machine code array are described. Identification of all possible execution paths in the CFG leading to the analysis of the machine code by applying the rules governing the occurrence of illegal instruction sequence is discussed. The process of locating, diagnosing and reporting of errors and possible error corrections are presented. Results of the analysis are discussed.

Chapter five deals with the optimization of embedded code by the elimination of redundant bank switching instructions in application programs for microcontrollers with banked memory architecture. A state transition diagram representing the memory bank switching corresponding to each bank selection instruction is drawn and a relation matrix, for the active memory bank state transition, is derived. The algorithm developed for eliminating the redundant bank switching instructions using the relation matrix and its implementation in Visual Basic is explained. Analysis of the machine code to take care of the intraprocedural, loops and interprocedural routines of an application program is

also shown. A compiler strategy that can automatically determine the optimum data allocation among the memory banks resulting in the minimum bank switching code is presented. Elimination of redundant codes based on some of the rules stipulated in chapter four are also considered.

Chapter six is to enumerate the conclusions of this research work. The advantages and disadvantages of static analysis on machine code for the validation and optimization of embedded system code are listed. The extension of the use of these techniques to other applications is suggested. The major contributions of this research work as well as suggestions for improving the performance of the techniques are described.

1.6 Summary

This thesis proposes a static analyzer for embedded system software, which is close to a target level testing tool that is portable. Primary goal is to develop techniques that can be implemented in tools that are useful for people developing embedded software for the early validation and optimization of embedded software by conducting a static analysis on the machine code. The focus of our work is to develop methods that automatically localize faults and thus enhance the debugging process as well as reduce human interaction time without software or runtime overhead. Analysis is done on machine code rather than source code because this eliminates the requirement of knowledge of the semantics of high level language/assembly language and it is independent of the compiler; developers are free to change compilers or compiler versions.

2

REVIEW OF DEBUGGING AND OPTIMIZATION TECHNOLOGIES

2.1 Embedded System Constraints.................................... 18
2.2 Software Development Tools 21
- Embedded Software
- Fault Localization Techniques
- Debugging Systems and Tools

2.3 Hardware and Software Integration 56
2.4 Control Flow Checking 58
2.5 Optimization 60
- General Optimizations
- Processor Specific Optimizations
- Interprocedural Optimizations
- Profile-Guided Optimizations
- Optimization of Bank Switching Instructions

2.6 Summary 69

Embedded applications are among the most complex software systems being developed today. Embedded systems control many of the common devices in use today. Since the embedded system is dedicated to specific tasks [38], design engineers can optimize it, reducing the size and cost of the product, or increasing the reliability and performance. Embedded applications have traditionally been event driven rather than computation dependent; viz not used for general purpose computing applications. Consequently, software development of embedded systems has become a very challenging and critical task. The state-of-the-art is to

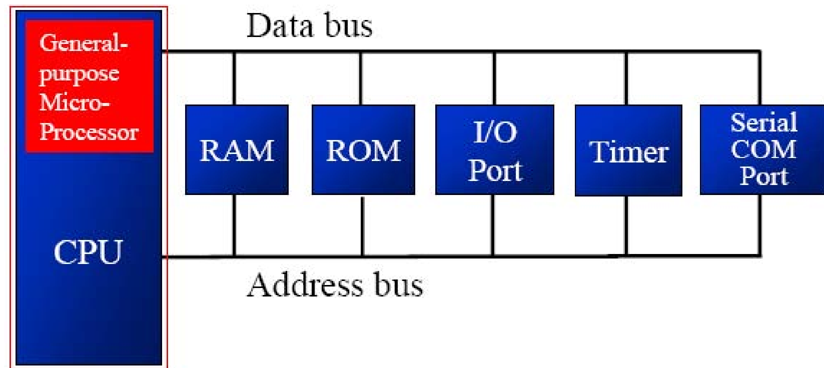
be incorporated with new and innovative technologies to ensure that the customer requirements are met quickly and cost effectively.

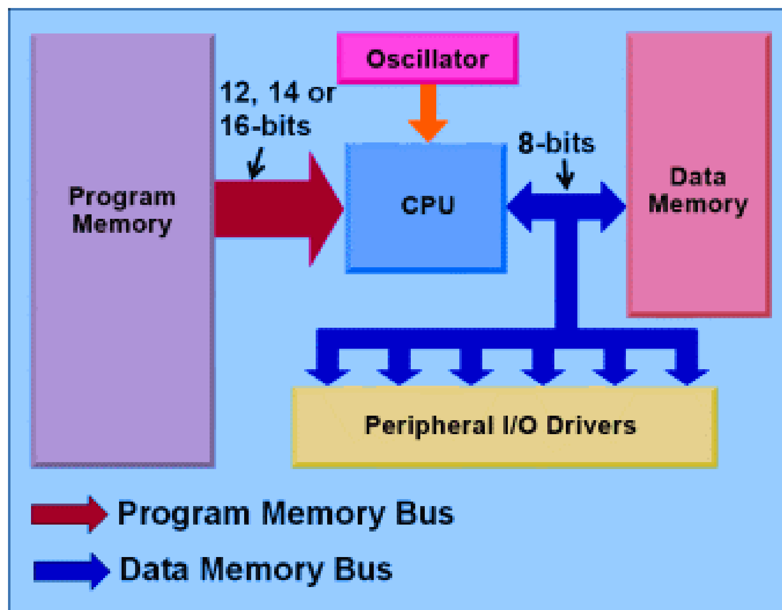
Superficially, there are great similarities between the way embedded systems are programmed and how desktop software is developed. For example, much the same programming languages - C and C++ - are the most common. The attitude toward bugs is where the approaches differ [47]. Desktop software is commonly shipped with known bugs. These may be limitations in functionality or degradations in performance over time, for example. This is considered acceptable, as failure of the software is unlikely to cause any significant harm - certainly not to put life in danger. The worst case generally results in loss of data. The attitude toward bugs in embedded software differs in three key respects:

- Many embedded applications are concerned with machinery, the malfunction of which, would be dangerous and/or expensive
- Issuing software updates for embedded applications is rarely convenient or economic
- An embedded system may be run for long periods without being reset or power-cycled. So bugs that accumulate in significance over protracted execution times are more likely to become critical.

2.1 Embedded System Constraints

General purpose microprocessors do not contain RAM, ROM or I/O ports as such. But a microcontroller has CPU, ROM, RAM, I/O ports, timer, ADC and other peripherals integrated. Fig. 2.1(a) and (b) shows the details. A typical embedded system contains a single chip microcontroller and is programmed for a pre-defined, dedicated task. The logic is often invisible as it forms part of the appliance or system.


General-purpose Micro-Processor
CPU
Data bus
RAM
ROM
I/O Port
Timer
Serial COM Port
Address bus


Oscillator
12, 14 or 16-bits
8-bits
Program Memory
CPU
Data Memory
Peripheral I/O Drivers
Program Memory Bus
Data Memory Bus

and inbuilt resources also being restricted. Although the amount of memory may not be small, it typically cannot be added on demand [7]. The current methods and tools for software testing and debugging require computing resources that are not available on the target environment. Target hardware of an embedded system normally will not support software development tools. Usually the application program is developed on a host platform and cross compilers and linkers are used to generate code and download it to the target processor by the help of loaders. Therefore, a large gap exists between the methods and tools used during evaluation on the host and those used on the target. Tools that run on the host provide a high level interface and give users detailed information on and control over their program execution. However, little is provided on the target. Typically, the best information obtainable is a low-level execution trace provided by an in-circuit emulator. Unfortunately, many errors are only revealed during testing in the target environment. The software must not only be correct, but must also interface properly with the devices it is controlling.

Embedded software is often constrained by

- Real-time constraints
- Embedded target environments
- Distributed hardware architectures
- Device control dependencies

Each of these properties of embedded software severely restricts execution visibility and control, which consequently restricts the testing and debugging process [37]. The current ability of designers to determine the reliability of embedded system depends on the techniques used in the specification, design and implementation of embedded software [2].

Nowadays, resorting to assembly language is rarely a convenient option. The complexity of the software, short development time and reliability requirements force the use of high level language for program development, while the critical modules are developed in assembly language. A thorough understanding of the efficient use of high level language and the effects and limitations of optimization are crucial. Otherwise, memory demand and real-time overheads do build up and would become apparent only late in a project, when a redesign of the software is not an option [7].

Consumer applications are characterized by tight time-to-market constraints and extreme cost sensitivity. This leads to some interesting challenges in software development. Currently most of the consumer products support upgradable software. If some bug is detected after the deployment of a product, a new version of the software can be reloaded into the device. But in safety critical systems such upgrading might be impractical.

2.2 Software Development Tools

To develop even a simple project, a selection of different software tools is beneficial. These are usually bundled together into what is called an Integrated Development Environment [35]. The development of error-free software for complex real- time systems, such as highly critical systems in defense, atomic power plants, air traffic control or space craft, is an achievable goal within the reach of current embedded software development technology. The software development technique includes process definition, state-of-the-art software engineering principles, rigorous inspection of work products across the process, independent software verification, sophisticated defect cause analysis and use of specialized tools to enhance development and testing. To meet the increased need

for software of the highest quality, knowledge engineering, expert systems, and value gained from "lessons learned" can be applied [48].

As processor architecture evolves and the complexity of instructions increases, the role of the compiler in application development becomes increasingly important. The applications developed in the embedded industry are becoming progressively more intricate, placing even more emphasis on software tools. An optimal compiler can not only increase the performance of an application, but can also decrease development cost and engineering cycle time, thereby accelerating time to market (TTM) [49]. With modern simulation technology, the code can be run, together with any real-time operating system on the host computer, and link it to a graphical representation of the user interface (UI). This enables developers to interact with the software as if they were holding the device in their hand. This capability makes checking out all the subtle UI interactions a breeze [7]. Code development techniques can be automated improving quality while shortening design cycles. With the spread of desktop computers, the mid-1980s have seen the introduction of automated environments and tools that make it practical and economical to use formal system-development methods. This technology known as CASE (Computer Aided Software Engineering), lets computer professionals develop and validate system designs and specifications, automating and enhancing the manual methods of the 1970s and 1980s [50]. CASE tools, which impose a systematic approach on program writing, grow ever more popular as adjuncts to assemblers and compilers. CASE tools enforce documenting and modeling an application from the initial user requirements through design and implementation, and can test for consistency, completeness, and conformance to standards. They help in simulating, organizing, documenting, and generating specifications for the application. They provide facilities for drawing and managing system architectural diagrams, describing and defining functional and data objects,

identifying relationships between system parts, and providing annotations to aid project management [51]. A Source-level Debugger together with an optimizing compiler provides a fully integrated real-time software development environment for embedded applications that encompasses source-level debugging, window-oriented editing, automated program building, execution profiling, and project/version control.

Right choice of the software development tools enables the programmers to develop more reliable, more capable, higher performance software in less time, at a lower development cost. Tools for embedded system development include: cross compilation systems, in-circuit tools, simulators, debuggers etc. The features, benefits and tradeoffs of these tools, and how they apply to each stage of software development, are examined.

2.2.1 Embedded Software

NASA, in their report [25] defines Embedded Software as software that is designed to execute in a computational device in order to control or to perform a specific process in support of an end item. The activities that involve testing, debugging, verification and validation in a typical software development process [52] are shown in Fig. 2.2. The starting point of system software development is requirement analysis and design. This is followed by implementation which involves developing code for all units and testing them individually. The final stage is integration and software component level testing [53]. An embedded system software development environment is an integrated collection of software development tools that manage the entire embedded software development process. Some of the technical and management activities included in independent verification and validation for embedded software are: requirements analysis and tracing; peer reviews, status monitoring and reporting, walk-through, dynamic

analysis, simulations, risk analysis, code inspection, software library maintenance, audits, and independent verification and validation (IV &V) testing using software analysis tools. These activities come into play during the various phases of the software development life cycle. Software tools are employed to automate many of these program analysis techniques. They are used to help identify actual or potential errors in the developed code, and to reformat and consolidate information to facilitate manual analysis. Software tools present a reliable, cost-effective means to supplement manual program analysis techniques. To maximize the visibility of software development quality, coding analysis is performed in parallel with code development. Automation reduces manpower costs, lowers skill levels necessary to do the work and improves the quality of the finished product [48].

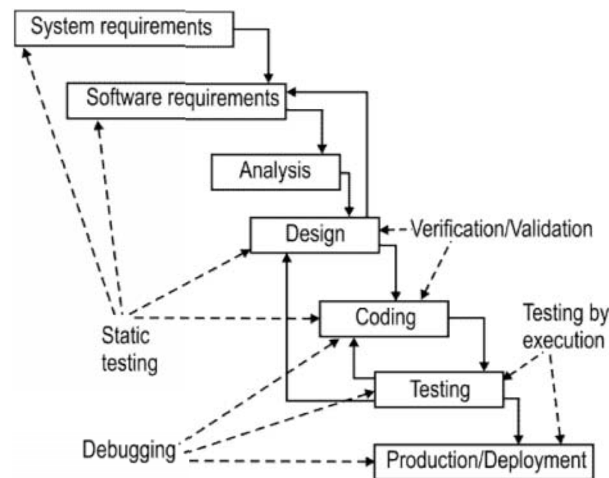


Fig. 2.2 Activities that involve testing, debugging, verification/ validation in a typical software development process.

It is advantageous to discover and eliminate software errors prior to the integration of a software system; this allows the system testing to proceed in a smooth and efficient manner. Software errors found prior to software integration is easier to repair than errors found after the product is in use [25, 48].

2.2.1.1 Assemblers and Compilers

Traditionally embedded processors mostly are programmed in assembly language due to efficiency reasons. With the advanced processor architecture and their complex applications, the need for a practical high-level language became a necessity, and several options emerged. With the increasing use of 32-bit technology, the two languages that persisted were C and Ada. The latter is prevalent in defense-oriented systems. Now, between one-quarter and one-third of embedded systems code is found to be written in C++ [7].

To implement the solution to a given programming problem, the factors relevant to a language decision probably include at least:

- Efficiency of compiled code
- Source code portability
- Program maintainability
- Typical bug rates (say, per thousand lines of code)
- The amount of time it will take to develop the solution
- The availability and cost of compilers and other development tools
- Experience of developers with specific languages or tools

The choice of programming language has far-reaching consequences, for it not only influences the number and types of development tools available, but also determines the software development schedule and future maintenance costs. Assembly language is generally best for smaller, less complicated applications-appliances like microwave ovens, consumer electronics such as video camcorders, and simple instruments like battery testers. Its advantages are that it requires less memory than high-level languages, executes faster and controls critical peripheral resources

more precisely. The use of assembly language implies time consuming programming, extensive debugging and low code portability. Re-design is particularly painful in assembler since many decisions are written into the code. Development tools for assembly language-range from simple, ROM-based debugging monitor programs to more powerful source-level debugging programs and in-circuit emulation systems with bus analyzers. More advanced assemblers offer, in order of increasing complexity and cost, cross-referenced symbol tables, file inclusion, conditional assembly, macros, and structured assembly statements [51].

The requirements of short time-to-market and reliability of embedded systems are obviously much better met by using high level language compilers instead of assembly. High-level languages are favored in larger, more complex applications such as automotive electronics, instrumentation, and industrial automation. They make programming faster, easier, and more accurate. Software development can be done faster and more reliably in high-level languages, particularly with access to a resource-aware compiler [54]. One key to low cost, high quality software is use of structured software development methods and test environments in conjunction with high level languages [48]. Drawbacks to high-level languages are their higher memory requirements and slower execution, but these are often negligible today, with the high bus speeds and power available in even mid range microcontrollers. C has emerged in recent years as the language of choice in embedded control. Its popularity is well deserved, for C combines compactness and speed; the best of assembly language; and portability, control flow constructs, data structures, and modular programming support; the best of high-level languages [51].

Development tools for high-level languages range from simple tools for inserting *PRINT* statements to more powerful source-level debugging programs and

in-circuit emulation systems with bus analyzers. Whatever the application, the entire cost of the development tools must be considered, including the host computer system, the development system (high-level-language compiler, assembler, linker, debugger monitor software, debugger hardware, and cables), and training (whether through formal classes or through reading manuals and learning the systems) [51].

It is well known that compiler generated code usually shows an overhead, both in code size and performance as compared to assembly code. Due to the need for efficient embedded systems, this overhead must be very low in order to make compilers useful in practice. In turn this requires new compiler techniques that take the specific constraints in embedded system design into account. The specialized architectures of the recent embedded processors are not yet sufficiently exploited by the existing compilers. No compiler can generate provably optimal code for arbitrary source programs. Therefore a common reference for "efficient" machine code is assembly code, manually written by expert programmers, which can be regarded as close to optimum. In almost all cases compilers for embedded processors are cross compilers i.e., they generate a code for a target processor, which is different from the compiler host machine. Consequently simulators are needed that emulate the target system by executing host instructions [38]. Application programmers want as much help as possible from the compiler in locating errors in their programs [55]. Compilers do various kinds of optimization and global analysis, but in the absence of application knowledge, it is hard to bind their runtime. The special demands on compilers, in the design of embedded systems are extensively discussed in "*Code Optimization Techniques for embedded processors-Methods, Algorithms and Tools*" written by Rainer Leupers [38].

2.2.1.2 Dependable Software

Software fault tolerance [2, 11, 56, 57, 58, 59, 60] is the ability for software to detect and recover from a fault that is happening or has already happened in either the software or hardware in the system in which the software is running in order to provide service in accordance with the specification. Software faults are most often caused by design faults. It is estimated that 60-90% of current system errors are from software faults [61]. Software faults may also occur from hardware; these faults are usually transitory in nature, and can be masked using a combination of current software and hardware fault tolerance techniques.

The key ingredient to software fault tolerance is diversity. Various forms of diversity are *design diversity, data diversity* and *functional diversity.* Design diversity is defined as the production of two or more software or systems aimed at *delivering the same service through separate designs and realizations*[10].The N-version programming (NVP) [11, 59] and recovery blocks (RB) are the two primary techniques for software fault tolerance through design diversity. The N-version programming uses different implementations, also known as *versions* or *variants,* of the software for the same purpose hopefully in a different way, and generates a consensus output by majority voting, median selection, or by some other process. Using N-version software, it is encouraged that each different version be implemented in as diverse a manner as possible, including different tool sets, different programming languages, and possibly different environments. N-version software can only be successful and successfully tolerate faults if the required design diversity is met.

Though the RB approach uses multiple versions of the software, the principle of operation is different than NVP. The output of the primary module goes the acceptance tests for verification of the correctness of the result. If it passes

through acceptance tests it is given as a final result. If it fails, then the system rolls back and starts executing an alternate module from the previously established correct intermediate point or system state, known as the *recovery point.* This sequence of operation goes on till an acceptable result is obtained, or till all the modules are exhausted.

Diversity in data space, like diversity in design space, tolerates certain classes of software failures. Diversity in data space can be achieved either by exploiting changes in data that might naturally occur with respect to time or by deliberately altering the data by re-expression which is the generation of logically-equivalent data sets. The N-copy system is similar to an N-version system but uses data diversity instead of design diversity. Each of N identical copies of a program operates on a different set of data produced by re-expression. A voter decides the system output.

Functional diversity is an artificial intelligence approach based on the premise that although the system may fail to achieve the final goal in its normal way, it may be able to achieve the goal in some other way by modifying plans and operations. Artificial intelligence techniques are used to reason its goals, to devise methods of accomplishing the task and to develop and carry out plans for achieving its goals.

Self-checking software are the extra checks, often including some amount of check-pointing and rollback recovery methods added into fault-tolerant or safety critical systems. While self-checking may not be a rigorous methodology, it has shown to be surprisingly effective.

2.2.1.3 Emerging Technologies

The increasingly wide range of processing devices has a number of possible impacts on the software designer. Obviously, suitable programming tools must be available to support this array of processors; it is preferable that the tools are consistent from one device to another. More importantly, the necessity of migrating both code and programming expertise from one device to another is becoming commonplace. This need not present major problems. By careful code design and adherence to recognized standards, porting may be quite straightforward.

In the last few years, although hardware design has become more complex, the amount of software has grown drastically, now often forming 70–80% of the total design effort. Since more software needs to be developed in a shorter time, an environment for testing is required sooner. Various solutions are available, including native code execution prototyping environments, instruction set simulation, and the use of standard, low-cost, off-the-shelf evaluation boards. In addition, low-cost host-target connection technologies are becoming common, typically using a JTAG interface [7].

2.2.2 Fault Localization Techniques

Fault localization is one of the most difficult activities in software debugging. A number of fault localization techniques have been developed to reduce the time in manually debugging a faulty program. A traditional approach to fault localization is to insert print statements in the program to cause the program to generate additional debugging information to help identifying the root cause of the observed failure. Essentially, the developer adds these statements to the program to get a glimpse of the runtime state, variable values, or to verify that the program has reached a particular program point. Another common technique is the

use of a symbolic debugger which supports additional features such as breakpoints, single stepping, and state modifying. Symbolic debuggers are included in many integrated development environments (IDE) such as Eclipse, Microsoft Visual Studio [62], Xcode and Delphi [63]. Traditional debugging techniques such as dumping memory, scattering print statements, setting breakpoints by users, and tracing program execution only provide utilities to examine a *snapshot* of program execution. Users have to use their own strategies to do fault localization.

Shapiro [64] proposed an interactive fault diagnosis algorithm, the *Divide–and–Query* algorithm, for debugging programs represented well by a computation tree. The computation tree (the target program) is recursively searched until bugs are located and fixed. Renner [65] applied this approach to locating faults in programs written in Pascal. With this method, users can only point out procedures that contain bugs; other debugging tools are needed to debug the faulty procedures. The similar result is obtained in [66].

The knowledge–based approach attempts to automate the debugging process by using techniques of artificial intelligence and knowledge engineering. Knowledge about both the classified faults and the nature of program behavior is usually required in this approach. Many prototype debugging systems have been developed based on this approach since the early 1980s [67,68]. However, knowledge about programs in the real world is complicated. These prototype systems can only handle restricted fault classes and very simple programs.

Program slicing proposed by Weiser [12, 69] is another approach to debugging. This method decomposes a program by statically analyzing the data–flow and control-flow of the program - referred to as *static program slicing*. Program dicing, proposed by Lyle and Weiser [70], attempts to collect debugging information according to the correctness of suspicious variables involved in static

program slices. *Focus* [71] is a debugging tool based on program dicing to find the likely location of a fault. Because static program slices contain many irrelevant statements that make fault localization inefficient, studying program slicing based on dynamic cases to get the exact execution path is warranted. Dynamic Program Slicing [72,73, 74] is a powerful facility for debugging and dependency analysis. Nevertheless, it has not been systematically applied to fault localization. In Agrawal's dissertation [74], he briefly alluded to the idea of combining dynamic program slices and data slices for fault localization.

Osterweil [75] tried to integrate testing, analysis, and debugging, but gave no solid conclusion about how to transform information between testing and debugging to benefit each other. Clark and Richardson [76] were the first to suggest that certain test strategies (based on the symbolic evaluation) and classified failure types could be used for debugging purposes. However, only one example is given to describe their idea, and no further research has been conducted.

STAD (System for Testing and Debugging) [77] is the first tool to successfully integrate debugging with testing. Its testing and debugging parts do not share much information except for implementation purposes (e.g., they share the results of data flow analysis). The debugging part of STAD will be invoked once a fault is detected during a testing session, and leads users to focus on the possible erroneous part of the program rather than locate the fault precisely. PELAS (Program Error–Locating Assistant System) [78] is an implementation of the debugging tool in STAD. Korel and Laski proposed an algorithm based on hypothesis–and–test cycles and knowledge obtained from STAD to localize faults interactively [79]. However, STAD and PELAS only supported a subset of Pascal, and limited program errors are considered.

Collofello and Cousins [80] proposed many heuristics to locate suspicious statement blocks after testing. A program is first partitioned into many decision–to–decision paths (DD-paths), which are straight-line codes existing between two consecutive predicates of the program. Two test data sets are obtained after testing: one detects the existence of faults and the other does not. Then, heuristics are employed to predict possible DD-paths containing bugs based on the number of times that DD-paths are involved in those two test data sets. The deficiency of their method is that only execution paths (DD-paths), a special case of dynamic program slicing, are examined. After reducing the search domain to a few statement blocks (DD-paths), no further suggestion is provided for locating bugs.

Hsin Pan and Eugene H. Spafford [81] have built a prototype debugging tool, *Spyder*, to assist users in determining statements involved in program failures, and restoring program state to a specific statement for verification. In their work, a set of heuristics is proposed to confine the search domain for bugs to a small region. The heuristics are based on dynamic program slices that are collected by varying test cases, variables, and location of variables. Although it is not guaranteed that faults can be found in the domains suggested by the proposed heuristics, a confined small region containing faults or information leading to fault discovery is provided for further analysis.

Aimed at drastic cost reduction, much research has been performed in developing automatic software debugging techniques and tools. Model-based software debugging (MBSD) techniques have been advocated as powerful debugging aid that isolates faults in complex programs [82]. By comparing the state and behavior of a program to what is anticipated by its programmer, model-based reasoning techniques separate those parts of a program that may contain a fault from those that cannot be responsible for observed symptoms. Model-based

reasoning approaches use prior knowledge of the system, such as component interconnection and statement semantics, to build a model of the correct behavior of the system. While delivering higher diagnostic accuracy, they suffer from high computation complexity.

Abreus' thesis [63] aims to contribute to advancing the state-of-the-art in automatic fault localization. He describes a spectrum-based reasoning approach [83] to fault localization in embedded software that shortens the test-diagnose-repair cycle by reducing the debugging effort. A program spectrum is an execution profile that indicates which parts of a program are active during a run. Spectrum-based fault localization entails identifying the part of the program whose activity correlates most with the detection of errors. Examples of tools that implement this approach are Tarantula [84], whose implementation focuses on the analysis of C programs, and AMPLE [85].

2.2.2.1 Source Level Debugger

Testing and debugging of software constitutes a significant amount of the development time. Debugging is an iterative, trial and error process, that is, more than one pass through each step may be needed in order to successfully remove the bug from the program. In order to remove a bug from software, the programmer first has to detect the erroneous behavior in the program and find out the type of error, locate the error in the code and modify the code piece at the error location to remedy the situation. An experienced programmer can do it in a better way [21]. Debugging time itself can account for up to 50% of the total time required for software development [86].

Source-level debugger is a tool whose interface is based on source code files rather than the binary object produced by the files. Source-level debuggers tightly couple the usually high-level implementation language with the debugging process.

They allow the programmer to debug an application program, while viewing the source code it was written in [51]. It is a powerful windowing debugger that enables program loading, execution, run control, and monitoring. It provides debugging high level as well as assembly level. It provides an exhaustive set of debugging features, to find and fix bugs: Setting and clearing Breakpoints on any executable statement line by marking the line on the screen or entering the line number, Single-Stepping through one language statement at a time, Examining and changing values of Variables, Display Stack Trace, graphical user interface, Memory Viewer, editor window, mixed language debugging [87]. The user sees all the comments in the source file and can concentrate on the debugging process itself rather than become concerned with the hexadecimal locations involved.

In an ideal world, source-level debugging would always be preferable to symbolic-level debugging. But a source-level debugger may not exist for the microcontroller, language, and platform chosen. And even if one is available, it may lie outside the developer's budget. Symbolic debugging uses the symbolic labels generated by the program rather than the absolute hexadecimal values of the labels. Primitive symbolic debuggers force the user to manually enter the symbols and values, while more advanced versions automatically read files for the symbols and values so that there is little chance of error. A file may be an assembly listing file itself, a special symbol table file generated from the assembler or from the linker, or the product of a post processing utility program. During the debug process, the user can freely use the symbolic labels to set breakpoints and examine memory locations. Program disassembly will match addresses and operands to the symbol table and show them in the assembly-language disassembly display in the proper places. The program disassembly listing therefore looks likes the generated source code listing [51].

2.2.2.2 Program Slicing

To help software engineers in debugging the program during the coding process, many new approaches have been proposed and many commercial debugging environments are available. Integrated development environments (IDEs) provide a way to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so on) without requiring compilation. Considerable work has been done in trying to automate the debugging process through program slicing [52, 86, 88].

Program slicing is a decomposition technique that extracts statements relevant to a particular computation from program. The notion of static program slicing was first proposed by Mark Wieser [12] as a debugging aid. When debugging unfamiliar programs programmers use program pieces called slices which are sets of statements related by their flow of data. Formalization of this debugging activity led to the first algorithms for program slicing [69, 89]. Different types of program slices are characterized by the type of dependency analysis and the type of statements in the slice. Weiser's static slice [12, 69] is an executable subprogram for computing variables of interest for any test case. To compute a static slice, Weiser's algorithm decomposes a program by statically analyzing the data flow and control flow of the program. An alternative approach is to compute the slice based on graph reachability in the program dependence graph (PDG), as presented by Ottenstein and Ottenstein [90, 91]. New ways of representing programs allowed Susan Horowitz [88] to extend Weiser’s method interprocedurally. Horowitz describes two graphs: the Program Dependence Graph (PDG), which represents a single procedure, and the System Dependence Graph (SDG), which connects procedures together. Once the program dependence graph is constructed, the intraprocedural slice can then be found with a very simple

algorithm given by Horowitz. He also goes into detail extending the method interprocedurally using the System Dependence graph to create function summaries. Summary edges were added to the SDG at call sites to capture certain transitive dependences [91, 92]. Static analysis identifies statements that, if executed, may affect the variable at the given location. A static slice is computed with respect to the program *P*, variable *var*, and a location *loc* in [91, 93]. Tip [94] explains how compiler optimization techniques can be used to obtain more accurate slices.

Several kinds of slices are useful in debugging. Dynamic slicing is one variation of program slicing introduced to assist in debugging [77, 95]. The problem of finding all statements that influence the value of a variable occurrence for a given test case is referred to as *Dynamic* Program Slicing. The particular test case that exercises the bug helps us focus our attention to only that "cross-section" of the program that contains the bug. When debugging using conventional debuggers, one often needs to *reexecute* the program being debugged from the start. For large programs such repeated execution from the beginning may be very cumbersome. Agrawal et al. provided a debugging tool with an execution backtracking facility with which program state can be restored at any desired earlier location without having to reexecute the entire program. Agrawal et al. in their paper [86] presents a novel way for quick localization and removal of program bugs using dynamic slicing and backtracking. Viravan [91] describes a dynamic slice computed with respect to the program *P*, a variable *var*, a location *loc*, and a test case. If program slicing was enhanced to consider weak control dependency, it would be possible to determine the transfer statements (e.g., return, break, continue, goto's) that affect the flow of control to a given location [91]. The dependences that are exercised during a program execution are identified and a precise dynamic dependence graph is constructed. Dynamic program slice of a

variable is computed by traversing the dynamic dependence graph and computing the transitive closure over data and control dependences, starting at the definition of variable at point of interest [16].

Frequently, obtaining a static slice may be sufficient to allow the user to localize a program bug. In such situations, the overhead of obtaining dynamic slices is clearly unnecessary. In other situations, especially when programs use data structures involving pointers, the sizes of static slices may approach that of the original program. In these situations, dynamic slices become extremely valuable. When debugging, a programmer normally has a test case on which the program fails. A dynamic slice, which normally contains less of the program than a static slice, is better suited to assist the programmer in locating a bug exhibited on a particular execution of the program [89].

Intra and interprocedural slicing of high level languages has greatly been studied in the literature; both static and dynamic techniques have been used to aid in the debugging, maintenance, parallelization, program integration, and dataflow testing of programs. But very little work has been published on the slicing of binary executable programs [14]. Cifuentes and Fraboulet [13] explain how to apply conventional intraprocedural static analysis to binary executables for the purposes of static analysis of machine code and assembly code, such as debugging code and determining the instructions that affect an indexed jump or an indirect call on a register. This analysis is useful in the decoding of machine instructions phase of reverse engineering tools of binary executables, such as binary translators, disassemblers, binary profilers and binary debuggers [13]. Cifuentes's paper views the main problem of disassembly as the separation of instructions from data. The instructions can jump around the data, so it is not obvious to a disassembler which is which. During disassembly, one could perform a reaching definitions analysis on

the assembly code to determine the destination of an indirect branch instruction. However, more sophistication is needed to obtain a complete result. Cifuentes suggests using a static slice of the program to determine the register contents. However, the handling of the jump instruction due to Agrawal [74] is essential to the analysis, because machine code is unstructured. Bergeron et al. in [96] suggested to use dependence graph based interprocedural slicing to analyse binaries but they did not discuss the handling of the arising problems and neither gave any experimental result. The slicing process for the control flow analysis and data dependence analysis of binary executables require special handling. A method for the interprocedural static slicing of binary executables is given in [14]. Special applications of the slicing of programs without source code are in assembly programs, legacy software and viruses: code understanding, source code recovery, bug fixing and code transformation.

2.2.2.3 Static Analysis Tools

Static bug detection methods attempt to analyze a program for possible bugs without running it. Static tools can verify that a program is correct for all inputs, whereas dynamic tools can only find errors triggered by input test cases. A static analysis of code can provide information which can hardly be discovered by traditional simulation or test techniques. A central advantage of this integrated approach is the potential for early discovery of design errors much before the costly experimentations on an actual physical implementation [97].

In certain situations, general purpose software tools that are language specific in nature can be very useful. These take the form of *static code analysis tools*. These tools look for a very specific set of known problems, some common and some rare, within the source code. All such issues detected by these tools would rarely be picked up by a compiler or interpreter, thus they are not syntax

checkers, but more semantic checkers. Some tools claim to be able to detect 300+ unique problems. Both commercial and free tools exist in various languages. These tools can be extremely useful when checking very large source trees, where it is impractical to do code walkthroughs.

One of the best known early tools for finding bugs in software is Lint [98], which used heuristics to find a variety of common errors in C programs. Compaq ESC is a static checking tool that asks users to supply invariants at procedure interfaces and other key program points [99]. Intrinsa's PREfix [100] is a tool which statically analyzes the program in C and C++ for undesirable properties like possible null pointers, leaked memory, use of freed memory, and use of an invalid resource. PREfix uses path− sensitive analysis to explore multiple execution paths in a function, with the goal of finding paths along which undesirable properties can hold. Metal is a static analysis tool which allows users to write invariants about a program in a state−machine based language [101]. An enhanced compiler checks that these invariants hold along all possible execution paths. Metal has been successful in reporting several bugs in large pieces of code, such as the Linux kernel. Hangal and Lam [9] use instrumentation to detect violations of likely runtime program invariants, and were able to find the root causes of several difficult bugs in large Java programs.

Abstract interpretation is a theory of effective abstraction and/or approximation of discrete mathematical structures as found in the semantics of programming languages, modelling program executions, hence program properties, at various levels of abstraction [102]. Abstract interpretation has been applied to static program analysis of embedded control software by P. Cusot, that is the automatic (without any human intervention), static (at compile time) determination of dynamic program properties (that always hold at runtime) involving complex

abstractions of the infinite state operational semantics [102]. This approach was successfully illustrated by the ASTR´EE static analyzer which is specialized for proving the absence of run-time errors in synchronous, time triggered, real-time, safety critical, embedded software written or automatically generated in the C programming language [103]. It was able to prove the absence of run-time errors in large industrial avionic control-command programs [102] without any hypotheses on the controlled systems (but, maybe, for ranges of variation of very few volatile input variables). This means that the software will go on functioning without any runtime error whatever the behavior of the controlled system can be, as long as the processor on which the program is running does not fail (a situation which can be handled by fault-tolerance techniques).

Several general static analysis techniques allow specification of code patterns which may be used to find bugs in programs. *Bug patterns* are code idioms that are often errors. Hovemeyer and Pugh [43] have implemented automatic detectors for a variety of bug patterns found in Java programs. They have found that the effort required to implement a bug pattern detector tends to be low, and that even extremely simple detectors find bugs in real applications. They have found that even well tested code written by experts contains a surprising number of obvious bugs and that simple automatic techniques can be effective at countering the impact of both ordinary mistakes and misunderstood language features. Automatic detectors for many bug patterns can be implemented using relatively simple static analysis techniques. In many ways, using static analysis to find occurrences of bug patterns is like an automated code inspection. They have implemented a number of automatic bug pattern detectors in a tool called FindBugs [43].

Automatic techniques to find bugs in software have been extensively studied in previous research [9, 43, 98, 100, 104]. Often, these techniques rely on formal

methods and sophisticated program analysis. While these techniques are valuable, they can be difficult to apply, and they aren't always effective in finding real bugs. A bug checker uses static analysis to find code that violates a specific correctness property, and which may cause the program to misbehave at runtime. This research is valuable, offers tremendous promise for improving software quality and many interesting and useful analysis techniques have been proposed as a result. However, these techniques have generally not found their way into widespread use.

Caveat is a static analysis tool designed to help verify safety critical software. It operates on ANSI C programs. It was developed by CEA, the French nuclear agency and is used as an operational tool by Airbus-France and EdF, the French electricity company. It is mainly based on Hoare Logic and rewriting of first order logic predicates. The main features of Caveat are property synthesis, navigation facilities, and proof of properties [104].

One important issue in bug-finding research is the need to evaluate the effectiveness of tools in finding bugs. One approach is to simply compare the output of a number of different bug finding tools; the union of the genuine bugs found by all tools compared can be considered a lower bound on the number of real bugs in the program. Rutar et. al. [105] compare a number of tools available for finding bugs in Java programs.

2.2.2.4 Static Analysis of Executables

Static analysis techniques are generally used to operate on source code. A binary executable is the machine code version of a high-level or assembly program that has been compiled (or assembled) and linked for a particular platform and operating system. Since the machine code is being executed, there is value in applying static analysis techniques to a binary executable. Several researchers [13, 96, 106, 107] have proposed algorithms for statically analyzing executables.

The first step in the static analysis must be the conversion of the machine code into assembly language instructions. However, this task itself is challenging, and has been the subject of much research. On architectures with instructions of varying size, it is difficult to locate the start of the first machine code instruction in a section consisting of both code and data. With Intel instructions [108], the opcode of the instruction can be between 1 and 3 bytes, and this can be followed by 1 to 8 bytes of immediate data and a scaling/offset byte. The instructions are not required to follow any alignment rules in memory. Because it is possible for control flow to jump around the program, it is difficult for a disassembler to find the alignment that yields the correct instructions. The disassembler must follow the path of execution in the same way as the processor, in order to skip around any data bytes in between the instructions. This may be as simple as performing a linear scan through the instructions, or as complex as interpreting the code to follow its path of execution. Also, malicious code could take advantage of the difficulties in disassembly to hide its existence. Various static analysis techniques have been developed to analyze such programs, in order to build up a control flow graph and a call graph.

One of the challenges in applying static analysis directly on the machine code of a compiled program is building up a control flow graph of a procedure, since indirect branch instructions accept the contents of a register for the destination address. Program slicing techniques can be used to reduce the assembly code to the smallest possible program to compute the value of that register, and determine the range of values in the register. A major stumbling block when developing binary-analysis tools is that it is difficult to understand memory operations because machine-language instructions use explicit memory addresses and indirect addressing. Balakrishnan and Reps [15] present several techniques that overcome this obstacle to developing binary-analysis tools. This work concerns static-analysis algorithms for analyzing x86 executables. By combining this

analysis called *value-set analysis* (VSA) with facilities provided by the IDAPro disassembler and CodeSurfer R (GrammaTech, Inc) toolkits, they have created CodeSurfer/x86, a prototype tool for browsing, inspecting, and analyzing x86 executables [15]. From an x86 executable, CodeSurfer/x86 recovers an intermediate representation that is similar to what would be created by a compiler for a program written in a high-level language. CodeSurfer/x86 provides an analyst with a powerful and flexible platform for investigating the properties and behaviors of potentially malicious code (such as COTS components, plugins, mobile code, worms, Trojans, and virus-infected code) [107].

Software applications are often distributed in binary form to prevent access to proprietary algorithms or to make tampering with licensing verification procedures more difficult. The general assumption is that understanding the structure of a program by looking at its binary representation is a hard problem that requires substantial resources and expertise. Software reverse-engineering techniques provide automated support for the analysis of binary programs. This usually involves building a control flow graph for each procedure, and looking for idioms in the code. An idiom is a sequence of instructions that, individually, do not have meaning, but when taken together form a meaningful operation. The software reverse-engineering process can be divided into two parts: *disassembly* and *decompilation*. Decompilation [13, 109] is the process of reconstructing higher-level semantic structures (and even source code) from the program's assembly-level representation that allows for comprehension and possibly modification of the program's structure. Translation of assembly code to high-level language code is of importance in the maintenance of legacy code, as well as in the areas of program understanding, porting, and recovery of code. Cifuentes and Fraboulet [110] present techniques used in the *asm2c* translator, which is a SPARC assembly to C translator. The techniques involve data and control flow analyses. The data flow

analysis eliminates machine dependencies from the assembly code and recovers high-level language expressions. The control flow analysis recovers control structure statements and simple data type recovery is also done. The techniques presented are extensions and improvements on previously developed CISC techniques. The choice of intermediate representation allows for both RISC and CISC assembly code to be supported by the analyses.

The task of the disassembly phase is the extraction of the symbolic representation of the instructions (assembly code) from the program's binary image [111]. Disassembly techniques can be categorized into two main classes: dynamic techniques and static techniques. Approaches that belong to the first category rely on monitored execution traces of an application to identify the executed instructions and recover a (partial) disassembled version of the binary. Approaches that belong to the second category analyze the binary structure statically, parsing the instruction opcodes as they are found in the binary image.

Reliable disassembler output is crucial for many security tools such as virus scanners [106] and intrusion detection systems [112]. Linn and Debray [113] present obfuscation techniques that successfully confuse current state-of-the-art disassemblers. Kruegel et al. [114] have developed and implemented a disassembler which utilizes static analysis techniques to correctly disassemble Intel x86 binaries that are obfuscated to resist static disassembly. The main contributions are general control-flow-based and statistical techniques to deal with hard-to-disassemble binaries.

Bergeron et al. [96] addressed the problem of static detection of malicious code in binary executables. Malicious codes are pieces of code that can affect secrecy, integrity, data and control flow, and functionality of a system. As malicious code can affect the data and control flow of a program, static flow analysis may

naturally be helpful as part of the detection process. Statically analyzing a program requires the construction of the syntax tree of this program, also called intermediate representation which is shown in Fig. 2.3. In order to translate an executable program into an equivalent high-level-language program, they used the disassembly tool IDA32 Pro, An advanced interactive multi-processor disassembler, which can disassemble various types of executable files (ELF, EXE, PE, etc.) for several processors and operating systems (Windows 98, Windows NT, etc.). This intermediate form is abstracted through flow-based analysis as various relevant graphs (control-flow graph, data-flow graph, call graph, critical-API graph, etc.) that capture security-oriented program behavior. The aim of flow-based analysis is to generate information about how control and data flow from one program point to another. In their system, security policies are specified using security automata, which enable to encode any safety property. Entrance into the bad state indicates that the security policy has been violated. Once the critical-API graph is computed, it is subjected to verification against the security policy to statically determine whether it exhibits malicious behaviour or not.

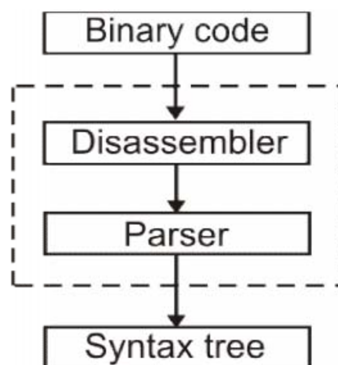


Fig.2.3 Construction of the syntax tree from the binary code of a program.

The classic virus-detection techniques look for the presence of a virus-specific sequence of instructions (called a *virus signature*) inside the program: if

the signature is found, it is highly probable that the program is infected. For example, the Chernobyl/CIH virus is detected by checking for the hexadecimal sequence [115]:

E800 0000 005B 8D4B 4251 5050
0F01 4C24 FE5B 83C3 1CFA 8B2B

This instruction sequence constitutes part of the virus body. Christodorescu and S. Jha [106] have presented an architecture for detecting malicious patterns in executables that is resilient to common obfuscation transformations. They have implemented a prototype tool called *SAFE* (*static analyzer for executable)* or for detecting malicious patterns in x86 executables, which they have successfully tried on multiple viruses. To detect malicious patterns in executables, they build an abstract representation of the malicious code (here a virus). The abstract representation is the "generalization" of the malicious code.

2.2.2.5 Static Analysis of Embedded Software

Considering the various constrains in embedded software developments, static validation methods and tools to analyze programs for possible bugs, without running them will be of great help. Most of these techniques proposed to find bugs in software automatically [26, 27, 28] are generally used to operate on source code which requires parsing of the source code that makes the analysis complex. Programmers develop microcontroller software in assembly language, high level languages like Pascal, Basic and dialects of C apart from standard ANSI C. It is characteristic for embedded system software development that a programmer inter-leave fragments of assembly code in high level language, to enable direct access to the device's hardware. Performing static analysis on the high level representation of the source code requires transforming the embedded assembly code to the high level representation. Static analysis on machine code rather than source code

eliminates this requirement and makes it independent of the compiler so that, developers are free to change compilers or compiler versions [116]

Venkitaraman and Gupta [30] describe the use of static analysis on embedded assembly code to validate DSP software for conformity with the Texas Instruments TMS320 DSP Algorithm Standard's “General Programming Rules”. ‘Hoist’ [23] generates a collection of C- functions that are ready to be linked into an abstract interpreter which helps people developing analyzers for embedded object code. J. Regehr et al. [24] had applied static analysis and transformation techniques for statically guaranteeing stack safety of interrupt-driven embedded software. They used a powerful dataflow analysis based on context-sensitive abstract interpretation of machine code. Schlich [29] has developed an approach to model check microcontroller assembly programs and implemented this approach in the model checker named **[mc]square**.

2.2.2.6 Dynamic Analysis

Some of the hardest bugs to track down are those that occur unexpectedly after a program has deployed. The use of incorrect intermediate results due to undetected bugs has been known to lead to catastrophes in mission−critical or even safety−critical situations [117]. This calls for a much deeper understanding of what happens inside a software program than the conventional visibility offered by the outputs of a program. Dynamic monitoring can also be used for malicious code detection. Sudheendra Hangal and Monica S. Lam proposed the idea of finding program anomalies through an on−line dynamic program invariant detection and checking engine (DIDUCE). It can find bugs that result from algorithmic errors, errors in inputs, and developers’ misconceptions of the APIs. It helps programmers locate bugs in unfamiliar code and, sometimes even in codes that have not been instrumented [9].

Program profiles have been analyzed to identify program characteristics that researchers have then exploited to guide the design of superior compilers and architectures. Extensive amounts of dynamic information can be collected (e.g., control flow, address and data values, data, and control dependences), and sophisticated dynamic analysis techniques can be employed to assist in improving the performance and reliability of software [16].

Because of the large amounts of dynamic information generated during a program execution, techniques for space-efficient representation and time-efficient analysis of the information are needed. To limit the memory required to store different types of profiles, lossless compression techniques for several different types of profiles [118, 119,120,121] have been developed. The control flow profile captures the complete control flow path taken during a single program run. These profiles can be analyzed for the presence of hot program paths or traces [118] that have been exploited for performing path-sensitive optimization and prediction techniques [122, 123]. Value profile captures the data or addresses values that are computed and referenced by each executed statement and are used to perform code specialization, data compression, and value encoding [124, 125]. *Address profiles* are used for identifying hot data streams that exhibit data locality, which can help in finding cache-conscious data layouts and developing data prefetching mechanisms [126]. The dependence profile captures the information about data and control dependences exercised during a program run. Data dependence represents the flow of a value from the statement that defines it to the statement that uses it as an operand. Control dependence between two statements indicates that the execution of a statement depends on the branch outcome of a predicate in another statement. *Dependence profiles* are used for computing dynamic slices [121], for studying the characteristics of performance-degrading instructions [127], and for studying instruction isomorphism [128]. More recently, program profiles are being

used as a basis for the debugging of programs. In particular, profiles generated from failed runs of faulty programs are being used to help locate the faulty code in the program [16].

A unified representation, called *whole execution traces* (WETs), and its use in assisting faulty code in a program is demonstrated by Zhang and Gupta [16]. WETs provide an ability to relate different types of profiles. For ease of analysis of profile information, WET is constructed by labeling static program representation with profile information such that relevant and related profile information can be directly accessed by analysis algorithms as they traverse the representation. An effective compression strategy has been developed to reduce the memory needed to store WET. Another reported method is a spectrum-based reasoning approach to fault localization in embedded software, where a program spectrum created gives an execution profile that indicates which parts of a program are active during a run. This aims to contribute to advancing the state-of-the-art in automatic fault localization that shortens the test-diagnose-repair cycle by reducing the debugging effort [63, 83]. Static analysis can be used to improve the efficiency of dynamic analysis techniques [106].

2.2.3 Debugging Systems and Tools

In the software life cycle, more than 50% of the total cost may be expended in the testing and debugging phases to ensure the quality of the software. Developing effective and efficient testing and debugging strategies is thus important [81,129]. Debugging during production or after deployment is very complicated [52].

Most modern microcontrollers are equipped with on-chip program memory using Flash technology. These memories have read and write access during

program execution. This capability can be utilized for downloading program during development process by loading a small firmware (communication program). Some firmwares are capable of controlling the program execution by adding breakpoints, monitoring registers as well as memory contents and port status. The disadvantage of this type of debugging is that timing constraints cannot be dealt with. This requires that some code exist in the target system before it is reprogrammed. At least two existing hardware-based methods can be used to put the initial code there in the first place: BDM and JTAG. By sending the right commands and data through a BDM port of a processor, one can push a copy of his programmer into the target's RAM space and hand over control to it. The BDM port can also be used to stimulate the I/O lines of the flash chip, thus programming it directly. It can also be implemented by hand with a few chips, a PC's printer port, and by a careful study of the processor's datasheets.

JTAG is a fundamentally different technology designed to facilitate reading and writing of a chip's I/O lines, usually while the chip's microprocessor (if it has one) is held in reset. Like BDM, however, this capability can be used to stimulate a RAM or flash chip to push a programmer application into it. And also like BDM, a JTAG interface can be built with just a few components and some persistent detective work in the target processor's manual. A JTAG bus transceiver chip, versions of which are available from several vendors, can be added to systems that lack JTAG support [7].

Approaches to dealing with limitations inherent in embedded software development can be divided into hardware solution and software solutions. The hardware solutions are attempts at gaining execution visibility and program control and include the bus monitors, ROM monitors, and in-circuit emulators. The hardware solutions have minima effectiveness for software development. They can

only gather information based on low-level machine data. The developer must then create the mapping between low-level system events and the entities defined in the program.

As on-chip functions become more complex, emulator vendors will no longer be able to see into the chip through the pins making them obsolete as well. In future architectures, the debugging capabilities provided by the chip will need to become more sophisticated. Perhaps the only possibility to view and control the execution of hardware is to gain that information from the hardware itself [37, 130].

While hardware approaches are very accurate and non-intrusive, they cannot be used in the target system, due to the added cost/size/weight and in the volume of data produced. Application-specific software instrumentation can efficiently gather information, however, it can be extremely costly to implement and maintain [131]. Modern RTOS uses inbuilt debugging features [21, 35, 132].

Most of the advanced processors have in built debugging and exception handling capabilities that can identify certain illegal opcodes, stack over/under flow etc. [133, 134, 135, 177]. The hardware scheme will increase the die foot print, power consumption and cost.

2.2.3.1 Testing on Host Machine

Embedded software is developed on a host machine which is different from the target on which the final software is to be run. Test at initial stage is done on host machine. Unlike host based application developers, embedded system developers seldom program and test on similar machines. The system integration requires special tools that mostly reside on the development platform, but that allow the programmer to debug a program running on the target system.

The host machine is used to test the hardware independent code and also to run the simulator. It gives a training which will be useful at later stage. It provides a cross compiler or a cross assembler. The worst case that can happen with a native debugger is to crash the computer. The consequence of some embedded system going out of control may be more direr [36].

2.2.3.2 Simulator

The simulation of the target program, instruction by instruction, on the host computer provides a very useful environment for software testing at almost any phase of the project. When the hardware is known but unavailable, a simulator will make rapid progress possible [132]. It permits accurate timings to be taken so that an engineer can fine-tune critical code sections early in the development cycle. A simulator uses knowledge of target processor and target system architecture on the host processor. It first does cross compilation and places this into host system RAM. The behavior of the target system processor registers is also simulated in RAM. It uses linker and locator and loads the code into RAM and functions like the code would have run at the actual target system. The execution of the code may be monitored in great detail without any intrusion at all. This facilitates 100% performance analysis and code coverage, which is not possible using other techniques. Of course, a simulator limited to the simulation of just the core CPU would be of limited utility. It does not resolve hardware dependant problems. The simulator must also address the interrupt and I/O systems [7, 36].

2.2.3.3 Oscilloscopes and Logic Analyzers

An oscilloscope is the most powerful general-purpose instrument available in the electronics world. An oscilloscope is essential in the embedded world for examining the basic conditions of power supply, oscillator and simple port activity. With expertise, it can be used for looking at more complex signals. A logic

analyzer has some of the characteristics of an oscilloscope, in that it can also display the value of an input signal against time. Since it has got many inputs it can be used to look at activity in data and/or address buses to gain a detailed view of the processor's execution [21, 35, 37]. The collection of data and its analysis is an iterative process, typically performed only during the design process [131]. Logic analyzers were most prominent in the days when systems were built up of multiple ICs and there was a need to study bus activity. Now in microcontrollers the ICs are very complex and the buses no longer accessible. They can still be very useful, however, in looking at complex digital activity, for example a parallel port, or serial data flow.

2.2.3.4 In-Circuit Emulators

In-circuit debugging is a powerful technique for testing and commissioning both program and hardware, allowing minimum invasiveness. There are many situations, however, when we need to be able to undertake the same sort of tests we were doing with the software simulator, for example testing specific sections of code or single stepping, but now with the code running in the target hardware. The solution to this need has been the in-circuit emulator (ICE). This is a device which replaces the microcontroller in the circuit, replicates its action in real-time as closely as possible. This connects with a host system across an Ethernet connection. The host computer has the power to control program execution, in much the same way as the software simulator does [7, 35, 37]. Real-time emulation means the faithful reproduction of all target signals in the same mutual relationships as those the real processor would generate [51].

The problem with ICEs, however, is that they lag behind the processor production time and become useless as the processor version changes. The behavior of a processor with on-chip instruction and data cache is invisible to an

ICE as it is to the logic analyzer, so most ICEs are designed to take advantage of the on-chip trace and breakpoint circuitry provided by the chip manufacturers. Furthermore, ICEs are usually expensive and hard to use [21, 36]. They are not usually good at replicating the action of the microcontroller in terms of the clock oscillator, and they may have power supply requirements which are less flexible than the microcontroller itself. They do not allow the genuine final operating condition of the system to be fully replicated.

2.2.3.5 On-Chip Debuggers

The technique, which is becoming increasingly common, is the debugging technique using dedicated processor pins [21]. In the latest processors this type of debugging is done via JTAG pins [136]. As the speed and complexity of processors increase, the likely cost of in-circuit emulators increases and their feasibility decreases. As a result, semiconductor manufacturers are increasingly adding debug facilities to the silicon itself. Some features of the ICE are designed into the microcontroller itself. Thus, a variety of on-chip test facilities came into being. This may vary from the provision of hardware breakpoints (address/data comparators), which should be supported by a monitor debugger, to a special "debug mode" that requires specific debugger support. Motorola used the terminology background debug mode (BDM) which is featured in Freescale 683xx (CPU32) series devices, while Microchip uses the terminology in-circuit debugger (ICD). Most of the modern processors support some dedicated pins by which a debugger program can observe some internal signals of the processor and extract debug information from interpretation of these signals. In this technique, the whole debugger software runs on a host machine and communicates with the target processor via these dedicated pins.

Most commonly, devices use a JTAG [137] connection to provide on-chip debug (OCD). A JTAG connection is quite cheap and easy to incorporate into a design [132]. This standard describes a 5-pin serial protocol for accessing and controlling the signal levels on the pins of a digital circuit, and has extensions for testing the circuitry on the chip itself. The standard was developed by the Joint Test Action Group (hence JTAG), and the architecture described by the standard is known either as 'JTAG boundary scan' or as 'IEEE 1149'. Assertion of OCD mode stops the processor and enables a debugger to read and write information to and from the machine registers and memory. To utilize OCD, an appropriate connector must be included on the target board, but this low-cost connector does not represent a significant overhead. Between the host computer and the target board, an OCD adapter is required. [7]. In the ARM SoC architecture [137] the Embedded ICE module introduces breakpoint and watchpoint registers which are accessed as additional data registers using special JTAG instructions, and a trace buffer which is similarly accessed. Some of the disadvantages of in circuit debugging are

- Some microcontroller resources are taken by the OCD function; this includes a few I/O pins, some program memory and other internal resources.
- The target microcontroller must be functioning, with its clock running.
- It is generally less powerful than a fully fledged ICE system.

2.3 Hardware and Software Integration

The most crucial step in embedded system design is the integration of hardware and software [7]. There are numerous ways to perform this integration. Doing it sooner is better than later, though it must be done smartly to avoid wasted time debugging good software on broken hardware or debugging good hardware

running broken software. Two important concepts of integrating hardware and software are *verification* and *validation (V&V)* [11, 25, 52, 138, 139, 140].

Embedded system verification refers to the tools and techniques used to verify that a system does not have hardware or software bugs. Software verification aims to execute the software and observe its behavior, while hardware verification involves making sure the hardware performs correctly in response to outside stimuli and the executing software. Software *Verification* is an iterative process aimed at proving or demonstrating that the program correctly satisfies the design specifications. Ideally, all of this verification is done before the hardware is built. The earlier in the process problems are discovered the easier and cheaper they are to correct. Verification answers the question, "Does the thing we built work?"

Embedded system validation refers to the tools and techniques used to validate that the system meets or exceeds the requirements. Validation is the process of evaluating software, at the end of the development process, to ensure that the system correctly serves the purpose for which it is intended. It answers the question, "Did we build the right thing?" Validation confirms that the architecture is correct and the system is performing optimally. Like verification, it is best to do this before the hardware is built. Tools that provide good visibility make validation easier.

The benefit of early V&V is clear: fewer bugs will be found and less rework will be performed during final system integration and test.

Pnueli et al. [141] describe CVT - a fully automatic tool for Code-Validation, i.e. verifying that the target code produced by a code-generator (equivalently, a compiler or a translator) is a correct implementation of the source specification. This approach is a viable alternative to a full formal verification of the code-generator program. Remarkably, the combination of automatic code

generation and validation improves the design flow of embedded systems in both safety and productivity by eliminating the need for hand-coding of the target code (and consequently coding-errors are less probable) and by considerably reducing unit/integration test efforts.

IAR visualSTATE is a set of highly sophisticated and easy to use development tools for designing, testing and implementing embedded applications based on state-chart diagrams. It provides advanced verification and validation utilities. One of its key features is to ensure at an early stage of design that the application behaves as expected, even before the hardware is realized [142].

2.4 Control Flow Checking

The development of a dependable computing system includes a set of methods like *fault avoidance, fault removal and fault tolerance*. In the design and implementation phase, *fault avoidance* by proved design methodologies, technologies and formal verification is important. In the test and debugging phase, *fault removal* is performed. Both methods can be referred to as *fault prevention*, preventing fault occurrence or introduction. In the operational phase, *fault tolerance* methods are applied to provide a proper system service in spite of faults. *Fault forecasting* is used to estimate the number, future incidence and consequences of faults [143].

Some type of faults such as transient faults or intermittent faults that affects the program control flow cannot be detected off-line, a runtime error detection mechanism is the only feasible mechanism to detect control flow errors (CFEs) [116, 144]. For the last three decades, many different control flow checking (CFC) mechanisms have been proposed to verify proper flow of application programs.

Control-flow checking is usually performed through signature monitoring, where the errors are detected by comparing the run-time signature with a pre-computed one. In the proposed software-only approaches [145, 146], additional software code is inserted into application programs resulting in significant code size penalties, and poor performance.

Two large classes among signature-monitoring techniques are *autonomous signature monitoring*, where the pre-computed signature is stored in a dedicated memory, and *embedded signature monitoring*, where the signature is embedded into the program [116]. The first approach [143, 147, 148] needs an additional hardware or the modification of the existing hardware. In the embedded signature monitoring approach [149, 150, 151, 152], the signature is embedded into the program under control. These approaches are based on low-level descriptions of the programs and signatures are embedded into program at compile time. The proposed techniques require special compilers to introduce the signature into program code. Michael A. Schuette and John Paul Shen [150] present an innovative approach, called signatured instruction streams (SIS), to the on-line detection of control flow errors caused by transient and intermittent faults. The program partitioning concept used to construct the control flow graph has been adopted for our work.

One of the recent approaches for developing safety critical applications is Software Implemented Hardware Fault Detection (SIHFD) [116]: a technique for the on-line detection of control flow faults in low-cost embedded systems. The proposed approach exploits the information available in the Control-Flow Graph (CFG) to embed suitable instructions of two types: *set* and *test,* in the program. During program execution these instructions check for each basic block, whether the block is reached from a legal one (according to the CFG); otherwise it possibly

indicates a control-flow error. They have performed an in depth analysis at the assembly code level in order to identify the control-flow errors.

Ragel Roshan G. and Parameswaran present a hardware-software technique to detect CFEs at the granularity of micro-instructions for the first time. MIs are instructions which control data flow, and instruction-execution sequencing, in a processor at a more fundamental level than the level of machine instructions. CFEs occur due to various low-level errors or failures. The three error models used in this paper are bit flips in instruction memory, transmission errors during communication, and errors in registers. *Bit flips in instruction memory* will be caused due to burst errors and will corrupt instructions. They will occur in on-chip or off-chip memory. *Transmission errors* may occur when bit vectors are transferred between any two levels of memory hierarchy or between different functional units. *Corruption of register values*, in particular those which determine the destination address or the condition of a branch instruction could cause an illegal branch in the control flow of an application. The technique proposed in this paper detects CFE caused by not only independent bit flips, but also bit bursts.

2.5 Optimization

Optimization is a procedure that mainly seeks to maximize performance and minimize code size. Some optimizations have a positive effect on both code size and performance whereas in other cases there is tradeoff between the two optimization goals [38]. In general, a program is processed in six main steps in a modern compiler:

- **Parser**: The conversion from high level source code to an intermediate language.

- **High-level optimization**: Optimizations on the intermediate code.
- **Code generation**: Generation of target machine code from the intermediate code.
- **Low-level optimization**: Optimizations on the machine code.
- **Assembly**: Generation of an object file that can be linked from the target machine code.
- **Linking**: Linking of all the code for a program into an executable or downloadable file.

The parser parses the source code, checking the syntax and generating error messages if syntactical errors are found in the source. If no errors are found, the parser then generates intermediate code, and compilation proceeds with the first optimization pass. The high-level optimizer transforms the code to make it better. The optimizer has a large number of transformations available that can improve the code, and will perform those that it deems relevant to the program at hand. When the high-level optimizer is done, the code generator transforms the intermediate code to the target processor instruction set. After the code generation is done, another phase of optimization takes place, where transformations are performed on the target code. There are many transformations that can only be applied on the target code. After the low-level optimizer is finished, the code is sent to an assembler and output to an object file [7]. The optimizations covered are best applied in the order presented as shown in Fig. 2.4 [49].

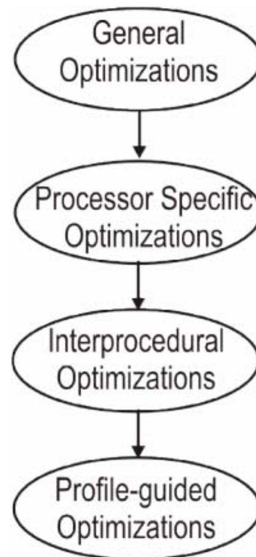


Fig.2.4 A typical optimization sequence in an advanced compiler.

2.5.1 General Optimizations

Most compilers have a series of options known as general optimizations. These consist of options that bundle together a variety of safe and easy optimization techniques. They will most often have a positive effect on the majority of programs. For example, the "optimize for speed" option does not include every possible optimization that may increase the performance of a program, it incorporates optimizations like inlining of intrinsic functions as well as simple loop optimizations. This option is the best place to start when the optimization phase begins. Function inlining is a well-known technique, in which function calls are replaced by copies of function bodies, so as to reduce the calling overhead. In case of multimedia applications mapped to VLIW processors, loop transformations are a very effective means of code optimization. A simple example is loop unrolling, where loop iterations are duplicated, resulting in larger basic

blocks and thereby in a higher potential for parallelization of instructions during scheduling [38]. If an application runs correctly in debug mode with optimizations turned off, but not after this option, it is probable that more complex optimizations will not work either. Using aggressive optimizations will often reveal errors in code that may have otherwise gone undetected.

Transformations typically applied by a low-level code-improving compiler are algebraic simplification of expressions, basic block reordering, branch chaining, common sub-expression elimination, constant folding, constant propagation, unreachable code elimination, dead store elimination, evaluation order determination, filling delay slots, induction variable removal, instruction selection, jump minimization, register allocation, strength reduction, and useless jump elimination [38, 39, 153]. Even though these techniques are normally considered machine independent, they sometimes have to be used carefully. Common sub-expression elimination, for instance reduces the number of computations to be performed, but on the other hand results in a higher number of registers required. On a processor with many functional units but only few registers, multiple recompilation of values can thus be more efficient.

New code optimization techniques dedicated to several classes of embedded processors are introduced in [38]. Specific processor architectural features are exploited in order to generate efficient code. The presented techniques are machine independent and are retargetable to a certain extent. Retargetability inherently reduces code efficiency because the fewer the assumptions a compiler makes about the target machine, the less machine specific hardware features can be exploited to generate efficient code. Machine specific code optimization should be developed with the idea of retargetability in mind.

2.5.2 Processor Specific Optimizations

When the intermediate representation (IR) statements are mapped to assembly instructions, all machine-specific features, such as special-purpose registers, complex instruction patterns, and inter-instruction constraints need to be taken into account. However, modern compilers often fail to generate highly efficient machine code as the instruction set extensions of modern processors like digital signal processors (DSPs), cannot be exploited directly in programming languages like ANSI-C. Examples for such extensions are saturated arithmetic or multimedia SIMD (Singe Instruction Multiple Data) instructions, where no analog constructs exist in programming languages.

To exploit such instruction sets within programming languages, inline-assembly was used in the past: small assembly snippets written and manually optimized by the programmer were embedded in the high-level source codes. The use of inline-assembly is disadvantageous because maintenance and portability of such source codes are poor. Nowadays, almost every compiler for DSPs offers *compiler known functions* or *intrinsics*. Using intrinsics, particular features of a processor can be exploited by the programmer [154]. The compiler maps a call to an intrinsic not to a regular function call, but to a fixed sequence of machine instructions. Using intrinsics, the resulting optimized source code is highly efficient since the compiler replaces the intrinsic by an extremely fast sequence of assembly instructions. But since intrinsics are non-standardized programming language extensions, source codes using intrinsics are no longer portable at all. Currently, only poor tool support exists to aid the programmer in replacing suitable source code fragments by efficient intrinsics.

Falk et al. [154] presents techniques for processor-specific code analysis and optimization at the source-level. In their work, the entire transformation process is

automated, but results are only presented for three very small loop kernels. In addition, just simple pattern matching techniques are used to optimize the code.

Some compilers have processor specific optimizations [49] that provide hints to the compiler about what processor the application is going to be run on. For example, if an application is going to target a processor capable of running special instructions, then the compiler needs to know it is free to generate these instructions where it feels they will be of most use. One important word of caution: If this application is run on a processor that does not support the generated instructions, the application will crash.

2.5.3 Interprocedural Optimizations

The options mentioned to this point only affect the span of one function. Interprocedural optimization or IPO works on the entire program, across procedure and file boundaries. This option allows the compiler to look at the whole program as though it were one file. In this way, actions and optimizations taken in one routine can affect those in another routine. The various steps of IPO process are as follows.

The first step consists of compiling the source files with the IPO option. The compiler creates object files containing the intermediate language (IL) used by the compiler. Upon linking, the compiler combines all of the IL information and analyzes it for optimization opportunities. Whole-program optimization is time consuming for large programs.

One of the main optimization techniques enabled with IPO is inline function expansion. Inline function expansion occurs when the compiler finds a function call that is frequently executed. Using internal heuristics, the compiler can decide to replace the function call with the actual code of the function. By minimizing

jumps through the code, this creates a higher-performing application. Other typical interprocedural optimizations are: interprocedural dead code elimination, interprocedural constant propagation and procedure reordering. The compiler's main goal is to ensure a correctly running program. If the compiler cannot predict the full impact of an optimization, it will take the safe route and desist from performing it [49].

2.5.4 Profile-Guided Optimizations

A compiler is limited to optimizing based on the data available at compile time. The actual execution of a program could behave in a way that is not intuitive from simply analyzing the available source code. Perhaps one of the most evolved optimization techniques is known as profile-guided optimizations, or PGO. This optimization allows the compiler to use data collected during program execution to aid in the optimization analysis. Knowing which areas of code are executed most frequently, the compiler can be more selective about the optimizations it performs and can also make improved decisions about how to perform them.

New research directions of optimization include low power consumption and retargetability for support of architecture exploration. Allocation techniques to statically allocate data to the scratchpad memory for energy saving were introduced in [155] and [156] whereas [157] presented a dynamic approach. These techniques are all based on the frequency of data access obtained from the execution profile and make no attempt to reduce the code size.

2.5.5 Optimization of Bank Switching Instructions

Efficient utilization of on-chip memory space is extremely important in modern embedded system applications based on microprocessor cores. Memory banking and memory paging are common techniques, which increase the size of

data and code memory without extending the address bus. Many MCUs have banked memories that cannot be addressed simultaneously. Switching between the memory banks requires at least one bank switching instruction which induces extra overhead in code size and execution time. The related literature for minimal placement of bank switching instructions is motivated by objectives, such as less runtime, low power, small code size, or a combination of these parameters.

Scholz et.al. in [17] assume the variables have already been assigned to memory banks and presents a novel optimization technique that minimizes the overhead of bank switching through cost effective placement of bank selection instruction. They formulate the placement of bank selection instructions as a discrete optimization problem that is mapped to a partitioned Boolean quadratic programming (PBQP) problem. Allocating variables to shared memory is useful to eliminate bank selection instructions. Mengting et al. in [31] presents a dynamic programming algorithm to generate the optimal assignment of variables in the shared memory to minimize bank selection instructions. Li *et al*. [32] prove that it is NP-Hard to insert the minimum number of bank selection instructions if all the variables are pre-assigned to memory banks. So they introduce a 2-approximation algorithm using a rounding method. They consider the case when there are some nodes that do not access any memory bank and design a dynamic programming method to compute the optimal insertion strategy when the Control Flow Graph (CFG) is a tree. An algorithm is presented in [33] devoted to reduce the number of page selection instructions with careful allocation of functions into pages.

The work presented in [34] aims to utilize variable partitioning techniques to minimize the size and time overhead introduced by bank switching. Current practice typically leaves it to the programmer to partition the data among different memory banks. Whether programming is done in assembly language or in a high-

level language, the programmer has to provide data manually by using assembler directives or compiler pragmatics. Compiler methods are preferable to programmer directives as they do not require programmer effort; are portable across different systems; and are likely to make better decisions, especially for large, complex programs [158]. Most of the current variable partitioning techniques aim at achieving the maximum instruction level parallelism for processors equipped with dual data memory banks accessible in parallel [159,160,161]. But these techniques will not benefit the bank switching optimization because no parallel bank accessing is allowed in this architecture. The problem of partitioning data into *scratch pad* SRAM and cache with the objective of maximizing performance has been addressed in [162]. A compiler method for automatically allocating program data among the heterogeneous memory units in embedded processors without caches resulting in reduced runtime is presented in [161]. All these works mentioned above are analyzing the source programs for optimum data partitioning.

Due to problems associated with code optimization, it is standard practice in the safety community to develop all high integrity applications with optimization disabled. This removes a potentially risky phase from the compilation process and aids traceability of the source code throughout the compilation [163].

Because software is incorporated in an increasing number of critical systems, there is a need to ensure that compilers produce machine code that correctly represents the algorithms specified at the source code level. This is a formidable task since an optimizing compiler translates a source code program to machine code while applying hundreds or thousands of compiler optimizations to even a relatively small program. The work presented in [153] describes a general approach for the automatic validation of code-improving transformations on low-level code. To ensure that a compiler produces correct machine code, compiler

developers must guarantee that all compiler optimizations are semantics preserving. The problem of proving the semantics preserving property of optimizing transformations is exacerbated for embedded systems development [164], where often either applications are developed in assembly code manually or compiler-generated assembly is modified by hand to meet speed and/or space constraints. Engelen *et al*. [164] describes the applicability of the approach to validate the optimization of embedded software using an interactive compilation system for code development. This is an important problem for embedded system developers, because the cost of malfunctioning software in embedded systems is huge.

2.6 Summary

Software debugging is an arduous task. The embedded software development constraints and the methods and tools for software testing and debugging are described. Static bug detection methods, static analysis tools developed and operation of these tools on binary executables are explained. Use of dynamic analysis for finding program anomalies are also discussed. Debugging technique using dedicated processor pins as well as integration of hardware and software are included. Techniques for the on-line detection of control flow faults in embedded systems are explored. Various optimization techniques are also explained.

3

METHODOLOGY

3.1 Program Partitioning 71
3.2 Rule Formation and Codification 73
3.3 Validation and Fault Localization 74
3.4 Optimization 75
3.5 System Realization 76
3.6 The Development Support Systems 77
3.7 Summary 78

Methodology adopted for the error-localization, validation and optimization of embedded system code is described here. This work adopts a machine level approach centered on early detection of instruction sequencing errors associated with the core as well as peripheral functioning of the target processor in an embedded system that would lead to malfunctioning at runtime. Static analysis is done with a control flow graph (CFG) constructed from the machine code, following the basic program partitioning concepts. The development environment and tools at each stage of the work are explained.

3.1 Program Partitioning

For the possible realization of this architecture oriented approach, its feasibility has been verified on systems based on PIC16F87X series of microcontrollers. This family of microcontrollers constitutes a RISC-based Harvard

architecture with instruction size of 14 bits and a data bus of 8-bit width [165, 166]. Each PIC16F87X instruction is a 14-bit word, divided into an opcode which specifies the instruction type and one or more operands which further supplement the operation of the instruction.

Static analysis techniques are generally used to operate on source code. But we need to apply them on machine code. For the implementation of the machine code analyzer first the machine code together with its address is read from an *Intel hex file*. All possible valid sequences of instructions of an application program can be represented by a directed graph, G =<V,E> called the program graph. V is the set of nodes where each node in the graph represents a machine code for a single instruction along with its address and E represents the set of edges where each edge (arc) represents valid control flow between two instructions. A program graph [14, 39, 96, 150] is constructed from the machine code using all the conditional and unconditional control transfer instructions and their destination addresses for the target processor under consideration.

Identification of the modules or subprograms in a program is done by following the basic program partitioning concepts [16, 39, 150, 167]. For the target processor the entry points and exit points in the program graph can be identified from the control transfer instructions [151] and their destination addresses, which are available from the popular Intel hex file or binary file format. All the merge nodes, which are nodes in the program graph with more than one incoming edges are identified. The CFG is abstracted from the machine code array as follows.

In order to get the correct sequencing of instructions, the program (machine code) is partitioned [150] into blocks of instructions by disconnecting from every merge node all of its incoming arcs [167]. Hence the program graph is partitioned into a collection of disconnected sub-graphs where each subgraph corresponds to a

set of instructions or subprogram. Since each subgraph is a tree, they have only one entry point (root node) and there is a unique path, and hence a unique sequence of instructions, from the entry point to each of the exit points. Now the CFG [88, 106, 163,168] can be constructed whereby each subgraph of the program graph is represented as a single node with the arcs representing valid control flow between sub-graphs that correspond to an exit point of a subgraph, represented by the source node of the arc. Each subgraph is a rooted tree with the merge node being the root node and all the arcs being directed away from the root node [150].

3.2 Rule Formation and Codification

For the analysis a number of rules are formed based on the instruction set and architectural features of the target processor. A thorough analysis of various instructions used for configuring the integrated peripherals of the target processors has been conducted. The likelihood of occurrence of typical instructions in a stream is explored and necessary rules are formed in a well-defined manner. These rules govern the validity of instruction sequence in a given program. Rules to identify the following illegal codes or code sequences have been formed.

- Opcode that does not decode to any known instruction of a processor
- Any literal byte selected or any instruction for configuring a register which sets the reserved/unimplemented bits
- Codes that may access unimplemented data memory bank locations.
- Codes that may result in improper read, write, initialization or configuration of registers associated with the core as well as peripherals.
- Discrepancy in the opcodes and operands
- A missed or redundant instruction in a possible execution path
- Code sequence that may result in a deadlock.

The target processor specific requirements in the machine code sequence are stipulated as rules of inferences and are of the form

(premises ⇒ consequent) or (antecedent ⇒ consequent)

The premises and consequences are expressed in propositional logic formulae. The premises are a sequence of machine code pattern and the consequences are the set of prerequisites or post requisites in the machine code pattern/sequence for the corresponding premises.

Machine code or code sequences against each of the rules are identified for the prototype microcontroller. Around one hundred rules have been formed for this family of microcontrollers.

3.3 Validation and Fault Localization

Using the CFG the entire program can be broken down into a fixed number of unique execution paths. The rules formed are applied in each of the execution path to locate an illegitimate/out of place code in the instruction stream. Each rule is applied independently and the instruction sequence in all possible execution paths are checked for compliance of the rule. Soundness of the rules is checked applying the principle that $f_1 \Rightarrow f_2$ is false only when f_1 is true and f_2 is false [169]; for all other conditions it is true. For validation of the code the rules formulated are applied so that the code or code sequence in each path is compared with a code or code sequence provided by each rule. Each comparison results in setting or resetting of a number of monitoring flags. At the exit point of each node in the CFG a number of conclusions are made based on which the analysis of the next node is done. At a merge node the conclusions made in the entire possible path to the merge node determines the analysis of the sequence of instructions that follows. The rule which has invalidated the code leads to the localization of error. The

necessary program has been developed in Visual Basic which reports the violation of the rules and location of errors if any. When this tool is integrated to the system development environment the debugging process can be made interactive and user friendly.

3.4 Optimization

When the machine code from the compiler/assembler is subjected to the analysis based on rules framed for validation, the result is a stream of machine code with the probable errors eliminated and the stream optimized for code efficiency. The algorithm developed to assist the programmer for eliminating redundant data memory bank selection instruction utilizes static analysis of machine codes. The algorithm rests on a relation matrix Δ [169], formed for the Active Memory Bank (AMB) state transition, corresponding to each bank switching instruction in the machine code sequence of an application program. This relation matrix also helps the machine code analyzer in identifying the registers as well as peripheral devices used in the application program.

The memory bank that was active just before the execution of a bank switching instruction is named Previously Activated Memory Bank (PAMB). A bank switching/selection instruction is said to be redundant when the execution of such an instruction switches the memory bank to an Active Memory Bank (AMB) that does not alter the PAMB. A detailed study on the various PIC families of microcontrollers has been made in this regard. The default memory bank state is assigned at start to the Previously Activated Memory Bank (PAMB) of each path of the 1st CFG node. For each memory bank switching code (MBSWC) in a valid path, the AMB state is obtained from the matrix Δ. A redundant MBSWC is located when AMB and PAMB are the same.

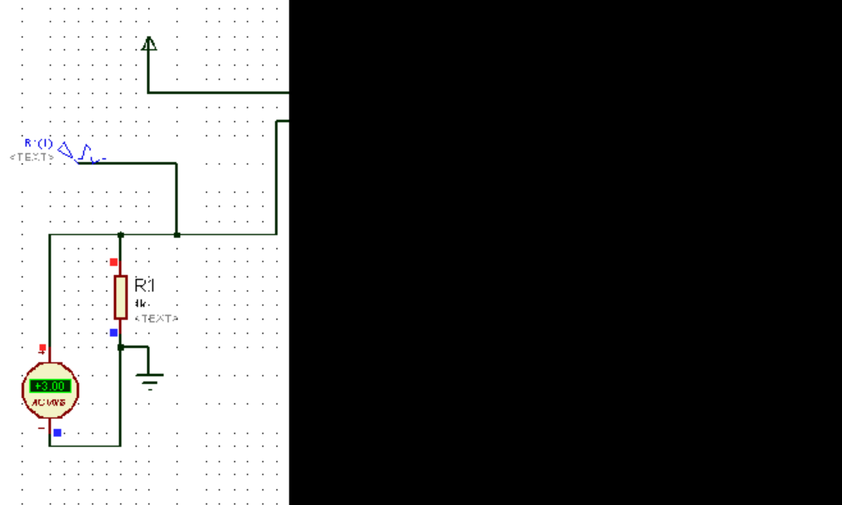
Analysis of a subprogram identifies the redundancy of the memory bank switching instructions associated with the intraprocedural routines. As a first step towards the suppression of false warnings, for all the CFG nodes, even though the first (pair of) bank switching instruction in any path is found to be redundant, they are not reported till the interprocedural analysis is over and the redundancy is confirmed. Hence the AMB associated with the merge node in a subprogram is taken as the union of AMBs of its incoming arcs. As a second step, the algorithm considers all the transparent nodes which do not contain any bank switching instructions.

For optimum data allocation to memory banks, data mapping matrices for all possible permutation of memory banks and combination of data are considered. The program with a data allocation scheme that results in the minimum number of bank switching codes is selected.

3.5 System Realization

The implementation of the validation and optimization algorithm developed is done in Visual Basic. Visual Basic is one of the most popular and friendly programming languages available today. The ability to develop object models and data base integration [170] provides the capabilities needed for this work. Program modules have been developed for:

- file reading and data extraction
- Program flow identification
- Merge nodes identification
- CFG construction
- Execution paths Identification
- Intraprocedural linear scanning


<TEXT>
R1
<TEXT>
+3.00

3.7 Summary

The work presented in this thesis utilizes a static analysis of the machine code to achieve error-localization, validation and optimization of the embedded system code. CFG construction based on the basic program partitioning concepts is explained. The application of governing rules to the possible execution paths for validation is explained. Optimization of the code with the use of a relation matrix is dealt with. The various program modules developed for the realization of validation and optimization are introduced. The tools and support systems used in this work are briefly introduced in this chapter.

4

CODE VALIDATION AND ERROR LOCALIZATION

4.1Validation Technique 81
- Background
- Applicability in RISC Architectures
- Control Flow Graph Construction
- Codification of Rules
- Analysis Technique

4.2 Tool Chain 96
4.3 Feasibility Study on PIC16F87X MCU 97
4.4. Code Validation and Error Detection 99
- Fault Localization
- Fault Diagnosis
- Error Correction

4.5 Results and Discussions 118
4.6 Summary 120

Testing and debugging of embedded software remains challenging, with only ad hoc methods and techniques available. The time required for the program development, the efficiency and code size of the developed program, all depends largely on the skill and expertise of the programmer. It is advantageous to discover and eliminate software errors in applications prior to the integration of the system; this allows the system testing to proceed in a smooth and efficient manner. Since embedded systems are typically resource constrained, addition of real time validation activities that becomes part of the final system will increase the demand

for resources. In this context it is desirable to have automated debugging and code validation methods which utilize the vast power of machines available today to reduce human debugging time and to improve the quality of software developed. Many of the debugging tools incorporate program slicing techniques [52, 86, 88] proposed by Mark Weiser [12].

A new application domain for the static analysis of machine code; a code validation technique to assist a programmer in developing error free and reliable embedded software which may reduce the development time as well as improve the quality of the software is discussed here. An incorrect sequence of machine code pattern is identified using slicing techniques on the control flow graph generated from machine code. With the help of rules of inferences formulated for the target processor, this tool detects codes and code sequences, which may cause the program to misbehave at run time. Codification of rules is done in propositional logic [169] and the compliance of the rules is checked in all possible execution paths. It can be used to debug a compiler or an assembler generated machine code on a host machine before the system realization and validation phase. With this analysis any logical errors as well as code redundancy made by the programmer for the target processor can be identified by an incorrect sequence of machine code pattern and can be reported. Unlike the reported techniques, this work neither require abstract version of each instruction in the target architecture since it does not deal with the syntax of the construction language nor require a full disassembler; only the ***hex code*** patterns or sequence of the machine code are compared for the analysis. Analysis can be done without any knowledge about the source code except the target processor. This work contributes a useful tool in steering novices towards correct use of difficult microcontroller features in developing embedded systems. During software development one has to go

through edit, compile and test routines repeatedly which can be reduced considerably by the proposed tool.

The present state-of-the-art technology in system development uses tools like in-circuit debuggers and loaders by which the compiled code can be transferred to the system and tested in real time. The integration of an automatic code validation tool will easily fit into such a development environment for error free and efficient program development.

4.1 Validation Technique

Embedded software development requires cross compilers and the code generation phase of a compiler is processor dependent. Existing compilers and assemblers can detect any lexical, syntactic or semantic errors in the source code, while they cannot detect any logical error made by the programmer. It is important that machine level code should be checked, because this is the code that actually directs the operation of the processor. The proposed approach incorporates rules of inferences based on the instruction set and architectural features of a processor to validate and optimize the compiled or assembled code of an application program. Hence any discrepancy in the instruction sequence made by the programmer in configuring the CPU and integrated peripherals functioning can be identified. This contributes to an effective method of early detection of instruction sequencing errors resulting from subtle deviations of the hardware specification that had slipped through conventional testing that would lead to malfunctioning at runtime. An advantage of this technique is that after the overhead of verification, the program executes with no run time penalty at all. The error sequence and redundancy removal can contribute to code optimization.

Rules of inferences can be formulated based on different strategies by classification of the plausible faults under various headings like discrepancy in opcodes or operands, illegal opcodes, missed/redundant instructions etc. Again clustering of instruction set can be done to form flag affecting instructions, flag checking instructions, instructions having same destination register etc. based on which rules can be formulated. Certain rules can be generalized while others depend on the architectural features of the processor. The rules hence formed are converted into machine code sequences/patterns with the help of an assembler for the target processor. Based on this code sequence, propositions are defined and the rules are coded into propositional logic formulae. Now these rules are validated by means of slicing techniques on the CFG generated from machine code of the application program. CFG is an intermediate representation of a program as a directed graph with vertices representing instructions and arcs (edges) representing transfers of control between them [163].

4.1.1 Background

The author had proposed a debugging system based on reasoning obtained by heuristics with a novel scheme of forming instruction clusters [171] which in turn have been used to form a knowledge base. A prototype based on Intel 8085 CPU has been developed. A thorough analysis of the various instruction streams [172] of the 8085/8086 family of processors [173] has been conducted. The likelihood of occurrence of typical instructions in a stream are explored and clusters formed in a well defined manner. The logic that is adopted is an offshoot of the method adopted at clustering. Basic rules have been formulated which guide the occurrence of instruction in a stream. Based on this a plausible sequence can be predicted and errors identified. An algorithm has been designed which check the appropriateness of instruction codes. The algorithm encompasses a wide range of processors, once appropriate clusters are

available for such processors. As a first level attempt, three schemes which produce A-Cluster, B-Cluster and C-Cluster are proposed.

Scheme 1: A-Cluster for a typical instruction is made up of instructions which are not likely to follow that particular instruction. For example, consider the instruction *MOV C,E*. An instruction like *POP B* is not likely to follow it, owing to the reason that both have the same destination for data. Hence *POP B* will be a constituent of the A-Cluster for *MOV C,E* which can be formulated into a rule:

A-Cluster rule: A data transfer instruction to immediately follow another, with the same destination is illegal or in general an instruction is not to be immediately followed by its A-Cluster element. Table 4.1 shows samples of A-Clusters for typical instructions. On an identical approach, rules can be framed in forming A-Clusters for the complete instruction set of typical processors.

Table 4.1 A-clusters for instructions MOV C, E; ADD B & POP B

I	***MOV C, E***	***ADD B***	***POP B***
	MOV C, r2	MOV A, r2	MOV B, r2
	MOV C, D8	MOV A,M	MOV C, r2
	LXI B,D16	MVI A,D8	MOV B, M
	POP B	LDA Adr	MOV C, M
[A-Cluster] I		LDAX rp	MVI B, D8
		SUB B	MVI C, D8
		POP PSW	LXI B, D16
		IN D8	PUSH B
		RIM	POP B

Scheme 2: B-Clusters are formed by instructions that have a bearing on status flags. This is done with a view to checking the sequence by identifying a possible flag checking instruction to follow a B-Cluster instruction. In the typical case of instructions that affect flag in different ways subdivisions BC1, BC2 and

BC3 can be formed to indicate each category. For example, a BC1 instruction will affect only one flag, BC2 might affect all flags, BC3 not affecting any flag, etc. Table 4.2(a) shows a sample of B-Cluster which depicts the cluster of instructions affecting only the carry.

Scheme 3: clusters are formed comprising flag checking instructions. These constitute C-Clusters. These are mostly conditional transfers which examine the status of one of the processor flags to determine whether the normal sequential flow is to be altered. Table 4.2(b) shows the cluster for conditional jump instructions.

Table 4.2 Examples of (a) B-Cluster and & (b) C- Cluster

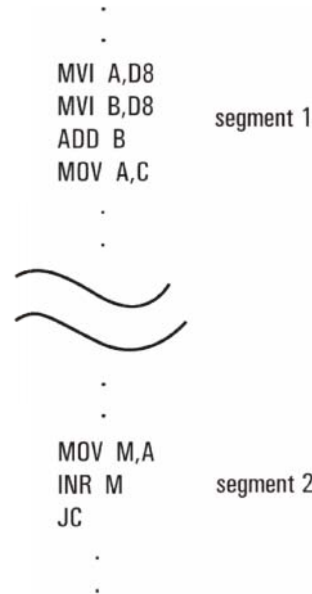
(a)	(b)
DAD rp	JZ
RLC	JNZ
RRC	JC
RAL	JNC
RAR	JPO
CMC	JPE
STC	JP
	JM

A rule is formulated to relate instructions in B and C Clusters as:

B and C-cluster rule: Any C-Cluster instruction is necessarily to be preceded by B-Cluster instruction of the appropriate subdivision. An exhaustive listing of all the rules that can be formed is not intended, only the basement is provided which is well indicative of the approach.

Fault location is attempted by making use of instruction clusters that are formed as per scheme 1. Cluster information is processed to identify an illegitimate

code or an out of place instruction. To illustrate this, the following program segments are considered.



In the first segment, the fourth instruction is out of place according to A-Cluster rule. Since the third instruction belongs to a B-Cluster (a flag affecting instruction), there has to be an instruction that belongs to a C-Cluster that checks a flag affected by the preceding B-Cluster instruction, in the stream. But if no such instruction appears before the occurrence of another instruction in the B-Cluster, then it points to a fault which may be due to a missed instruction or error in the operand/opcode field, the possibility of instruction being illegal according to Cluster scheme 1 having been eliminated earlier. This amounts to diagnosis. In segment 2 the execution of the second instruction which belongs to B-Cluster affects all condition flags except carry. In the third instruction again a fault is located according to second rule formulated. There could be an instruction belonging to BC1 or BC2-Cluster before an INR M instruction, the possibility of the third instruction being illegal having been eliminated earlier. Adequate rules

have been framed based on such observations and a knowledge base created. Inferences are drawn which ultimately converge to the fault. In the program segment 1, the opcode of the fourth instruction could be ADD instead of MOV. Similarly the opcode of the third instruction in segment 2 could be JZ instead of JC. Again in segment 1 the error could be in the operand field of the fourth instruction. If the operand field is changed to C, A from A, C the error may be corrected. The concept described can be extended and tailored to diagnose such errors. The necessary algorithm has been formulated. A sample of analysis for the two program segments is given here.

```
IF A-Cluster rule  Successful for I
THEN IF I ∈ [B-Cluster]
        THEN IF any I1 ∈ [C-Cluster] follows that checks flag affected by I
                THEN IF any I2 ∈ [B-Cluster] before I1
                        THEN next I
                         ELSE fault or missing instruction or error in the operand/opcode field
                 ELSE out of place or missing instruction or error in the operand/opcode field
        ELSE out of place instruction or error in the operand field
ELSE next I
```

```
IF B & C Cluster rule fails for I
THEN IF any I1 ∈ [B-Cluster]  precedes that affects flag checked by I
         THEN next I
         ELSE report mismatch [C-Cluster]I
ELSE next I
```

Sequential information is called for, when used as a debugging tool. Any actions taken and the results of tests are stored for use as input to the diagnostic process at the next level [174]. This historical background helps in placing the missed instruction or correcting such errors. Steps at forming clusters again can be standard, but the number of clusters and their configuration vary from processor to

processor. The ultimate objective is to foretell the likelihood of errors and their sources, so that remedial action can be taken in advance.

4.1.2 Applicability in RISC Architectures

This method is particularly significant in RISC machines [38, 137, 163] as the reduced number of powerful opcodes to be identified in the instruction stream simplifies the complexity of the algorithm to be developed. The existence of some relations among the various instructions occurring in a stream forms the basis of this approach. RISC processors evolved from CISC architectures due to the observation that complex instructions are used only rarely in practical applications [38]. The majority of new processors uses RISC technology and typically has limited addressing modes and a small number of instructions. Another characteristic of RISCs is a load-store architecture with a large number of general purpose registers in order to reduce the number of memory accesses in a machine program. In fact a careful assignment of program variables to registers is the most important optimization of a compiler for RISC. Still for a fixed application, the code size for a RISC generally exceeds the code size for a CISC processor. The largest difference with RISC devices over CISC ones lies in the numbers of instructions to implement source code functionality [163]. *The small size and simplicity of RISC instruction sets is well suited to the style of error-localization, validation and optimization presented in this thesis.*

4.1.3 Control Flow Graph Construction

Most automatic analysis tools including ours use an intermediate representation, such as the control flow graph (CFG) or the program dependence graph (PDG) [88], that is not sensitive to superfluous changes in control flow [106]. Control flow information, being a collection of processing elements linked

by (conditional) transfers of control, is represented naturally as a directed graph with vertices representing computations and arcs representing transfers of control between them. Such a graph is called a control flow graph [168], which can be constructed for programs written in any imperative programming language [163].

Given a machine code program, a forward control flow graph is generated as follows. Identification of the modules in a program is done by following the basic program partitioning Concepts [16, 39, 150, 167]. The principal aim of the code validation approach is to check the correct sequencing of instructions of an application program. The sequence of instructions in an application program can be represented by a directed graph, called the program graph represented by $G = < N, E >$. N represents the set of nodes in the program graph where each node represents a machine code and its associated address for a single instruction. The edges $E = \{x \mid x \in (N \times N)\}$ represent the directed edges (control flow) between the two nodes (instructions). A node in a program graph is called a merge node [167] if it has more than one incoming arc. By disconnecting from every merge node all of its incoming arcs, a program graph is partitioned into a collection of disconnected subgraphs where each of them correspond to a set of instructions in the program or a subprogram. Each such subprogram has exactly one entry point, corresponding to the merge node, and one or more exit points, including but not necessarily limited to the leaf nodes of the subgraph tree. Since each subgraph is a tree there is a unique path or unique sequence of instructions from the entry point to each of the exit points.

For a total number of K subprograms (subgraph) the ith subprogram can be $G = \langle V_i, E_i, e_i, V_{if}, P_i \rangle$ represented as an acyclic digraph:

where,

$(\forall i) = 1$ to K

$V_i = \{v_{i1}, v_{i2}, \ldots, v_{in}\}$ represents the nodes in the ith subgraph.

$E_i = \{x \mid x \in (V_i \times V_i)\}$ represents control flow edges between the nodes in the subprogram.

e_i, is the initial node (merge node) of the ith subgraph.

$V_{if} = \{x \mid x \in V_i \wedge x \text{ is a leaf node}\}$.

$\mid V_{if} \mid = M$, represents the number of paths in the ith subgraph. If L represents the number of machine codes (nodes) in the jth path of the ith subgraph, then

$(\forall j) = 1$ to M

$P_i = \{x \mid x = \{x_{j1}, x_{j2}, \ldots x_{L-1}, x_{jL}\} \wedge x_{j1} = e_i \wedge x_{jL} \in V_{if} \wedge (\forall q) = 1 \text{ to } L\ (x_{jq}, x_{j(q+1)}) \in (V_i \times V_i)\}$, represents the set of elementary paths in the ith subprogram.

The program graph can be compacted into a streamlined equivalent graph called the control flow graph (CFG), where each node in the CFG represents a subprogram of the program graph. The entry point of a subprogram is either the instruction stored immediately following a branch instruction or a destination of a branch instruction. The exit point is either a branch instruction or the instruction immediately preceding a branch destination [151]. Arcs in the CFG represent a valid control flow between subprograms that correspond to an exit point of a subgraph, represented by the source node of the arc.

Though the static control flow analysis of machine code has to overcome several difficulties [14] an appropriate file format and some architecture specific heuristics makes the problem manageable. Whether the application program is developed using a compiler or assembler the executable code is available in Intel hex file or any other standard format which contains absolute address, code and

checksum. Intel hex file format is widely used in microprocessors and microcontrollers as de-facto standard for representation of code for programming into microelectronic devices. With a specific architecture of the target processor on which the code is to be executed many of the problems associated with the static slicing of executables become manageable.

For the implementation of the code validation first the machine code is read from the Intel hex file and stored in an array. For the target processor the entry points and exit points in the program graph can be identified from the conditional and unconditional control transfer instructions [151] and their destination addresses which are available from the popular Intel hex file or binary file format. While constructing the program graph a flow array is created first, this represents each code with its source and destination addresses. Finding nodes (instructions) in the program graph without incoming edges, the dead codes are eliminated for the construction of CFG. Dead code elimination also helps in the efficient utilization of the available hardware. Merge nodes are identified and the CFG is constructed by eliminating the incoming arcs of the merge nodes. Within each subprogram the codes and their addresses of all valid paths are identified. The sequence of instructions executed by the processor corresponds to a path in the program graph. Each rule is applied independently and the instruction sequence in all possible execution paths are checked for compliance of the rule and inferences are reported if any rule is violated.

4.1.4 Codification of Rules

Rules of inferences are formulated for the target processor after conducting a through analysis of their architectural features as well as instruction set. The processor specific requirements in the machine code sequence are stipulated as rules of inferences and are of the form

(premises $\Rightarrow$ consequent) or (antecedent $\Rightarrow$ consequent)

The premises and consequences are expressed in propositional logic formulae. The premises are a sequence of machine code pattern and the consequences are the set of prerequisites or post requisites in the machine code pattern/sequence for the corresponding premises.

The set of machine codes for a target processor forms the universal set represented by ‘Z’.

The set of ‘s’ number of rules stipulated for the processor is represented by

$R = \{r_1, r_2, r_3, \ldots\ldots\ldots r_s\}$

Then $(\forall k) = 1$ to s

The set of ‘v’ machine codes or their hex values representing the pattern/ sequence generated for the kth rule, $M_k = \{c_{k1}, c_{k2}, c_{k3}, \ldots c_{kv}\}$ where $M_k \subset Z$, can be obtained with the help of an assembler for the processor concerned. Reasoning about the sequence of machine codes for validating a prescribed rule is attempted using propositional logic.

The sequence of machine codes executed in a valid path is defined by the set

$\pi = \{x \mid x$ is the hex value of a machine code in a valid path$\}$

For coding each of the rules propositions are defined for the elements of M_k as well as among the elements of M_k as follows.

$(\forall k) = 1$ to s

$(\forall)m,n = 1$ to v

C_m: $c_{km} \in \pi$ ie. m^{th} code in the set M_k of the kth rule $\in \pi$.

$\neg C_m$: $c_{km} \notin \pi$

C_{mn}: $(c_{km} \in \pi) \wedge (c_{kn} \in \pi) \wedge (c_{km}$ precedes $c_{kn}) \wedge m \neq n$ ie; (c_{kn} succeeds c_{km})

C_{mnd}: $(c_{km} \in \pi) \wedge (c_{kn} \in \pi) \wedge (c_{kn}$ immediately follows $c_{km}) \wedge m \neq n$

$C_{x\text{-}y}$: any hex code in the range x-y $\in \pi$ where the range of consecutive hex codes x-y $\in M_k$.

Now ($\forall$m,n,o,p,q,r) = 1 to v

The rules can be coded based on its property to the following typical formulas.

$\mathbf{A}(C_m \Rightarrow C_o)$

$\mathbf{A}(C_{mn} \Rightarrow C_q \wedge (C_o \vee C_p))$

$\mathbf{A}(C_m \wedge C_o \Rightarrow C_q \wedge C_r)$

$\mathbf{A}(C_m \wedge C_o \Rightarrow C_q \wedge (C_r \wedge \neg C_n))$

$\mathbf{A}(C_n \Rightarrow C_{mn})$

$\mathbf{A}(C_m \Rightarrow C_{mnd})$

$\mathbf{A}(C_{x\text{-}y} \wedge C_o \Rightarrow C_m \wedge \neg C_p)$

Where '**A**' is the path quantifier "for every path" in temporal logic [29, 175].

4.1.5 Analysis Technique

Soundness of the rules is checked using the principle that $f_1 \Rightarrow f_2$ is false only when f_1 is true and f_2 is false [169]; for all other conditions it is true. Compliance of the rules is checked individually. For a rule encoded in the form $\mathbf{A}(f_1 \Rightarrow f_2)$ checking of truth value of f_1 (premises) leads to the checking of truth value of f_2 (consequent). If in any of the possible execution paths, $\pi \models_i f_1$, where i $\leq$ $|\pi|$, then all the codes that would get executed when f_1 is true are marked. Then the satisfiability of f_2 is checked by traversing these nodes in the graph backward or forward based on the requirement. The result is achieved with a machine code

pattern matching as well as an on-the-fly disassembler and on-the-fly state space creation wherever necessary.

A sample program segment of a data acquisition system developed in assembler for PIC16F877 microcontroller given in Table 4.3 is used to describe the partitioning concepts for the abstraction of CFG. Fig. 4.1(a) gives the program graph which is constructed as described in section 4.1.3. Fig. 4.1(b) shows the formation of subprograms **1**, **2** and **3** by eliminating the incoming arcs of the merge nodes 6 and 8 and Fig. 4.1(c) gives the CFG. There exists valid control flow between instructions in a subprogram which would appear to be a cycle in the subprogram represented by this node. An isolated node in the CFG may indicate the presence of an interrupt service routine in the application program. Each node in the CFG is represented by the address location of its associated instruction. Using the CFG the entire program can be broken down into a fixed number of unique execution paths similar to Ball-Larus paths [176]. For the program considered in Table 4.3, the two paths identified are 0 to 5→6→B→C→D→7→8→A and 0 to 5 →6→B→C→D→7→8→9. The analysis is done on a sequence of nodes where each node represents an instruction so that the instructions associated with each execution path can be analyzed to find any invalid sequence and the programmer is notified with suitable warnings. For validation of the code the rules formulated are applied so that the code or code sequence in each path is compared with a code or code sequence provided by each rule. Each comparison results in setting or resetting of a number of monitoring flags. The result of one test determines the next code or code sequence to be compared with the code or code sequence that succeeds or precedes the execution path.

Table 4.3 A sample assembly language program used to describe the partitioning concepts and analysis of the proposed code validation technique.

```
MPASM  5.03          IOP_PRG_1_ADC.TXT   1-19-2010  13:35:57       PAGE  1

LOC    OBJECT                                SOURCE TEXT
       CODE
                        LIST    p=16F877 ; PIC16F877 is the target processor
                        #include "P16F877.INC" ; Include header file
                        LIST
                        P16F877.INC  Standard Header File, Version 1.00    Microchip Technology, Inc.
                        LIST
                             org     00          ; Start up vector.

0000   1683       adconfig  bsf      STATUS,RP0        ; select bank1
0001   3002                 movlw    0x02              ;config.byte foradcon1.
0002   009F                 movwf    ADCON1            ;leftjust,000, DDDAAAAA(pcnfg3:pcfg0)
0003   1405                 bsf      TRISA,00          ;porta bit0 analog input.
0004   1283                 bcf      STATUS,RP0        ; select bank0.
0005   141F                 bsf      ADCON0,ADON       ;ADON,channel-0,fosc/2.
0006   200B       loop      call     delay             ;provide acq.time after channel select.
0007   151F                 bsf      ADCON0,02         ;issue SOC.
0008   191F       polling   btfsc    ADCON0,02         ;polling ADCON0 DONE bit0?
0009   2808                 goto     polling           ;wait.
000A   2806                 goto     loop              ; repeat.
000B   0000       delay     nop
000C   0000                 nop
000D   0008                 return
                            END
```

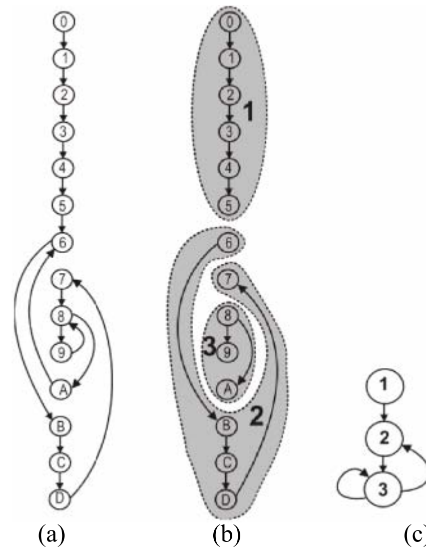


*Fig. 4.1 CFG abstraction details for the sample program given in Table 4.3.(a) shows the program graph, (b) shows the formation of subprograms **1**, **2** and **3** by eliminating the incoming arcs of the merge nodes 6 and 8 whereas (c) shows the CFG.*

Analysis starts with the initial node of the first subprogram. At this point the *reset* condition of the embedded processor is assumed with all registers assigned their reset values and the status monitoring flags are initialized accordingly. At the exit point of each node in the CFG a number of conclusions are made based on which the analysis of the next node is done. At a merge node the conclusions made in the entire possible path to the merge node determines the analysis of the sequence of instructions that follows.

Validation of the given sample program by checking the compliance of one of the rules stipulated for the peripheral Analog to Digital Converter (ADC) integrated to the processor is explained as follows. For the correct operation of the ADC, proper configuring of registers (*adcon0* and *adcon1*) associated with this peripheral is necessary. Hence after configuring the register *adcon1* appropriately, the device should be made ON by setting the bit *adcon0*<*ADON*>, before issuing start of conversion by setting *adcon0*<2>. This results in a machine code sequence 0x009F, 0x141F, 0x151F. Coding of this rule is done with the following propositional logic formula.

$\mathbf{A}(C_1 \wedge C_3 \Rightarrow C_{12} \wedge C_{23})$, where

C_1: 0x009F $\in \pi$; C_2: 0x141F $\in \pi$ and C_3: 0x151F $\in \pi$.

The antecedent of the above formula is satisfied at node 7 of subprogram **2** of Fig. 4.1(b). Searching for the consequence backwards in the path validates this rule at node 5 of subprogram **1**. Otherwise the analysis can pin point a missing *adc on* instruction for the code at location 7h if the machine code 0x141F $\notin \pi$ for the program segment considered. Adding new checks only requires formulating the rule, identifying code patterns and expressing the rule using propositional logic.

4.2 Tool Chain

The tool chain developed explains the practicality of this approach as shown in Fig. 4.2. The application program developed in assembler or compiler in the form of Intel hex file is read and the CFG is created. The rules developed are applied one by one to the possible execution paths and the possible bugs in the program are reported with suggestions for error corrections if any. In fact the machine code from the compiler/assembler is subjected to a new pass based on rules framed for this analysis and hence is a post pass analysis. This pass results in a stream of machine code with the probable errors governed by the rules eliminated and optimized for code efficiency.

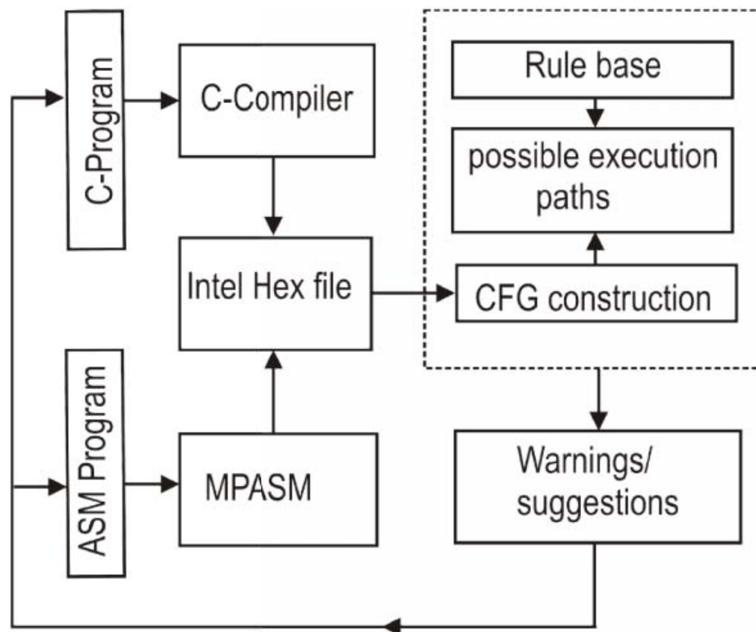


Fig. 4.2 The tool chain used for the proposed validation technique of embedded system machine codes.

4.3 Feasibility Study on PIC16F87X MCU

This section describes the different aspects of a target processor that is considered for a case study. Most of the embedded systems are based on microcontrollers, which are special purpose computers on a single chip and a wide variety of them are available for various applications. The feasibility of this approach on systems based on PIC16F87X series of microcontrollers is considered. These are products of Microchip Technology Inc., and these MCUs offer high performance and low cost. Due to the availability of processors with wide range of memory, I/O and integrated peripheral configurations they are highly suitable for various industrial/consumer applications and systems. This family of microcontrollers constitutes a RISC-based Harvard architecture with instruction sizes of 14 bits and a data bus 8-bit wide [165,166]. There are three memory categories such as program memory, data memory and EEPROM data memory which are implemented with different technologies. They have 13-bit program counter capable of addressing an 8K × 14 program memory space. The PIC16F876/877 devices have 8K×14 bit words of FLASH program memory and the PIC16F873/874 devices have 4K×14. The *reset* vector is at 0000h and the interrupt vector is at 0004h. The data memory is partitioned into four banks which contain the general purpose registers and the Special Function Registers (*SFR)*. Bits *RP1; bit six of status register* (*status*<*6*>) and *RP0* (*status*<*5*>) are the bank select bits. The register file can be accessed either directly or indirectly through the file select register (*fsr*). Each PIC16F87X instruction is a 14-bit word, divided into an opcode which specifies the instruction type and one or more operands which accomplish the operation of the instruction.

A thorough analysis of various general purpose instructions and those used for configuring the integrated peripherals of the target microcontrollers has been

conducted. The likelihood of occurrence of typical instructions in a stream is explored and necessary rules are formed in a well-defined manner. The instruction set of PIC16F87X series of microcontrollers is highly orthogonal and is divided into byte oriented, bit oriented and literal and control operations. Analysis is done on machine code by comparison of bit patterns thereby identifying the opcodes and operands, whatever is the language/compiler adopted for developing the source code.

For this target processor, the general purpose registers and the special function registers used in a program can be identified by considering the active memory bank associated with each instruction. A state transition diagram developed for the processor (described in section 5.2.1), is used to determine the active data memory bank associated with each machine code in the instruction stream. So the atomic proposition C_m: $c_{km} \in \pi$ ensures that the machine code c_{km} is preceded by the required memory bank switching instructions. Configuring an SFR can be done either using a bit set/ reset instruction or moving the required literal to *w* register and transferring the *w* register content to the SFR. The analysis takes care of all possible combinations of instructions appropriate to the situation. The content of *w* register varies with each instruction that moves a data to this. So an on-the-fly state space is created to get the *w* register content associated with each instruction. So a proposition C_{wap} is defined as

C_{wap}: the *w* register content associated with the write to SFR instruction is appropriate.

Hence the program states are identified as a combination of machine code patterns and the state spaces created, which drastically reduces the memory requirements. The prescribed rules and the approach can be easily extended and changed to fit other platforms as well.

4.4. Code Validation and Error Detection

A code validation and error detection system is of great importance during a program development process. Debugging tries to locate and fix faults or bugs after failures are detected during test or use [81]. Many fault localization techniques used in current debugging tools (e.g., setting breakpoints) were developed in the 1960s and have changed little [86]. A fault in the instruction stream which amounts to a bug in the program is rectified by the system in three phases namely fault localization, fault diagnosis and fault correction. Previous studies [129] found that locating faults is the most difficult and important task in debugging.

4.4.1 Fault Localization

While traditional debugging techniques such as dumping memory, scattering print statements, setting breakpoints by users, and tracing program execution only provide utilities to examine a *snapshot* of program execution; users have to use their own strategies to do fault localization [81]. Shapiro [64] and Renner [65] proposed an interactive fault diagnosis algorithm, where the target program is recursively searched until bugs are located and fixed. With this method, users can only point out procedures that contain bugs; other debugging tools are needed to debug the faulty procedures. Many prototype debugging systems have been developed based on knowledge based approach since the early 1980s [67, 68]. These prototype systems can only handle restricted fault classes and very simple programs.

Locating the fault is attempted by making use of rules that are formed for the use and configuring of registers associated with the core as well as peripherals. For the proper functioning of the system the peripherals integrated to the devices

must be initialized and configured properly. Rules are formulated based on reasoning obtained by heuristics that are parameterized on the architectural parameters as well as the instruction set of the processor. These rules hence formed are processed to locate an illegitimate/out of place code in the instruction stream. Around one hundred rules have been formed for this family of microcontrollers of which validation technique for some of them are described in this chapter. The governing rules are listed in Table 4.4 (a-i).

In order to explain the formation of the rules that governs this analysis the peripheral *ADC* integrated to this processor is considered as a typical example. In this family of microcontrollers the *ADC* has the unique feature of being able to operate while the device is in *sleep* mode. To operate in sleep, the *ADC* clock must be derived from its internal *RC* oscillator. When the clock source is another clock option (not *RC*), a *sleep* instruction will cause the present conversion to be aborted and the *ADC* module to be turned off, though the *ADON* bit will remain set. When the *RC* clock source is selected, *ADC* module waits one instruction cycle before starting the conversion. This allows the *sleep* instruction to be executed, which eliminates all digital switching noise from the conversion. To allow the conversion to occur during *sleep*, ensure the *sleep* instruction immediately follows the instruction that sets the start of conversion bit [166]. These facts are formulated into the following rule.

If sleep instruction immediately follows the instruction that sets the ADCON0's GO /$\overline{DONE}$ bit (bsf adcon0, GO), then the clock select bits of ADCON0 <ADCS1:ADCS0> should be set.

Table 4.4 (a) to (i) List of governing rules formed for the PIC16F87X microcontrollers.

Table 4.4(a)

Rules evaluating the configuration of I/O port and CPU core registers
1. The PCON<$\overline{POR}$> bit must be set after a power on reset.
2. After a POR, using indirect addressing without loading a valid data into FSR is illegal as FSR content is don't care/unknown after a reset condition.
3. After a POR a CALL or GOTO instruction shall see that the PCLATH register is suitably programmed to access the required page of the program memory.
4. Any instruction that affects status, carry or digital carry bit with its destination register as Status register results in an illegal opcode.
5. A bank selection instruction that will not change the active memory bank is a redundant instruction.
6. Use of ADRESH and ADRESL registers as general purpose registers when ADCON0 <ADON> bit is set shall raise a warning even before giving SOC command.
7. A call instruction without a corresponding return/retlw instruction is invalid.
8. A flag checking instruction shall be preceded by a corresponding flag affecting instruction.
9. If the device is PIC16F873/876, the use of PORTD or PORTE register is invalid
10. When the bidirectional ports are used their corresponding TRIS register, which is the data direction register should be configured appropriately.
11. If TRISE<PSPMODE> is made set, then TRISE<2:0> must be configured as inputs and ADCON1<PCFG3:PCFG0> must be set to configure pins RE2:RE0as digital I/O.

Table 4.4(b)

Rules for the Read and Write of EEPROM & Flash program memory
1. An instruction that clear EECON1 <WR> or EECON1 <RD> is redundant.
2. An instruction that sets EECON1 <WREN> and <WR> together is illegal.
3. A bsf eecon1, WR instruction must be preceded by the sequence bsf eecon1, WREN, bsf eecon1, EEPGD or bcf eecon1, EEPGD and bcf p1r2, EEIF.
4. A bsf eecon1, RD must be preceded by either a bsf eecon1, EEPGD or bcf eecon1, EEPGD.
5. If bsf eecon1, RD is preceded by a bcf eecon1, EEPGD, then it shall be preceded by a movwf EEADR which shall be preceded by a movlw k instruction and the k value msb must be clear if the device is PIC16F873/ 874.
6. If a bsf eecon1, WR instruction is preceded by a bcf eecon1, EEPGD then it shall be preceded by a bcf intcon, GIE; movlw 55h; movwf eecon2; movlw AAh; movwf eecon2 and shall be followed by bcf eecon1, WREN.
7. If bsf eecon1, RD is preceded by a bsf eecon1, EEPGD, then it shall be preceded by a movwf eeadrh; which shall be preceded by a movlw k instruction and the k value's 4-msbits must be clear if the device is PIC16F873/ 874and the k valu's 3 msbits must be clear if the device is PIC16F876/ 877 and it shall be preceded by a movwf eeadr and it shall immediately follow two NOP instructions.
8. If a bsf eecon1, WR instruction is preceded by a bsf eecon1, EEPGD then it shall be preceded by a movwf eeadrh which shall be preceded by a movlw k instruction and the k value's 4-msbits must be clear if the device is PIC16F873/ 874and the k valu's 3 msbits must be clear if the device is PIC16F876/ 877 and it shall be preceded by a movwf eedath; which shall be preceded by a movlw k instruction and the k value's 2-msbits must be clear; movwf eeadata; bcf intcon, GIE; movlw 55h; movwf eecon2; movlw AAh; movwf eecon2 and shall be followed by two NOP instructions; bcf eecon1, WREN.

Table 4.4(c)

Rules validating the configuration of ADC module
1. Instruction that configure ADCON1 register shall occur before instruction that configure ADCON0 register. *2. If ADCON0 <ADON> is made set, then ADCON1<PCFG3:PCFG0> shall not be configured as b'011x' (all port bits configured as digital I/O).* *3. If ADCON0 <CHS2:CHS0> is >b'101' and the device is 16F876/873 give a warning signal.* *4. ADC channel select instruction shall occur only once unless a new channel is to be selected.* *5. Port pin's configuration ADCON1<PCFG3:PCFG0> shall correspond to the analog channel select ADCON0 <CHS2:CHS0> and their corresponding TRIS register must be set.* *6. If sleep instruction immediately follows the instruction that sets the ADCON0's GO /$\overline{DONE}$ bit (bsf adcon0, GO), then the clock select bits of ADCON0 <ADCS1:ADCS0> should be set.* *7. ADC is on and no configuration done on ADCON0 <CHS2:CHS0> until issue of a SOC implies selection of channel_0, so the analog input channel AN0 must have the corresponding TRIS bit selected as input.* *8. If ADC is on and no configuration done on ADCON0<ADCS1:ADCS0> implies the default clock of FOSC/2. Verification shall be done on this selection using the clock input value of the programmer.* *9. If the clock selection bit configuration ADCON0 <ADCS1:ADCS0> does not matches with the minimum T_{AD} time of 1.6 µs after evaluating the same from the clock input value of the programmer, either the error or the correct configuration can be prompted.* *10. If a programming on ADCON0 <ADCS1:ADCS0> is done for a second time in the instruction stream, a warning shall be generated.* *11. After selecting an analog input channel ensure that a time corresponding to T_{ACQ} is left before setting ADCON0 <GO/$\overline{DONE}$ > bit.* *12. If ADCON0 <GO/$\overline{DONE}$ > bit is made set in the same instruction that turns on ADCON0 <ADON> bit, report the presence of an illegal opcode.* *13. Configuration of A/D result format select bit ADCON1<ADFM> shall be considered when ADRESH and ADRESL registers are read.* *14. Reading ADRESH:ADRESL registers without reading ADCON0<GO/$\overline{DONE}$ > bit and ensuring its zero status is an error.* *15. If an instruction to set P1E1<ADIE> bit is there, there should be an instruction to set INTCON<GIE> bit, provided the program contains the corresponding interrupt service routine.* *16. Any instruction to test the status of P1R1<ADIF> bit indicates the interrupt mode of operation of ADC, so the satisfactory configuration of ADC shall be checked.* *17. If the instructions to set INTCON<GIE> and P1E1<ADIE> bits are in the instruction stream, an instruction to clear the P1R1<ADIF> bit shall be there in the ISR.* *18. Without setting ADCON0 <ADON> bit, setting ADCON0 <GO/$\overline{DONE}$ > bit results in a possible missed instruction.*

Table 4.4(d)

Rules validating the configuration of Timer0 module
1. *Configuring of OPTION_REG <T0CS> and clearing of INTCON<T0IF> shall precede a write to TMR0 register.*
2. *If OPTION_REG<T0CS> is made set (counter mode), an instruction to set TRISA<4> shall precede.*
3. *Polling of INTCON<T0IF> shall be followed by clearing of INTCON <T0IF>.*
4. *The use of an instruction to clear INTCON<T0IE> shall precede configuring of OPTION_REG<T0CS>, a write to TMR0 register and shall be followed by polling of INTCON<T0IF> instruction.*
5. *An instruction to set INTCON<T0IE> shall be preceded by instructions to clear INTCON<T0IF> and set INTCON<GIE> or all this done simultaneously.*
6. *Occurrence of an instruction to set INTCON<T0IE> shall precede configuring of OPTION_REG <T0CS>and a write to TMR0 register.*
7. *IF an instruction to set INTCON<T0IE>occur in the main program, the ISS shall contain an instruction to clear INTCON<T0IF> before re-enabling INTCON<T0IE>.*
8. *A sleep instruction shall not immediately follow a write to TMR0 register.*
9. *A prescaler assignment shall precede a rate select which shall precede any write to TMR0 register.*
10. *If the prescaler is assigned to WDT (OPTION-REG<PSA>=b'1') then a clrwdt instruction shall precede an instruction that assigns PSA to TMR0.*
11. *If the prescaler is assigned to TMR0 (OPTION-REG<PSA>=b'0') then a clrf tmr0 instruction shall precede an instruction that assigns PSA to WDT and a clrwdt instruction shall follow immediately.*

Table 4.4(e)

Rules validating the configuration of Timer1 module
1. *T1CON<TMR1ON> is made set only after configuringT1CON register.*
2. *If T1CON<T10CSEN> is made set then instruction that set TRISC<1:0> is redundant.*
3. *If T1CON<TMR1CS> is made clear then configuring T1CON<*$\overline{T1SYNC}$*>is redundant.*
4. *If T1CON<TMR1CS> is made set and T1CON<T1OSCEN>is made clear then a instruction to set TRISC<RC0>shall follow.*
5. *If T1CON<TMR1CS> is made set and T1CON<* $\overline{T1SYNC}$ *>is made clear then a sleep instruction shall not follow.*
6. *Configuring T1CON<*$\overline{T1SYNC}$*:TMR1CS >=b'11' (TMR1 in asynchronous counter mode) followed by configuring CCP1CON or CCP2CON register in capture or compare mode is invalid.*
7. *If T1CON<*$\overline{T1SYNC}$*:TMR1CS >=b'11' (TMR1 in asynchronous counter mode) then a write to TMRH or TMRL shall be preceded by an instruction to clear T1CON<TMR1CS> (stop timer).*
8. *If T1CON<*$\overline{T1SYNC}$*:TMR1CS >=b'11' (TMR1 in asynchronous counter mode) then a read TMR1H register if any shall occur before any read TMR1L register.*
9. *Before reading or writing a TMRH or TMR1L register the status of INTCON<GIE> shall be b'0' (clear or all interrupts are disabled).*
10. *If T1CON<TMR1CS >and T1CON<T1OSCEN > are made set, then the software shall include a delay routine before T1CON<TMR1ON> is made set.*
11. *In special event trigger mode with CCP1 or CCP2, configuring of TRM1 in interrupt mode is redundant.*
12. *Any combination of the sequence bsf p1e1<TMR1IE>, bsf intcon<GIE>, bsf intcon<PEIE> shall be preceded by a P1R1<TMR1IF> clear instruction.*
13. *If any combination of the sequence bsf p1e1<TMR1IE>, bsf intcon<GIE>, bsf intcon<PEIE>, then the ISS shall contain an instruction bcf p1r1<TMR1IF> before re-enabling P1E1<TMR1IE>.*
14. If *P1E1<TMR1IE>is not made set and T1CON<TMR1ON> is made set then a bcf p1r1<TMR1IF> instruction shall precede and an instruction bit test on P1R1<TMR1IF>shall follow.*

Table 4.4(f)

Rules validating the configuration of Timer2 module
1. An instruction to set P1E1<TMR2IE> shall be preceded by a bcf p1r1<TMR2IF> instruction.
2. If P1E1<TMR2IE> is made set either INTCON <PEIE>or both INTCON <GIE:PEIE> shall be made set.
3. A write to TMR2 or PR2 register shall occur before T2CON<TMR2ON> is made set.
4. If P1E1<TMR2IE>, INTCON <GIE:PEIE> are made set then ISS shall contain a bcf p1r1<TMR2IF>before re-enabling P1E1<TMR2IE>.
5. A polling of P1R1<TMR2IF> shall be followed by a bcf p1r1<TMR2IF>instruction.

Table 4.4(g)

Rules validating the configuration of Capture/Compare/PWM module
1. *If CCP1CON<CCP1M3:CCP1M0> is configured in capture mode and CCP2CON<CCP2M3:CCP2M0> in compare mode then CCP2 module shall be in special event trigger mode.*
2. *If CCP2CON<CCP2M3:CCP2M0> is configured in capture mode and CCP1CON<CCP1M3:CCP1M0> in compare mode then CCP1 module shall be in special event trigger mode.*
3. *If CCP1CON<CCP1M3:CCP1M0> is configured in compare mode and CCP2CON<CCP2M3:CCP2M0> neither disabled nor in capture mode, then it shall be in special event trigger mode.*
4. *If CCP2CON<CCP2M3:CCP2M0> is configured in compare mode and CCP1CON<CCP1M3:CCP1M0> neither disabled nor in capture mode, then it shall be in special event trigger mode.*
5. *If CCP1CON/CCP2CON are enabled in any of the capture/compare mode, then an instruction to clear P1R1<CCP1IF>/ P1R2<CCP2IF>shall precede.*
6. *If P1E1<CCP1IE>/ P1E2<CCP2IE> and INTCON<GIE:PEIE> are made set, then P1R1<CCP1IF>/ P1R2<CCP2IF> must be cleared in the interrupt service routine before re-enabling P1E1<CCP1IE>/ P1E2<CCP2IE>*
7. *Polling of P1R1<CCP1IF>/ P1R2<CCP2IF> must be followed by clearing of those bits.*
8. *If CCP1CON<CCP1M3:CCP1M0> is configured in capture mode, then TRIS<RC2> shall be made b'1' (input).*
9. *If CCP2CON<CCP2M3:CCP2M0> is configured in capture mode, then TRIS<RC1> shall be made b'1' (input).*
10. *If CCP1CON<CCP1M3:CCP1M0>/ CCP2CON<CCP2M3:CCP2M0> is configured in capture/compare mode T1CON<1:0> shall be configured for TMR1 enabled, TMR1CS-0(internal clock) [i.e. Timer1 in timer mode] or T1CON<2:0> shall be configured for TMR1 enabled, TMR1CS-1(external clock) and* $\overline{T1SYNC}$*-0[Timer1 in synchronous counter mode].*
11. *A capture mode change instruction for CCP1CON register shall be preceded by clearing bit PIE1<CCP1IE> and followed by clearing P1R1<CCP1IF>*
12. *A capture mode change instruction for CCP2CON register shall be preceded by clearing bit PIE2<CCP2IE> and followed by clearing P1R2<CCP2IF>*
13. *If CCP1CON<CCP1M3:CCP1M0> is configured in compare mode, then TRIS<RC2> shall be made b'0' (output).*
14. *If CCP2CON<CCP2M3:CCP2M0> is configured in compare mode, then TRIS<RC1> shall be made b'0' (output).*
15. *If CCP1CON<CCP1M3:CCP1M0> is configured in PWM mode, then TRIS<RC2> shall be made b'0' (output).*
16. *If CCP2CON<CCP2M3:CCP2M0> is configured in PWM mode, then TRIS<RC1> shall be made b'0' (output).*
17. *Configuring CCP1CON<CCP1M3:CCP1M0> for PWM operation shall be preceded by a write to PR2 register, write to T2CON to enable Timer2 and set TMR2 prescale value and a write toCCPR1L register and CCP1CON<5:4> bits.*
18. *Configuring CCP2CON<CCP2M3:CCP2M0> for PWM operation shall be preceded by a write to PR2 register, write to T2CON to enable Timer2 and set TMR2 prescale value and a write toCCPR2L register and CCP2CON<5:4> bits.*
19. *A CLRWDT instruction shall be executed before a SLEEP instruction.*

Table 4.4(h)

Rules validating the configuration of USART module
1. Instructions that set or clear PIR1<TXIF> and PIR1<RCIF>are illegal.
2. Any instruction to load data into TXREG for the first time shall be preceded or followed by an instruction that sets TXSTA<TXEN>.
3. If any instruction to clear TXSTA<TXEN> precedes an instruction that loads data into TXREG, then an instruction that sets TXSTA<TXEN> shall follow.
4. An instruction to set TXSTA<TX9> shall be followed by an instruction to set or clear TXSTA<TX9D>prior to an instruction that loads data into TXREG.
5. Instruction sequence that set RCSTA<SPEN> and clear TXSTA<SYNC> shall be preceded by loading of SPBRG register for the appropriate baud rate.
6. Instruction sequence that set RCSTA<SPEN>, PIE1<TXIE> and clear TXSTA<SYNC> shall have instructions that set INTCON<PEIE> and INTCON<GIE> in that execution path prior to an instruction that sets TXSTA<TXEN>.
7. A third load instruction to TXREG register onwards shall be preceded by a bit test instruction on TXSTA<TRMT>.
8. Instruction sequence that set RCSTA<SPEN> and clear TXSTA<SYNC>shall ensure that TRISC<7:6> is made b'10'.
9. Instruction sequence that set RCSTA<SPEN>, PIE1<TXIE> and clear TXSTA<SYNC> shall have instructions that set INTCON<PEIE> and INTCON<GIE> in that execution path prior to an instruction that sets RCSTA<CREN>.
10. Instruction sequence that set RCSTA<SPEN>, clear TXSTA<SYNC> shall have an instruction that sets RCSTA<CREN> prior to a reading of RCREG register instruction.
11. A reading of RCREG register instruction shall be preceded by a bit test instruction on PIR1<RCIF> if not inside an interrupt service routine.
12. A reading of RCREG register instruction shall be preceded by a reading of RCSTA register instruction or an instruction to bit test on RCSTA<RX9D> if there exist an instruction that sets RCSTA<RX9> in that execution path or an instruction to bit test on RCSTA<FERR>.
13. If there exists an instruction that sets RCSTA<RX9>, it shall be prior to an instruction that sets RCSTA<SREN> or RCSTA<CREN>.
14. If an instruction sequence that set RCSTA<SPEN>, TXSTA<SYNC>, TXSTA<CSRC>, TXSTA<TXEN>, loads TXREG is followed by an instruction that sets RCSTA<SREN>, it shall be followed by an instruction that clear TXSTA<TXEN>.
15. If a sleep instruction immediately follows a load TXREG register instruction, then a bit clear instruction on TXSTA<CSRC> shall precede.
16. An instruction sequence that set RCSTA<SPEN>, TXSTA<SYNC>, clear TXSTA<CSRC> shall be followed by instruction sequence that clear RCSTA<SREN> and RCSTA<CREN> before an instruction that set TXSTA<TXEN> .
17. If a sleep instruction follows an instruction sequence that set RCSTA<SPEN>, TXSTA<SYNC>, clear TXSTA<CSRC> there shall be an instruction that sets RCSTA<CREN> prior to the sleep instruction.

Table 4.4(i)

Rules evaluating the configuration of Master Synchronous Serial Port Module
1. *If SSPCON<SSPM3:SSPM0>is configured as b'0100' or b'0101' (SPI-slave mode) then any instruction that set SSPSTAT<SMP> is illegal.*
2. *If SSPCON<SSPM3:SSPM0> is set to SPI-master mode then any instruction that set SSPCON<SSPEN> shall be preceded by instructions that clear TRISC<5>, set TRISA<5> and clear TRISC<3>.*
3. *If SSPCON<SSPM3:SSPM0> is set to SPI- slave mode then any instruction that set SSPCON<SSPEN> shall be preceded by instructions that clear TRISC<5>, set TRISC<3 and, set TRISA<5.*
4. *When SSPCON register is configured in i2C master mode any instruction that set or clear SSPCON<CKP>is redundant.*
5. *If SSPCON<SSPM3:SSPM0> is set to b'0110' or b'0111' (i2C slave mode) then any instruction that set SSPCON<SSPEN> shall be preceded by instructions that set TRISC<4:3>.*
6. *If SSPCON<SSPM3:SSPM0> is configured in i2C master mode and SSPCON2<ACKEN:RCEN>is made b'11' then it shall be preceded by an instruction that set or clear SSPCON2<ACKDT>.*

For the target processor considered, the code sequence for the first part of this rule can be identified as 0x151F (*bsf adcon0, 2*), 0x0063 (*sleep*) and for the second part as 0x179F (*bsf adcon0, 7*), 0x171F (*bsf adcon0, 6*). This rule can be codified to the form

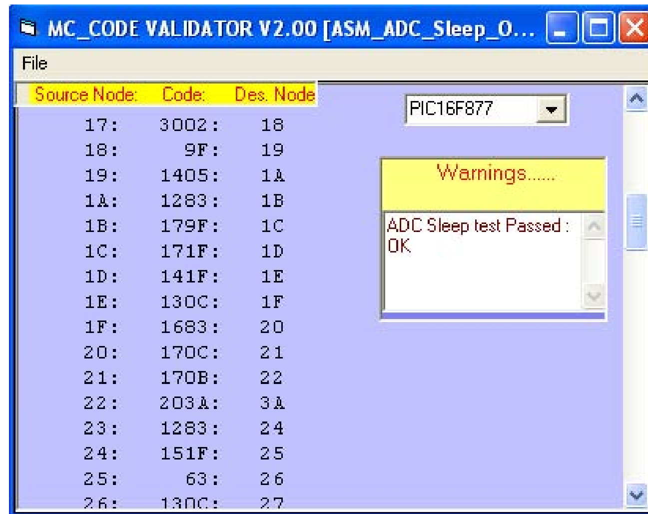
$$\mathbf{A}(C_{12d} \Rightarrow C_{31} \wedge C_{41})$$

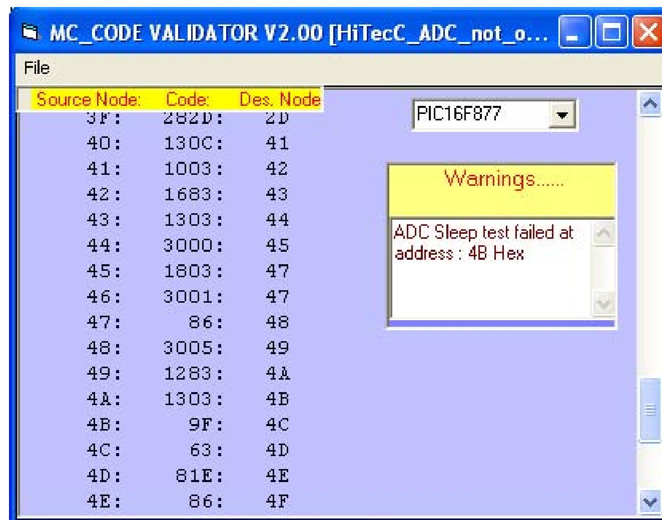
where C_1: 0x151F $\in \pi$; C_2: 0x0063 $\in \pi$; C_3: 0x179F $\in \pi$ and C_4: 0x171F $\in \pi$. The occurrence of the first code pattern identified by a linear scan through a valid execution path leads to the testing of precedence of the second code pattern in the same path and any discrepancy can be located. In case the configuring of *adcon0* register is done by moving the appropriate literal to *W* register and transferring the *W* register content to this SFR, then the codification of the rule can be of the form

$\mathbf{A}((C_{wap} \wedge C_1 \ \wedge C_{12d}) \Rightarrow (C_{31} \ \wedge C_{41}))$, where

C_1:a hex code for write to *adcon0* register (0x009F) $\in \pi$; C_2: 0x0063 $\in \pi$; C_3: 0x179F $\in \pi$ and C_4: 0x171F $\in \pi$.

The necessary tool/ program for this study and implementation is developed in Visual Basic. With the help of the graphical user interface, the user can select the required Intel hex file for validation. Programs developed for *ADC* conversion in sleep mode using assembler and high level languages like HI-TECH C as well as mikroC are tested with this technique. The screenshots shown in Fig. 4.3(a) and (b) explain the reporting of violation of the above rule for evaluation programs developed in assembler and HI-TECH C. Along with the warnings, the screen shot displays the source node address, the hex value of the opcode at this address and the destination address of each of the edges in the program graph. When this tool is integrated to the system development environment the debugging process can be made interactive and user friendly. All the peripherals integrated to the processor are considered one by one and necessary rules are formulated.


MC_CODE VALIDATOR V2.00 [ASM_ADC_Sleep_O...
File
Source Node: Code: Des. Node
17: 3002: 18
18: 9F: 19
19: 1405: 1A
1A: 1283: 1B
1B: 179F: 1C
1C: 171F: 1D
1D: 141F: 1E
1E: 130C: 1F
1F: 1683: 20
20: 170C: 21
21: 170B: 22
22: 203A: 3A
23: 1283: 24
24: 151F: 25
25: 63: 26
PIC16F877
Warnings......
ADC Sleep test Passed : OK


MC_CODE VALIDATOR V2.00 [HiTecC_ADC_not_o...
File
Source Node: Code: Des. Node
40: 130C: 41
41: 1003: 42
42: 1683: 43
43: 1303: 44
44: 3000: 45
45: 1803: 47
46: 3001: 47
47: 86: 48
48: 3005: 49
49: 1283: 4A
4A: 1303: 4B
4B: 9F: 4C
4C: 63: 4D
4D: 81E: 4E
4E: 86: 4F
PIC16F877
Warnings......
ADC Sleep test failed at address : 4B Hex

4.4.2 Fault Diagnosis

Responsibility of the fault diagnosis is to delve deeply into the bug and to determine the root cause of the malfunction. Identification of the cause is implicit in the rule by which the error is located. The bugs in the program located by the governing rules are characterized according to the causes of their occurrence. They can be due to an improper use or configuring of a register, a missed or a redundant instruction resulting in a discrepancy in the opcodes or operands, illegal codes or even a deadlock.

4.4.2.1 Discrepancy in the Opcodes or Operands

Table 4.4(a)-(i) lists certain rules formulated to diagnose the improper use of registers associated with the core as well as any discrepancy in the configuration of *SFRs* associated with *ADC* and other peripherals. Machine code sequences against some of these rules can be identified in the instruction stream as discrepancy in the opcode or operand field by applying them independently and appropriate warning or suggestions can be generated. Illustration of identifying the discrepancy in the configuration of *SFR*s associated with *ADC* for the validation of rule 7 of Table 4.4(c) is illustrated with the help of the program segment of a Data Acquisition System, given in Table 4.3. This program segment configures the *ADC* registers and issues a start of conversion with a delay. ADC is made on but no channel selection instruction included until issuing a start of conversion (*bsf adcon0, 02).* This condition assumes the default channel '0'. So the instruction stream should contain the machine code value of a *bsf trisa, 00* instruction before the machine code value of *bsf adcon0, 02*.in order to configure the tris bit corresponding to analog '*channel0*' as input. The machine code pattern for each of these propositions can be generated with an assembler. The formula that checks this condition is

$\mathbf{A}(((C_o \wedge (\neg((C_{wap} \wedge C_p) \vee C_q \vee C_r \vee C_s)))\mathbf{U}C_n) \Rightarrow C_{mn})$, where

'**U**' is the until operator in temporal logic.
C_o: 'c_{ko}'or 0x141F (*bsf adcon0,adon*) $\in \pi$
C_p: 'c_{kp}'or 0x009F (*movwf adcon0*) $\in \pi$
C_q: 'c_{kq}'or 0x159F (*bsf adcon0,03*) $\in \pi$
C_r: 'c_{kr}'or 0x161F (*bsf adcon0,04*) $\in \pi$
C_s: 'c_{ks}'or 0x169F (*bsf adcon0,05*) $\in \pi$
C_n: c_{kn}'or 0x151F (*bsf adcon0,02*) $\in \pi$
C_m: c_{km}'or 0x1405 (*bsf trisa,00*) $\in \pi$

For this program module suppose the instruction No. 3 is *bsf trisa, 01* instead of *bsf trisa, 00*. Then checking of the above formula in subprograms **1** and **2** of Fig.4.1(b) identifies the violation of this rule as a discrepancy**.** So an invalid setting of *trisa* register or a missing channel select instruction can be reported to the programmer.

4.4.2.2 Illegal Opcodes

An illegal opcode refers to an instruction opcode that does not match to any known instruction of a processor. Identifying illegal opcodes is efficient if not all bit patterns are used by the processor as instruction codes [143]. Identifying an illegitimate code from the thirty five, 14-bit instructions of the RISC architecture considered for the proposed code validation is possible by decoding the six most significant bits. Advanced simulators provide warning messages like 'Attempt to execute illegal opcode - *nop* executed' when an attempt has been made to execute an instruction opcode that does not decode to any known PIC instruction. The tool that has been developed can not only identify such codes but also identify illegal codes by considering unimplemented bits in various registers which are read as zeros and reserved bits in certain registers which are to be maintained clear always. Any literal byte selected for configuring a register which sets the reserved/unimplemented bits or any instruction that sets such a bit results in an

illegal opcode. Again configuring two bits of certain SFRs simultaneously like rule number (12) of Table 4.4(c) results in an illegal opcode. Some of the memory banks contain unimplemented data memory locations. Such address ranges in each memory bank also provides the required data for analysis. Adequate rules can be framed based on such observations.

In addition to the above, certain rules like rule number (4) of table 4.4(a) can also help in identifying an illegal code. Rule number (4) of table 4.4(a) is based on the fact that the result of an instruction with the status register as destination may be different than intended. Even though the instructions like *movwf* and *swapf* are not affecting any flag, using these instructions with status register as destination may change the *Z*, *C* or *DC* flag bits so that the previous status of these flags which was the result of a corresponding flag affecting instruction will be lost. But users may wish to save key registers like *w* and *status* on the stack while context saving during interrupts and the two instructions mentioned above are used for this purpose. So these instructions are not taken as illegal in order to avoid false warnings. For this rule the set of illegal machine codes (their hex values) obtained with the assembler is:

M_k = {783, 583, 983, 383, B83, A83, F83, 483, 883, D83, C83, 283, 683}. The formula used to identify the illegal codes will be of the form $\mathbf{A}(C_1 \vee C_2 \vee C_3 \ldots \vee C_n)$ for ‘n’ number of illegal codes. In this formula there is no implication and the truth condition of any of this proposition identifies an illegal opcode. The proposed code validation procedure can assess not only the illegitimate code but also any code that attempts to write, read or configure a register/memory improperly or those which are out of place and also generate error messages, warning signals or suggestions for correction.

4.4.2.3 Missed Instructions

When a lengthy program is developed especially in assembly language the possibility of a missing/redundant instruction cannot be ruled out. Some of the results of the analysis in this direction are described here. Rules are formulated to identify an instruction that should or should not follow its predecessor. Clustering of instructions can be done with a view to identifying an instruction that should not follow a particular instruction. Flag testing instructions, flag affecting instructions, instructions with same destination register etc. form such clusters. A typical case is that of identifying any flag testing instruction before a corresponding flag affecting instruction in any of the possible execution path. A possible missing instruction or out of place instruction can be reported. This can be tested by

$\mathbf{A}(C_n \Rightarrow C_{mn})$

Where C_n: a flag checking instruction $\in \pi$ and C_m: a corresponding flag affecting instruction $\in \pi$.

Machine code sequence governing rule 1 of Table 4.4(c) is given in Table 4.5 which forms the elements of the set M_k for this rule. Any of the *adcon0* register configuration code given in second column should occur only after one or more of the *adcon1* register configuration codes given in first which can be tested by coding the rule as shown below.

For ‘m’ number of codes in the first column and ‘n’ number of codes in the second column:

$(\forall i) = 1$ to n

$\mathbf{A}(C_i \Rightarrow C_{1i} \vee C_{2i} \vee C_{3i} \vee C_{4i} \ldots \vee C_{mi})$

Where C_i: any code in the second column of the table $\in \pi$ and

$C_1, C_2, C_3, \ldots, C_m$: any code in the first column of the table $\in \pi$.

Table 4.5 Code sequence governing rule 1 of Table 4.4(c)

(1)			***(2)***		
Hex code	***Instruction***		***Hex code***	***Instruction***	
009F	movwf	ADCON1	009F	movwf	ADCON0
101F	bcf	ADCON1,0	101F	bcf	ADCON0,0
109F	bcf	ADCON1,1	119F	bcf	ADCON0,3
111F	bcf	ADCON1,2	121F	bcf	ADCON0,4
119F	bcf	ADCON1,3	129F	bcf	ADCON0,5
139F	bcf	ADCON1,7	131F	bcf	ADCON0,6
141F	bsf	ADCON1,0	139F	bcf	ADCON0,7
149F	bsf	ADCON1,1	141F	bsf	ADCON0,0
151F	bsf	ADCON1,2	151F	bsf	ADCON0,2
159F	bsf	ADCON1,3	159F	bsf	ADCON0,3
179F	bsf	ADCON1,7	161F	bsf	ADCON0,4
			169F	bsf	ADCON0,5
			171F	bsf	ADCON0,6
			179F	bsf	ADCON0,7

Another instance of missed instruction occurs in the use of indirect addressing. Indirect addressing is possible by using the *indf* register. Any instruction using the *indf* register actually accesses the register pointed to by the file select register. The content of *fsr* is unknown on a power on reset. So any indirect addressing not preceded by a valid data loading to *fsr* (rule (2) of Table 4.4(a)) can be identified as a missed instruction. Table 4.6 shows the instructions and the corresponding codes using the *indf* register for indirect addressing. Any of the instruction in this table should be preceded by a *movlw 0x'xx'* (0x30xx) and *movwf fsr* (0x0084) instructions which can be tested similarly.

Table 4.6 Code sequence governing rule 2 of table 4.4(a)

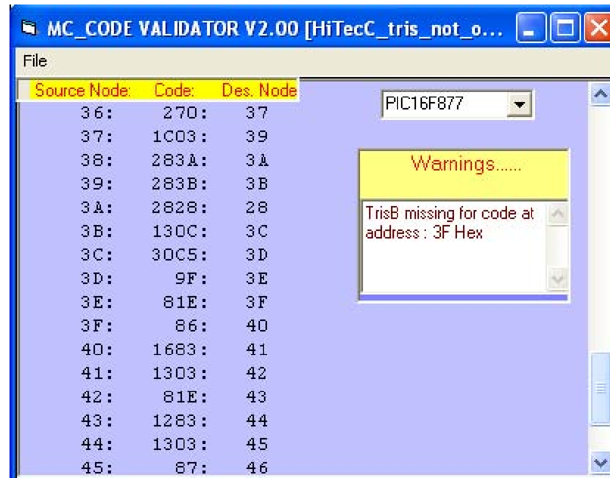
Hex code	***Instruction***	***Hex code***	***Instruction***	***Hex code***	***Instruction***
0700	addwf INDF,0	0D80	rlf INDF,1	1600	bsf INDF,4
0780	addwf INDF,1	0C00	rrf INDF,0	1680	bsf INDF,5
0500	andwf INDF,0	0C80	rrf INDF,1	1700	bsf INDF,6
0580	andwf INDF,1	0200	subwf INDF,0	1780	bsf INDF,7
0180	clrf INDF	0280	subwf INDF,1	1800	btfsc INDF,0
0900	comf INDF,0	0E00	swapf INDF,0	1880	btfsc INDF,1
0980	comf INDF,1	0E80	swapf INDF,1	1900	btfsc INDF,2
0300	decf INDF,0	0600	xorwf INDF,0	1980	btfsc INDF,3
0380	decf INDF,1	0680	xorwf INDF,1	1A00	btfsc INDF,4
0B00	decfsz INDF,0	1000	bcf INDF,0	1A80	btfsc INDF,5
0B80	decfsz INDF,1	1080	bcf INDF,1	1B00	btfsc INDF,6
0A00	incf INDF,0	1100	bcf INDF,2	1B80	btfsc INDF,7
0A80	incf INDF,1	1180	bcf INDF,3	1C00	btfss INDF,0
0F00	incfsz INDF,0	1200	bcf INDF,4	1C80	btfss INDF,1
0F80	incfsz INDF,1	1280	bcf INDF,5	1D00	btfss INDF,2
0400	iorwf INDF,0	1300	bcf INDF,6	1D80	btfss INDF,3
0480	iorwf INDF,1	1380	bcf INDF,7	1E00	btfss INDF,4
0800	movf INDF,0	1400	bsf INDF,0	1E80	btfss INDF,5
0880	movf INDF,1	1480	bsf INDF,1	1F00	btfss INDF,6
0080	movwf INDF	1500	bsf INDF,2	1F80	btfss INDF,7
0D00	rlf INDF,0	1580	bsf INDF,3		

Application of rule (15) in table 4.4(c) can identify another possible missed instruction owing to the fact that enabling *ADC* interrupt *pie1<ADIE>* without enabling *intcon<GIE>* will not generate the desired interrupt. Another possibility of a missing instruction can be due to a lack of appropriate *tris* configuration. When the bidirectional ports are used their corresponding *tris* register, which is the data direction register should be configured appropriately as stated in rule (10) of table 4.4(a). So if any of the port read or write instruction is not preceded by the required *tris* register configuring, a missing *tris* register configuring can be reported to the programmer for appropriate action. The formula is of the form

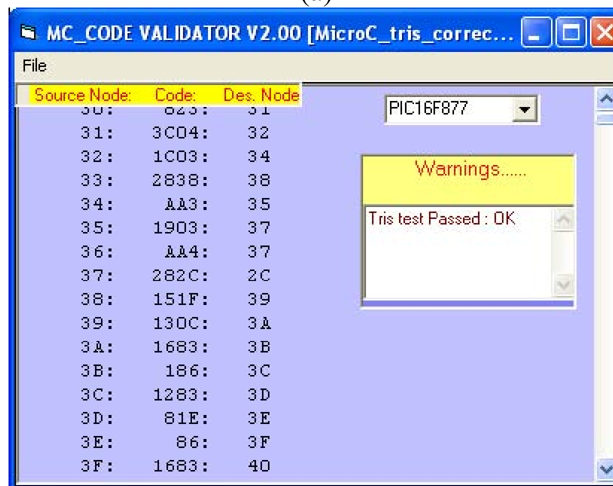
$\mathbf{A}(C_m \Rightarrow (C_n \wedge C_{wap} \wedge C_{nm}))$, where

$C_m : c_{km}$, a hex code for write/read to a port $\in \pi$ and $C_n : c_{kn}$, a hex code for write to a corresponding *tris* register $\in \pi$**.** The correct and wrong versions of the

required program is developed in HI-TECH C, mikroC and assembler and tested with the tool. Results of the analysis for a missing *trisb* setting if any, when *portB* is used as an output is shown in the screenshots of Fig. 4.4(a) and (b).The results



(a)



(b)

Fig. 4.4 Screenshots for the results of the analysis for TRISB register configuring for programs developed in high level languages. (a) in the erroneous program a warning is generated of the use of portB as output port without corresponding trisb setting at location 3Fh. (b)in the correct program the rule is validated as the antecedent and its consequent are satisfied at locations 3Eh and 3Bh respectively.

explain that the technique for validation is independent of the compiler/assembler. The required program has been developed in Visual Basic. The concept described can be extended and tailored to diagnose such errors.

4.4.2.4 A Deadlock

A possible deadlock during the execution can be identified with a particular code sequence which can be properly informed to the programmer. As an example a delay program which stores the count value in a register and its erroneous version are listed in Table 4.7. Fig. 4.5 (a) and (b) shows their representation by a directed graph respectively. Each node is numbered in **bold** and represents a single instruction. The presence of edge (**4**,**2**) in Fig. 4.5(b) indicates the possibility of a deadlock in the program and the need for a careful inspection of the code in this region can be reported.

For the instruction set of these RISC processors the use of a conditional transfer instruction is facilitated most of the time in association with an unconditional transfer instruction. So the presence of an edge from an exit node, which is a *goto* instruction that immediately follows a conditional transfer instruction, to an entry node that is not the conditional transfer instruction itself in any possible execution path is indicative of a possible deadlock in the program which can be formulated as a rule and applied in the algorithm. Then a possible deadlock during the execution can be identified by the formula

$$\mathbf{A}(C_{cgd} \Rightarrow C_{g-})$$

where,

C_c: a conditional branching instruction $\in \pi$; C_g: a goto instruction $\in \pi$; C_{g-}: $c_{kg} \in \pi$ transfer control to its preceding conditional branching instruction.

The hex values of the machine codes together with their addresses can reveal all the above information. Whether the reported deadlock is a false positive or not can be decided by the programmer.

Table 4.7 A delay program and its erroneous version resulting in a deadlock

No deadlock			***Deadlock***		
delay	movlw	0xFF	delay	movlw	0xFF
w1	movwf	count	w1	movwf	count
loop	decfsz	count,1	loop	decfsz	count,1
	goto	loop		goto	w1

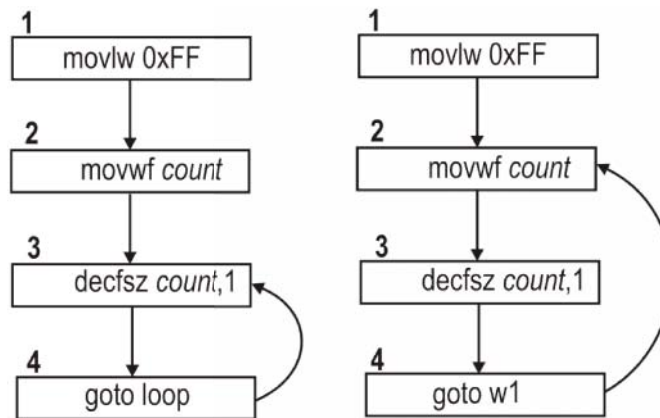


Fig. 4.5 The directed graph representations for the delay routines given in Table 4.7 where each node numbered in bold represents an instruction and arrows represent the control flow between instructions. (a) shows the digraph for the correct version and (b) shows the same for the incorrect version.

4.4.3 Error Correction

The proposed code validation technique is verified using programs typically run on microcontrollers like traffic signaling as well as Data Acquisition System. Since the causes of errors are implicit in the governing rules, suggestions for error correction can be reported along with the warnings in many cases. In addition to

this certain device specific corrections can be reported based on the feedback from the programmers. As a typical case the processor's data sheet illustrates how the *AD conversion time per bit* T_{AD} is related to the device clock period T_{OSC} with each *ADC* clock selection bit pattern *adcon0<ADCS1:ADCS0>*. Based on this, rules (8) and (9) of Table 4.4(c) are formed to correct any discrepancy in the configuration of *ADC* clock selection so that the minimum T_{AD} time of 1.6μs is assured. The programmer is asked to input the device clock frequency he is going to use and based on this following propositions are made.

C_{clk0}: the device clock frequency ≤ 1.25MHz

C_{clk1}: the device clock frequency ≤ 5MHz

C_{clk2}: the device clock frequency ≤ 20MHz

Hex code corresponding to the four instructions that clear or set *adcon0*<6> and *adcon0*<7> ∈ π defines the four propositions C_a, C_b, C_d and C_e respectively. Then any correction needed in the configuration can be informed by testing the following formulas.

$$\mathbf{A}(C_a \wedge C_d \Rightarrow C_{clk0})$$
$$\mathbf{A}(C_d \wedge C_b \Rightarrow C_{clk1})$$
$$\mathbf{A}(C_e \wedge C_a \Rightarrow C_{clk2})$$

By incorporating a decompiler/disassembler, the reported errors can be corrected either by eliminating, inserting or modifying the necessary code.

4.5 Results and Discussions

This work presents an architecture oriented code validation technique assisting the embedded system software designer in debugging, to make it more effective at early detection of errors. Commercial static analysis tools typically check for standard deficiencies which might miss bugs resulting from subtle

deviations of the hardware specification. But the technique presented here is based on rules of inferences formulated for the target processor that look into platform-specific properties. Integrating this offline debugging technique with the development environment can result in reduced debugging time and improved quality of the software through the elimination of logical mistakes as well as redundant codes made by the programmer. After the overhead of verification there is no hardware overhead for detection of such errors; nor any increase in the code size by this approach, as no signatures or labels are embedded into the program so also no run time overhead.

For the code analysis, the truth value of the propositional formulae is identified mainly using the machine code pattern and state spaces are created only for cases where the machine code is insufficient to evaluate the truth value of a proposition. This eliminates the need for an instruction simulator and gives a feasible solution to the state explosion problem in the state-of-the-art model checkers. In conventional model checkers the user specifies the requirements in the form of formulae and a parser converts those given by the user into the format needed by the local model checking algorithm. However in the work presented here, all the required formulae are in-built. A set of rules can be framed for a family of microcontrollers and a user can select a processor of his choice for validation. This technique can be easily extended not only to a wide range of RISC processors but also to other architectures.

Results of our analysis reveal that the technique is independent of assemblers/compilers. In the proposed approach, the compiler introduced redundancy can be identified since the machine code reveals the platform specific choices made by the compiler. It is common to select a processor based on its performance and to rely on the compiler to deliver this performance. This is particularly true of high-performance RISC devices. It is unacceptable to be limited

by available debugging technology. Though the technique described is general the implementation is highly architecture oriented and a case study on PIC16F87X series of microcontrollers is presented here. Since the analysis is performed on the low level language the range of types of analysis that can now be performed on the code is significantly reduced. To make the system interactive a disassembler/decompiler is to be incorporated. The coverage of faults depends on the rules governing the validation of codes.

4.6 Summary

An approach towards code validation of RISC based microcontrollers, at the level of machine instruction stream is described in this chapter. The code validation technique, the background that promoted this work and the feasibility of this approach in RISC microcontrollers are explained. Abstraction of CFG from Intel hex file and codification of rules are explained. The rules governing the occurrence of illegitimate/out of place instructions and code sequences for executing the computational and integrated peripheral functions of the target processor under consideration are discussed. The three phases that leads to code validation like fault localization, fault diagnosis and fault correction are considered. A prototype developed based on PIC16F87X microcontrollers and the results of analysis with sample programs are also presented. The chapter concludes with the discussions on the results.

5
CODE OPTIMIZATION

5.1. Motivation and Approach 123
5.2. Detection of Redundant Bank Switching Codes 126
- Relation Matrix Formulation
- Realization
- Tool Evaluation

5.3 Optimization Technique 141
- Variable Partitioning
- Optimum Memory Bank Allocation

5.4 Redundant I/O Port Configuration................................. 147
5.5 Redundant ADC Channel Selection.............................. 148
5.6 Software Realization 149
5.7 Summary.. 150

Optimization is a procedure that mainly seeks to maximize performance and minimize code size. As processor architectures have exponentially increased in complexity, the compiler optimization techniques [38, 39, 178] are continually advancing. However, no single optimization technique will work for every application. It is best to apply one optimization at a time, verify the results and measure any performance improvements before moving on. Optimization is an important task when developing resource intensive applications like embedded systems. Embedded systems are usually designed for a single or a specified set of tasks. Being specific the system design as well as its hardware/software development can be highly optimized.

Optimization by elimination of redundant codes is one among the optimization technique adopted by many software development tools. An architecture oriented approach towards the optimization, by the static analysis of embedded system machine codes resulting in the elimination of redundant code is described in this chapter. Many of the embedded system controllers use partitioned memory architecture. Memory banking and memory paging are common techniques used for microcontrollers, which increases the size of program and data memory without extending the address buses of the CPU. In these memory banks that cannot be accessed simultaneously, switching between them requires at least one bank selection instruction, which induce extra overhead in code size and execution time. The code size is a major factor rather than speed for the programs running in many embedded systems, since smaller code size often means less consumption of ROM as well as energy, and hence minimizing the number of bank selection instructions is an important research topic. Current compilers provide limited support to optimum generation of bank switching codes.

This work presents a code optimization technique to minimize the data and program memory bank switching instructions that assist a programmer in developing efficient embedded software. A relation matrix formed for the memory bank state transition corresponding to each bank selection instruction is used for the detection of redundant codes. It also proposes the optimum memory bank allocation to the variables in a program by the compiler that results in minimum number of bank switching codes. An algorithm is developed and utilized to detect the redundant memory bank switching instructions in the resulting machine codes from a compiler for different data allocation schemes of the application program. Then it selects the program with minimum bank switching instructions as the optimum solution. The basics of the algorithm developed, enhancements made to

the algorithm in order to suppress the false warnings and the results of the case study using Microchip's PIC16F877 microcontroller are presented.

The technique presented here achieves optimization of bank switching instructions without much computational burden by statically analyzing the machine code with a comparatively simple algorithm. Our algorithm can detect redundant data memory bank selection instructions that remain even after the application of optimization techniques by the compilers. For a program developed in assembly language also, the redundant bank switching instructions, especially for lengthy programs can be eliminated with this technique. With a well defined Control Flow Graph (CFG) constructed from the machine codes this algorithm fits well into large problem sizes as well. Redundant data and program memory bank selection instructions in the intraprocedural sequence, loops and interprocedural routines in the application program can be eliminated.

5.1. Motivation and Approach

For any memory space, larger the memory is, the larger the address bus needs to be. Previous efforts on partitioned memory are to enable memory access in parallel thereby increasing memory bandwidth and thus improving program performance. Such partitioned memory banks are found in processors like Motorola DSP 56000, Intel 8086, i80186 etc onwards. One way of avoiding large address buses is to divide the memory into a number of smaller blocks - called banks/ pages - each identical in size in most of the cases so that a smaller address bus can be used [35]. Smaller address buses result in smaller chip die size, higher clock frequencies and less power consumption. It can access all banks in an identical way, with just one of the banks being identified at any one time called the active memory bank (AMB) [17] as the target of the address specified. The contents of memory temporarily bank-switched out of the processors address space

are inaccessible to the processor. Many MCUs have banked memories that cannot be addressed simultaneously. For example, Freescale 68HC11 8-bit microcontrollers [179] allow multiple 64KB memory banks to be accessed by their 16-bit address registers with only one bank being active at a time. Bank switched SRAMs are employed with ultra-low-power sensors to achieve high code density [180] and allow the gating of individual memory banks [181]. Other examples include Intel 8051 processor family and MOS technology 6502 series microcontrollers. Certain modern microcontrollers use bank switching to manage read-write memory, non-volatile memory, input-output devices and system management registers. Most of the PIC microcontrollers [165] adopt a banked structure for their data as well as program memory of which a case study on PIC16F87X series of microcontrollers has been made in this work.

Unlike other memory management techniques, bank switching is nearly always initiated by the application program explicitly, although some real time operating systems take detailed control of the bank switching operation out of the application programmer's hands. A bank-sensitive program statement requires that the appropriate bank is to be made active prior to its execution. Otherwise, the program semantics are violated. This introduces an additional burden on the programmer; there is always a possibility for redundant bank switching instructions Thus, if data in one bank must be copied to another bank, bank selection instructions are always necessary. Obviously, placing all the variables accessed by a function in the same memory bank will reduce the number of bank selection instructions and the total required cycles for the application. However, conventional compilers have no way of knowing which functions call which variables and are therefore unable to optimize their memory assignment. Nor do these compilers have any way of knowing whether or not a particular memory bank will be selected at any point in the code. As a result, these compilers automatically

generate bank selection instructions for every memory access, whether or not that bank is already selected, unnecessarily bloating the code - often increasing the code space requirement. Compiler vendors have addressed this issue by providing bank qualifiers - extensions to the C-code. This allows the compiler to see the exact bank an object resides in and reduces the number of bank selection instructions for more compact code. However, trying to track all the memory addresses across multiple code modules and ensuring all pointers to have the appropriate qualifiers is a time consuming and tedious process. This requires substantial expertise as well as run the risk of introducing programming errors [8]. All the related works [17, 31, 32, 33, 34] are analyzing the source programs for minimal placement of bank switching instructions.

Analysis of a high level program cannot easily determine the current bank state. But with a static analysis of the machine code, the state transitions at each bank switching instruction can be easily determined. Static analysis examines the code of programs to determine properties of the dynamic execution of these programs without running them. This technique has been used extensively in the past by compiler developers to carry out various analysis and transformations aiming at optimizing the code [40, 96]. Today's compilers cannot efficiently exploit the architectural features of advanced embedded processors. This results in the introduction of redundant codes in their output. This work presents an algorithm developed, to detect the redundant bank switching codes in the machine code generated by the compiler. So the compilers can insert bank selection instructions for every memory access in the conventional way and the output file in the Intel Hex file format is tested with the algorithm developed to detect all the redundant bank switching code. So the compiler is deprived of any complicated analysis needed during compilation to minimize the bank switching code as done by some advanced compilers like HI-TECH OCG (Omniscient Code

Generation)[8]. Now appropriate allocation of data variables to the available memory banks can again increase the redundant bank switching codes detected by the algorithm developed, resulting in minimum number of such codes in a given application program. To the best of the authors' knowledge only [34] presents a data partition technique aimed at minimal placement of bank selection instruction resulting in code as well as runtime saving. Our non profile-guided compiler method is static and is independent of the compiler but implementation of algorithm depends on certain architectural features of the target processor. The optimization of the code is done again following certain algorithms. Some of the instructions inside a loop will be executed unnecessarily which can be eliminated by code motion.

5.2. Detection of Redundant Bank Switching Codes

The goal of our optimization is to eliminate the *redundant* bank selection instructions in a program while ensuring that the banked memory is accessed correctly. The detection of redundant bank switching code is done with the help of a relation matrix derived from the architectural features of the target processor like number of memory banks and instruction set (memory bank switching codes). Though the implementation depends on the target processor the formation of the relation matrix can be generalized as explained below. The feasibility of the approach has been verified on systems based on PIC16F87X series of microcontrollers.

A detailed study on the various PIC families of microcontrollers has been made in this regard. PIC 16F84A have just two banks [35] and the address of either bank is the 7-bit RAM address. The active bank is selected by bit 5 in the Status register. The programmer must ensure that the bank bit in the Status register is correctly set before making any access to memory. The data memory in

PIC16F87X *devices* is partitioned into four banks of 128 Bytes each, which contain the general purpose registers and the Special Function Registers (*SFR)*. For selecting a particular bank, bits *RP1;bit six of status register* (*status<6>*) and *RP0* (*status<5>*) are to be configured appropriately. All PIC16F87X devices are capable of addressing a continuous 8K word block of program memory. The *call* and *goto* instructions provide only 11 bits of address to allow branching within any 2K program memory page. While doing a program branching with *call* or *goto* instruction the upper two bits of the address are provided by *pclath*<4:3>. When doing such branching, the user must ensure that the upper page select bits are programmed so that the desired program memory page is addressed.

The data memory space in PIC18F series devices is divided into as many as 16 banks that contain 256 bytes each [182]. The Bank Select Register, *BSR*<BSR3:BSR0> holds the four bit bank; the instruction itself includes the 8 Least Significant Bits, which can be thought of as an offset from the bank's lower boundary. The BSR can be loaded directly by using the *movlb* instruction.

In general, the address space is partitioned into memory banks and the CPU can access one bank at a time, which is called the active memory bank (AMB), using bank selection bits or bank selection instruction. For implementing this code optimization through static analysis of machine code, the memory bank that was active just before the execution of a bank switching instruction is named as Previously Activated Memory Bank (PAMB). ***A bank switching/ selection instruction is said to be redundant when the execution of such an instruction switches the memory bank to an Active Memory Bank (AMB) that does not alter the PAMB.***

Based on the study on the various PIC families of microcontrollers following generalizations are made for the partitioned data memory architecture. If

P is the number of memory banks, so that $2^{\mathbf{r}} = \mathbf{P}$, then the number of bits that decides the bank selection in the bank selection register will be **r**. The number of machine codes controlling the bank selection will be **P** if the bank register is loaded with a *mov* instruction. For each PAMB state there will be one bank selection instruction, which is redundant. If *bitset* and *bitclear* instructions on the *BSR* are used for bank switching there will be 2**r** number of machine codes for this operation and for each PAMB state there will be **r** number of bank switching instructions, which are redundant.

5.2.1 Relation Matrix Formulation

The family of Microchip PICmicro MCUs constitutes a RISC-based Harvard architecture with instruction size of 14 bits and data width of 8 bits [165]. The data memory banks in these embedded controllers contain the General purpose Registers and Special Function Registers. For proper functioning of the device, proper configuring of these registers is essential. Since these registers are spread across different banks they are to be accessed through the bank switching instructions, which limits the data partitioning optimization for hardware dependent code. In case of program memory paging when a branching instruction that crosses the page boundary is made, the programmer should ensure the required page switching before these instructions. The disadvantage of bank-switched architectures is the code size and runtime overhead caused by bank selection instructions.

Instead of having a single bank selection instruction, the PIC16F87X architecture provides only bit access to the bank selection register, which is the *status* register. The assembly instructions that clear or set the bits *RP0* and *RP1*of the status register are *bcf status, RP0; bcf status, RP1; bsf status, RP0*; and *bsf status, RP1* and are represented by the symbols *a*, *b*, *c* and *d* respectively. The hex

codes corresponding to these instructions are 1283h, 1303h, 1683h and 1703h respectively. The four data memory banks are named B0, B1, B2 and B3. On a power on reset, the default bank that is active is bank'0' represented as B0. Depending on the PAMB state, the AMB state occurs with each bank switching instruction. The assembly instructions that set or clear the bits RP0 and RP1of the status register, corresponding machine code sequence, and their symbols used in the state transition diagram are shown in Table 5.1.

Table 5.1 Bank switching instructions and their symbols

Mnemonics	Machine Code in Hex	Symbol
bcf status,RP0	1283	a
bcf status,RP1	1303	b
bsf status,RP0	1683	c
bsf status,RP1	1703	d

The Active Memory Bank is a discrete function [169] of Previously Activated Memory Bank (PAMB) and bank switching instruction. Let the finite sets

B = {B0, B1, B2, B3} represents the symbols of PAMB states and

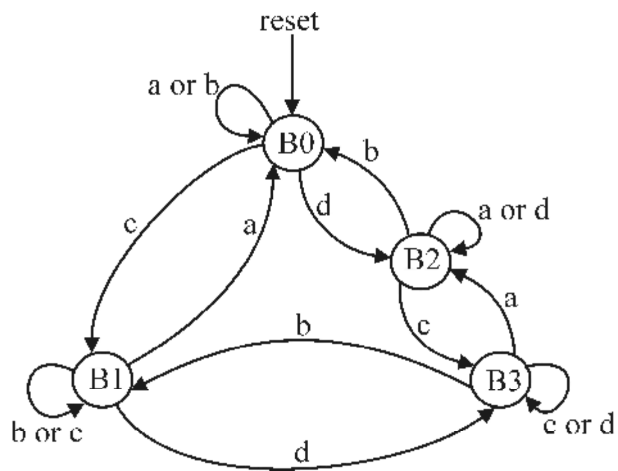
I = {a, b, c, d} represents the symbols of bank switching instructions respectively.

∂ is a mapping of B×I →B which denotes the next-state function.

Then Δ, a (2^r×2r) relation matrix, can be obtained by first constructing a table whose columns are preceded by a column consisting of successive elements of B and whose rows are headed by a row consisting of the successive elements of I as shown in Table 5.2. The relation matrix Δ is obtained as

$$\Delta = \begin{bmatrix} B0 & B0 & B1 & B2 \\ B0 & B1 & B1 & B3 \\ B2 & B0 & B3 & B2 \\ B2 & B1 & B3 & B3 \end{bmatrix}$$

Previously Activated Memory Bank	Active Memory Bank with Bank Switching Instructions			
	a	b	c	d
B0	B0	B0	B1	B2
B1	B0	B1	B1	B3
B2	B2	B0	B3	B2
B3	B2	B1	B3	B3



incorporating the necessary algorithm. Eliminating such instructions from a machine code sequence results in a code optimized for space and speed metric.

For the target processor considered, most of the time the compiler/macros/user places two instructions to select the required data memory bank. They are

(*bcf status, RP0* $\vee$ *bsf status, RP0*) $\wedge$ (*bcf status, RP1* $\vee$ *bsf status, RP1*).

i.e. $(a \vee c) \wedge (b \vee d)$

To select bank B3 (i.e. *status*<RP1:RP0> = b'11'), the two probable instructions are *bsf status, RP0* (c) and *bsf status, RP1* (d). With a PAMB state B2 (i.e. *status*<RP1:RP0> = b'10'), only instruction c is needed and instruction d is redundant since; ∂ (B2, c) = B3 which is evident from the matrix Δ as well as the state transition diagram. This redundancy corresponds to a loop in the state transition diagram which the algorithm identifies and that instruction is eliminated. Even though the order of the instructions is reversed, the algorithm identifies the first instruction as redundant since the state transition is to B2 itself or ∂ (B2, d) = B2. The other situation is selecting a bank which is already the active bank. For selecting the bank say B1, the two probable instructions are *bsf status, RP0* (c) and *bcf status RP1* (b). With a PAMB state B1;

∂ (B1, c) = B1 and ∂ (B1, b) = B1

The algorithm identifies both the bank selecting instructions which are redundant as evident from the matrix Δ as well as the state transition diagram and can be removed. The relation matrix is independent of the application program, but it depends on the architectural features of the target processor. If **P** is the number of memory banks, so that $2^r = \mathbf{P}$, then the number of rows of the relation matrix will be **P.** If the bank switching is done with a data transfer instruction then the number

of columns of the relation matrix also will be **P** and in case the bank switching is done with individual bit set/reset instructions the number of columns will be 2**r**. With an identical approach the redundant program memory page switching instructions (which are *bcf pclath*, 3; *bcf pclath*, 4; *bsf pclath*, 3; *bsf pclath*, 4) to switch the four pages in the program memory also can be eliminated. Hence almost all possible redundancy introduced in the program with respect to the bank selection instructions are identified and can be eliminated.

5.2.2 Realization

A novel algorithm to assist software developers for eliminating redundant data and program memory bank selection instructions has been developed; this also helps developing efficient embedded software utilizing static analysis of machine codes.

The relation matrix Δ, formed for the AMB state transition, corresponding to each bank switching instruction in the machine code sequence of an application program, is used for eliminating the redundancy. For the implementation of the code optimization the machine code is read from the *Intel hex* file and stored in an array. Checking of redundant bank switching instructions should follow the sequence of instructions executed by the processor which correspond to a path in the program graph. In order to get the correct sequencing of instructions, the program (machine code) is partitioned into blocks of instructions by disconnecting from every merge node (a node in the program graph with more than one incoming arc) all of its incoming arcs [167]. Hence the program graph is partitioned into a collection of disconnected subgraphs where each subgraph corresponds to a set of instructions or subprogram. Since each subgraph is a tree, they have only one entry point (root node) and there is a unique path, and hence a unique sequence of instructions, from entry point to each of the exit points. Now the CFG can be

constructed where each subgraph of the program graph is represented as a single node and the arcs represent valid control flow between subgraphs [38, 150]. From the CFG the set of elementary paths in a subprogram are identified in the same way as described in section 4.1.3 of chapter 4.

The flow chart given in Fig. 5.2 explains the algorithm to detect the redundant bank switching codes. N represents the total number of CFG nodes. M represents the number of paths in the nth subgraph. L represents the number of machine codes (nodes)

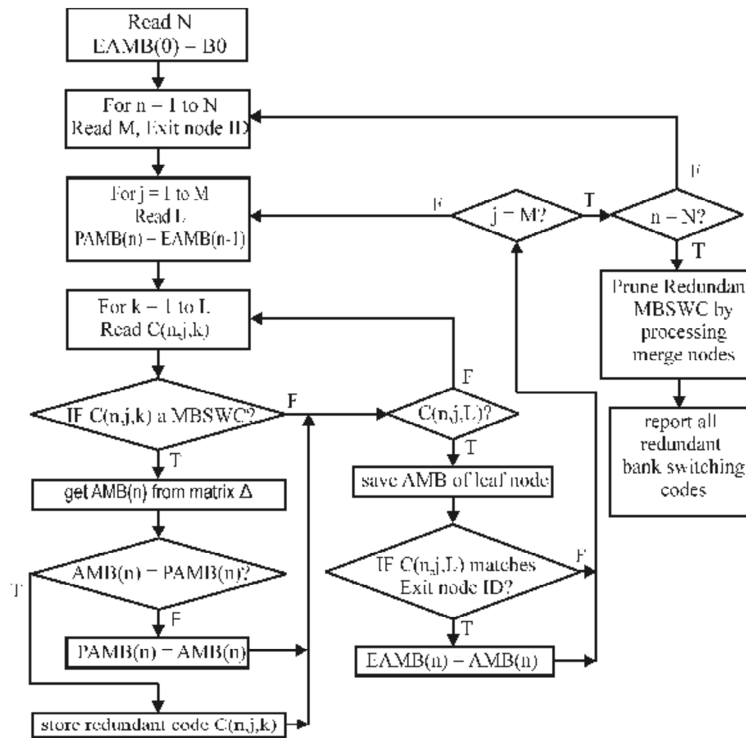


Fig. 5.2 Flowchart explains the identification and pruning of redundant MBSWC in the machine code sequence of a program.

in the jth path of the nth subgraph. C(n,j,k) represents the kth machine code in the jth path of the nth subprogram. Since the bank B0 is the default active bank on *reset*, B0 is assigned at start to the PAMB of each path of the 1st CFG node. For each memory bank Switching Code (MBSWC) in a valid path the AMB state is obtained from the matrix Δ. A redundant MBSWC is located when AMB = PAMB. The AMB associated with the v_{if}, which is given an Exit ID is assigned to the Exit Active Memory Bank (EAMB) which becomes the starting PAMB of each path of the next CFG node. For the analysis of a subprogram a linear scan is sufficient. Analysis of a subprogram takes care of the redundancy of the memory bank switching instructions associated with the intraprocedural routines in an application program.

Analysis of the last CFG node is followed by the processing of the merge nodes. The initial node of a subprogram is a merge node where there is more than one incoming edge. So the first MBSWC in each path of a subprogram cannot be eliminated just by observing it to be redundant from the EAMB state of its previous subprogram. Hence the AMB associated with each of the incoming arc at the merge node are to be considered. Each of these incoming edges corresponds to a source node which is nothing but a leaf node, and hence an AMB is associated with it. Hence the AMB at the entry node of a subprogram need not be unique. A typical case is that of a function call from different call sites. A call site corresponds to a node which contains an instruction implementing a function call. All the call sites need not have the same AMB state. A loop in a program is another case. Therefore, for all the CFG nodes, even though the first (pair of) bank switching instruction in any path is found to be redundant, they are not reported till the interprocedural analysis is over and the redundancy is confirmed. This is the first step done towards the suppression of false warnings. Hence the AMB associated with the first

instruction in a subprogram is taken as the union of AMBs of its incoming arcs. During processing of the merge nodes, if it is found that all the incoming edges to a merge node are having the same AMB associated with them, then the redundancy marked for the first (pair of) bank selection instruction in that node or in any path of that subprogram is considered to be redundant and can be eliminated. When it is not so a decision is made by considering the AMB combinations of the incoming edges as follows.

If B0 and B2 only then the instruction *bcf status,RP0* is redundant

If B0 and B1 only then the instruction *bcf status,RP1* is redundant

If B1 and B3 only then the instruction *bsf status,RP0* is redundant

If B2 and B3 only then the instruction *bsf status,RP1* is redundant

As a second step towards suppression of the false warnings, the algorithm considers all the transparent nodes which do not contain any bank switching instructions. If the active memory bank associated with the incoming edges of a transparent node are not equal then the leaf nodes of this subprogram are assigned with the combination of incoming edges' AMBs. Again within a CFG node if any of the paths is without a bank switching instruction its leaf node is treated similarly. When the initially detected redundant codes are pruned the AMBs associated with all the incoming edges to the entry node are taken care of. Hence the algorithm takes care of the redundant data memory bank selection instructions associated with all the loops and interprocedural routines of the application program.

5.2.3 Tool Evaluation

The code analyzer developed for the detection of redundant bank switching instructions in an application program is realized in software using Visual Basic. The tool is evaluated using programs typically run on microcontrollers. For programs developed in assembler the necessary pair of MBSW instructions were inserted prior to all bank sensitive instructions and tested. Fig. 5.3 shows the CFG of a sample program used for the analysis which has got six nodes '*n1*' through '*n6*'. Each bank sensitive instruction in the program is preceded by an appropriate pair of MBSWC. Each node in a program graph is assigned with an address and its associated machine code. The hex values of the addresses corresponding to the pair of MBSWC are shown encircled and the resulting active memory banks such as B0, B1 etc. are also shown. B0 results with the instructions $a \wedge b$, B1 results with the instructions $c \wedge b$, B2 results with the instructions $a \wedge d$ and B3 results with the instructions $c \wedge d$. The AMB associated with the incoming edges of '*n1*' through '*n6*' are also shown. With the MC_CODE ANALYZER v1.02 only the inraprocedural analysis has been done. Here the analysis of each CFG node considers the EAMB associated with the exit node of its predecessor only. Results of the analysis for the sample program above with MC_CODE ANALYZER v1.02 are given in Fig. 5.4 which show all the redundant bank switching codes associated with all the intraprocedural routines along with their address locations. The source node address, the machine code at this address location and the destination node address of the program graph are also displayed in the screenshot. The addresses of these redundant codes are single starred or double starred in the Fig. 5.3, the latter being the first (pair of) MBSWC in the subprogram.

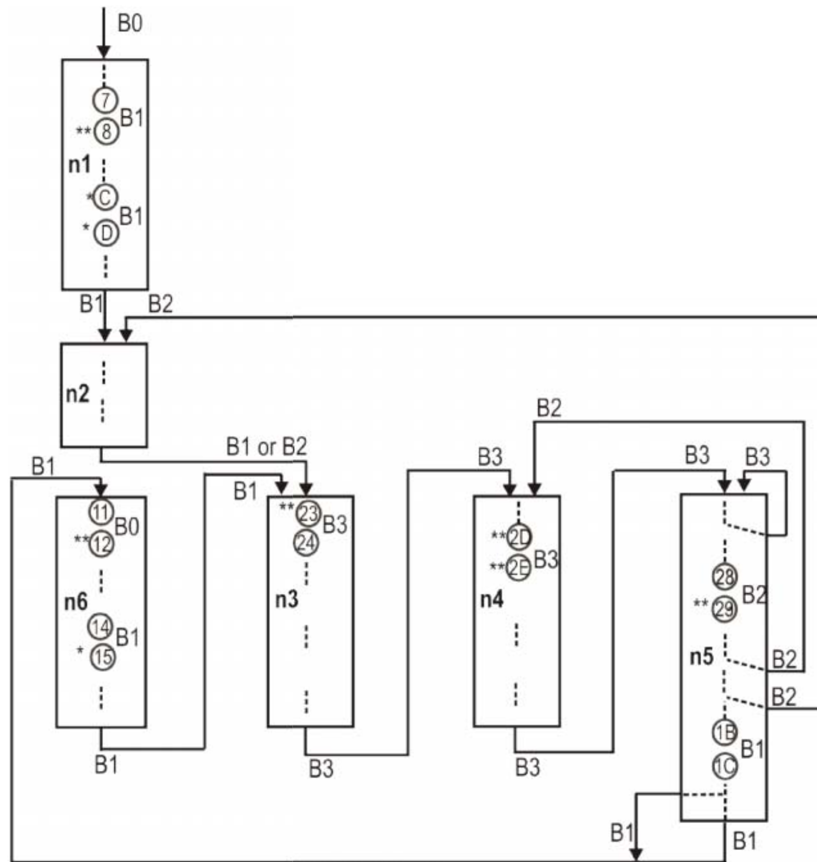


Fig. 5.3 CFG of the sample program for the analysis.

With the MC_CODE ANALYZER v3.00 the inraprocedural, interprocedural and transparent node analysis has been conducted. The first (pair of) redundant bank switching code/codes in any of the subprogram (the nodes which are marked **), already identified with the MC_CODE ANALYZER v1.02 are pruned with this analysis to avoid any false warnings. Here the first/first pair of bank switching code/codes of each subprogram which were found redundant by the previous analysis are reported to the programmer only if they are found redundant with the interprocedural analysis too.

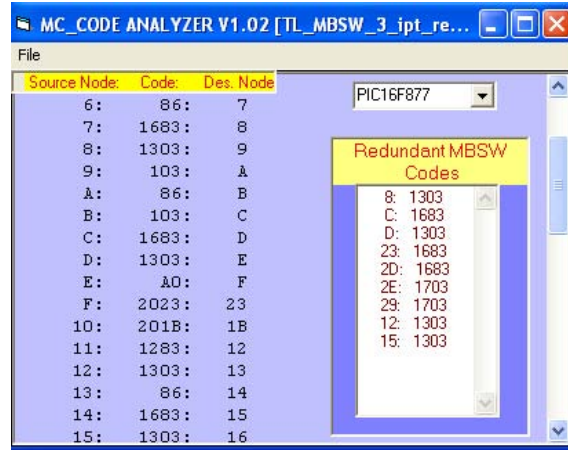


Fig. 5.4 Screen shot of the developed MC_CODE ANALYZER v1.02 for the sample program.

Screenshot explaining the results of this analysis for the same sample program with the MC_CODE ANALYZER v3.00 are given in Fig. 5.5. The machine codes at addresses 8h, 23h, 2Dh, 2Eh, 29h and 12h are pruned as follows. The redundancy reported in the first analysis for the code at location 23h is eliminated in the second analysis since '*n2*'is a transparent node and therefore the leaf nodes of this subprogram are assigned with the combination of incoming edges' AMBs. Then the

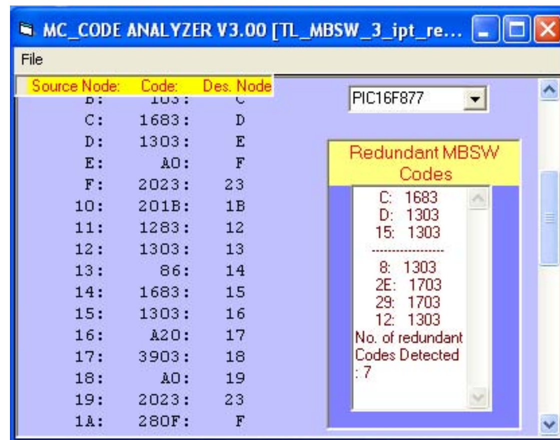


Figure 5.5 Screen shot of the developed MC_CODE ANALYZER v3.00 for the sample program.

incoming edges of node '*n3*' can have active memory banks either B1 or B2. With a PAMB of B1, the instruction '*c*' is redundant since ∂ (B1, *c*) = B1, but with a PAMB of B2, the instruction '*c*' is not redundant as evident from the state diagram; hence the code at location 23h is eliminated from the result. Similarly for the node '*n4*', codes at 2Dh and 2Eh are reported redundant in the first analysis since EAMB of the exit node of '*n3*' is B3. But with the second analysis only code at location 2Eh is reported and 2Dh is eliminated since the incoming edges AMB combination is B3 and B2 only. With a PAMB of B3 or B2 the instruction '*c*' is not redundant, but the instruction'*d*' is redundant since ∂ (B3, *d*) = B3 and ∂ (B2, *d*) = B2. For the node '*n5*', since the incoming edges are having the same AMB B3, code at location 29h is reported in both the analysis which is clear from the relation matrix. For the machine codes at addresses 8h and 12h no change since node '*n1*' is having only one incoming edge and for 'n6' the incoming edges are having the same AMB which is B1. The codes which are found redundant in the first analysis but eliminated later lead to the suppression of false warnings.

Results of the analysis done on machine codes generated with different compilers as well as assembler are given in Table 5.3. HI-TECH Software is a world-class provider of development tools for embedded systems and is the number one third party vendor of compilers for Microchip Technology Inc. For a program module 'delay_time_rout' downloaded from [183] and compiled using HI-TECH C PRO, the algorithm detected six redundant codes. Sample programs available with HI-TECH C PRO compiler are tested and the results are given as sl. no. 2 to 6. These programs are compiled with the optimization enabled; hence the results prove that the tool developed is superior to the compiler. Serial numbers 7 to 12 gives the results of the analysis on programs available with *PROTEUS VSM* design tool. The results of the analysis for an ADC program compiled using HI-TECH C PRO, mikroC and also the same program developed in assembler are also included

(sl. No. 13 to 15) to test the independence of the tool developed on the compiler. Serial number 12 is a program compiled with *PICBASIC*.

For a traffic signaling program developed in assembler with each bank sensitive instruction preceded by a pair of necessary bank switching instructions, the algorithm detected all the redundant bank switching codes and this is presented as sl. no.16 of the table.

Table 5.3 Results of the analysis

Sl.No.	Program	Code size	MBSWC present	Redundant MBSWC detected.	% Saving in Code Size
1	delay_time_rout	223	6	6	2.7
2	Lcd_demo	176	12	10	5.7
3	Timer_demo	49	3	0	0
4	Intr_demo	44	2	0	0
5	Pic_demo	700	16	14	2
6	Bootloader	225	19	1	0.44
7	ADC	63	7	1	1.6
8	Doorbell	643	2	0	0
9	PICCLOCK	292	2	0	0
10	RS232LCD	102	5	1	0.98
11	GEPE456	1403	10	2	0.14
12	GLCD_T~1	1044	16	0	0
13	HiTecC_ADC	84	18	8	9.5
14	mikroC_ADC	56	10	2	3.6
15	ASM_ADC	81	9	1	1.2
16	Traffic_signalling	48	16	7	14.6

The tool developed counts the total number of bank switching codes originally present in the program as well as the number of redundant bank switching codes. Using the simulation log in *PROTEUS VSM* the number of program words in each program is also found. Hence the percentage saving in code size is computed and presented in the table. A corresponding saving in run time can also be computed. Including the profile data can give the execution frequency of each node so that the better approximation of the runtime saving can be computed which will be conducted as a future work. The same application program can be compiled using different compilers and the machine codes from each of these compilers can be tested with the tool developed. Now by counting the number of redundant codes reported in each of these cases, an evaluation of the different compiler's optimization capability in this regard is possible.

5.3 Optimization Technique

This work considers a compiler strategy of allocating **z** number of data variables in an application program to **P** number of data memory banks in the target processor, with the objective to deliver the machine code with minimum number of bank switching codes. Since the number of bank switching codes cannot be expressed as a linear function of the data variable, an ILP solver is not applied in our approach.

5.3.1 Variable Partitioning

For a banked memory with **P** banks each of equal size, **z** number of data variables can be assigned to the available banks in $\mathbf{P}^{\mathbf{z}}$ possible ways provided $\mathbf{z} \leq$ bank size. If the banks are of unequal size the case reduces to the same, provided $\mathbf{z} \leq$ smallest size of the banks. When $\mathbf{z}>$bank size the data mapping can be considered as the problem of finding all possible $\mathbf{z}\times\mathbf{P}$ integer matrices [184, 185, 186] $\boldsymbol{A}$ with

$a_{ij} \in \{0,1\}$, that satisfies the given constraints on its rows and columns. The cardinality of the set of such data mapping matrices depends on these constraints. The first constraint is that, every data variable is considered as a single unit and is allocated to only one memory bank:

$$(\forall i): 1 \leq i \leq \mathbf{z} : \qquad \sum_{j=1}^{\mathbf{P}} a_{ij} = 1$$

Second constraint is that the sum of the sizes of all variables allocated to a particular memory bank $\mathbf{B}_j$ must not exceed the size of that memory bank $m(\mathbf{B}_j)$:

$$(\forall j): 1 \leq j \leq \mathbf{P} : \qquad \sum_{i=1}^{\mathbf{z}} a_{ij} \leq m(\mathbf{B_j})$$

Third constraint is that **z** must not exceed the sum of sizes of all banks:

$$\mathbf{z} \qquad \sum_{j=1}^{\mathbf{P}} m\left(\mathbf{B_j}\right)$$

The polynomial-time solvability of this case has been proved [187]. Indeed, more constraints may decrease the runtime by decreasing the space of feasible solutions. For example six variables can be allocated to two memory banks in 2^6 (64) ways provided each bank size ≥ 6. But with the constraint of bank size=3, the feasible number of data mapping matrices (cardinality of the set of matrices) reduce to 20.

The set of data mapping matrices can be obtained with a depth first search algorithm. Adding one more row and column to an $z \times \mathbf{P}$ matrix subject to the following constraints gives the matrices.

$(\forall j) = 1$ to P

$a_{(\mathbf{z}+1),j} = m(B_j)$

$(\forall i): 1 \leq i \leq \mathbf{z}$

$a_{i,(\mathbf{P}+1)} = 1$

So without any HLL directives the compiler can try all possible combination of data variable allocation. Prior to all bank sensitive instructions the compiler can insert as many bank switching instructions as needed. The resulting machine codes are tested with the algorithm developed to detect the redundant bank switching codes. The program that results in the maximum number of redundant bank switching code corresponds to the minimum number of bank switching codes in the program and can be selected as the optimum data allocation scheme for a given application.

5.3.2 Optimum Memory Bank Allocation

The compiler designers and MCU manufacturers suggest certain tips for speed optimization. In processors using banked memory architecture, the bank switching instructions can be reduced by properly selecting the order in which the variables are initialized at the start up of a program. They also suggest using variables in same bank in arithmetic expressions, to avoid bank switching. Another suggestion is that the variables accessed most often in the program can be allocated to the memory spaces that are cheapest to access. A careful assignment of program variables to registers is the most important optimization of a compiler for RISC.

For a given application program, the data variables can be allocated to the available memory banks by considering all possible permutation of memory banks and combination of data as represented by the set of data mapping matrices explained in section 5.3.1. In each of these programs corresponding to the various data allocation schemes, the compiler puts the necessary MBSWC prior to all bank sensitive instructions without applying any algorithm for the minimal placement of bank switching codes. This results in a unique Intel hex file output corresponding to each of these programs. These files become the input to the machine code analyzer developed which detects the number of redundant bank switching

instructions present. The more the reported number of redundant codes, optimum is the memory bank assignment. So the number of eliminated code is compared each time and the most efficient code is selected.

We now discuss an example to illustrate how the approach described above works in practice. For the target processor under study there are four memory banks. So **z** number of data variables can be assigned to the 4 memory banks in $4^{\mathbf{z}}$ ways when **z**≤bank size. For testing this tool for optimum data allocation a traffic signaling program having three data variables is considered. The three data variables are named ***S***, ***T*** and ***U*** and are assigned to the four banks in 4^3 (64) ways resulting in 64 programs each with a unique data allocation scheme. In these programs the three data variables ***S***, ***T*** and ***U*** can be placed in the four memory banks available, first by placing the entire three in one bank, second by placing the three data in any of the two banks and third in any of the three banks out of the four available. Considering the permutation of memory banks and the combination of data in each of the above cases , programs one to four are with all the three data allocated to any one of the banks so that there are $4P_1= 4$ ways of data allocation; programs five to forty are selecting any of the two banks at a time, so that for the three variables there are $4P_2\times3C_2= 36$ ways of allocating the data and programs forty one to sixty four are selecting any of the three banks for the three variables in $4P_3\times3C_3 = 24$ ways.

For the target processor since the special function registers are implemented in data memory bank, accessing these registers must ensure the proper bank switching. The *SFRs* used in the program considered are *trisb* and *portB*. Each bank sensitive instruction in the program is made preceded by a pair of necessary bank switching instructions. There are eight number of bank sensitive instructions so that the number of bank switching instructions altogether in the program is

sixteen. Fig. 5.6 shows the CFG of this program in which the data variables ***S*** and ***U*** are allocated to B3 and ***T*** is allocated to B0. This results in the worst case allocation where the number of redundant codes identified is two which are starred in the figure.

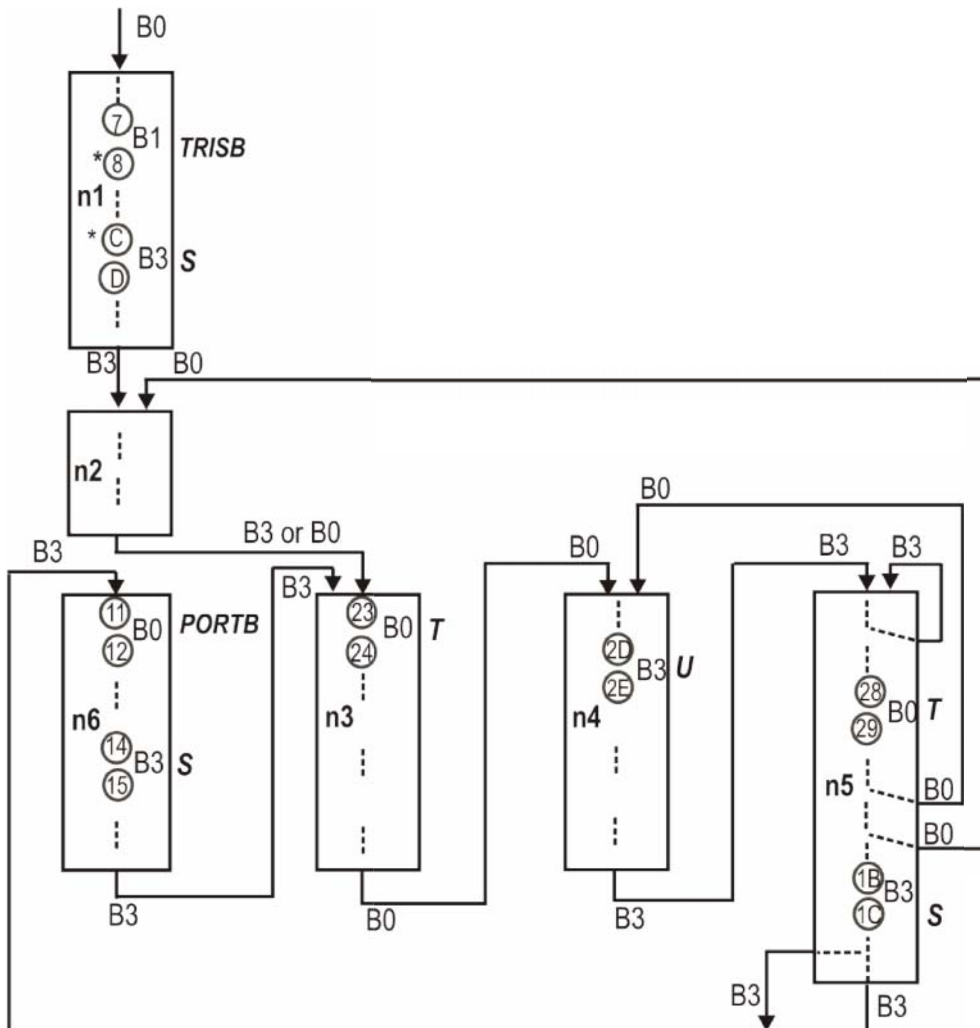
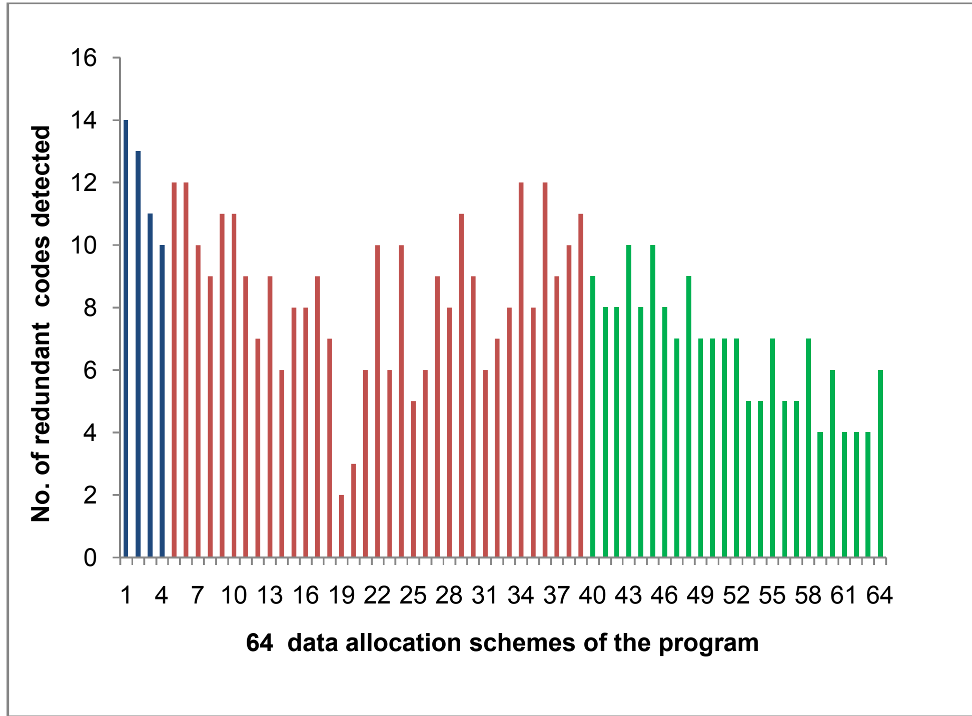


Fig. 5.6 CFG of the sample program with the worst case data allocation scheme.

Fig. 5.7 shows the bar graph for the number of redundant bank switching instructions reported in the 64 data allocation schemes of the program considered. The first four cases are with all the three variables ***S***, ***T*** and ***U*** in one bank. Programs five to forty are with the data variables ***S***, ***T*** and ***U*** assigned to any of the two memory banks. Similarly programs forty one to sixty four are with data assigned to any of the three banks out of the available four. The worst case reported is when ***S***, in B3, ***T***, in B0 and ***U*** also in B3 (sl.no.19 in bar graph) where out of the sixteen bank switching instructions only two are redundant. The optimum data assignment is with ***S***, ***T*** and ***U*** assigned to B0 (sl.no.1 in bar graph) where fourteen out of the sixteen are redundant. The total number of bank switching instruction depends also on the use of special function registers in a program which are implemented in these memory banks. Data allocation schemes 5, 6, 34 and 36 in the bar graph give the indication that there is a tendency for optimum data assignment even though all the data are not in B0. Distributing the data allocation to two banks in these cases is more efficient than allocating all the data to B2 or B3.

From the results the conclusion obtained is that a compiler can insert the required bank switching instructions prior to any bank sensitive instruction without any complicated analysis on the source code. The compiler can attempt all possible data allocation schemes for a given application program. Using this tool it can determine all the bank switching code to be eliminated along with the optimum data allocation to the available banks. When the reported redundant codes are eliminated, the program runs successfully.



No. of redundant codes detected
16
14
12
10
8
6
4
2
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64
64 data allocation schemes of the program

These redundant codes can be identified and reported through the static analysis of machine code developed. If a processor has got 'M' number of I/O port pins then 'M' number of port_pin_flags are used in the analysis. A flag is set if the corresponding I/O port bit is configured as input in the program code sequence and reset otherwise. Initialization of each of these flags is according to the power on reset condition of that processor. Using the relation matrix Δ described in section 5.2.1 of this chapter and the bank switching codes in the given program the use of all registers as well as ports in the executables can be identified. A linear scan through the successive nodes in the CFG of an application program can reveal the existence of a code, the execution of which results in the same pin configuration for a port as that of the previous one. The resulting codes can be pruned with the interprocedural analysis where the statuses of the port_pin_flags of all the incoming edges of a merge node are taken into account.

5.5 Redundant ADC Channel Selection

Programs spend most of their time going around loops. Loops therefore are the most promising sources to attempt speedups in a program, and it behoves an optimizing compiler to generate particularly efficient code for loops. In embedded systems a real time program will be executed in an infinite loop. Though the control flow trace may indicate a number of repeated paths, the possible execution paths are fixed. Analysis is to be conducted through the possible execution paths only. The paths taken by the program can be obtained from the possible combinations of control flow edges.

Some of the instructions inside a loop will be executed unnecessarily. A typical case is that of selecting the *ADC* channel which should be done only once unless a new channel is to be selected. So channel selection instruction is to be outside the loop. Evaluation of rule (4) of Table 4.4(c) included in section 4.4.1 of

chapter 4 achieves this result. Similar instances can be identified and the programmer can be instructed to revise the program for optimization.

5.6 Software Realization

Algorithms for the detection of redundant codes in an application program are realized in software using Visual Basic. The block diagram given in Fig. 5.8 explains the various steps in the static analysis of the machine code developed. The

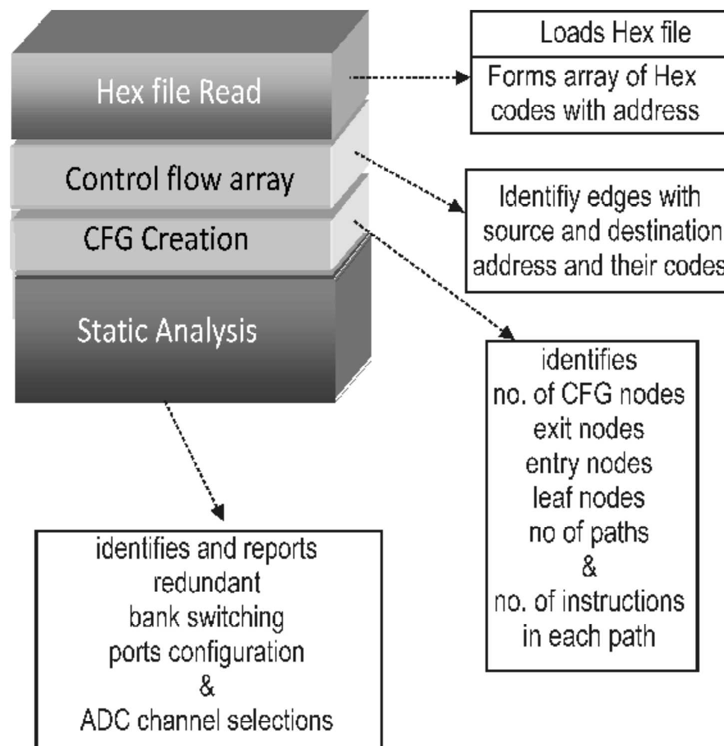


Fig. 5.8 Various steps realized in software for the code optimization.

selected Intel Hex file of the application program is read and an array of machine codes are stored with their address values. For generating the program graph a

control flow array is formed which identifies the edges in the program graph with the source and destination addresses and the corresponding codes. The CFG is constructed by eliminating the incoming edges of the merge nodes, thereby identifying the entry nodes, exit nodes and leaf nodes. Then the number of paths in each CFG node and the codes in each path are identified. Now the algorithm is applied to identify the redundant codes which are reported to the developer.

5.7 Summary.

This chapter describes the optimization techniques and their realization, for the static machine code analyzer. This tool helps to eliminate the redundant codes in bank switching instructions in partitioned memory architectures, in port configuration as well as in ADC channel selection. Formation of a relation matrix with PAMB and bank switching instructions is illustrated. A state transition diagram representing the data memory bank switching with the execution of each bank switching instruction to the corresponding AMB state is presented. A data allocation technique to minimize the bank switching is described. Various steps in the software realization of the machine code analyzer are discussed. Elimination of redundant codes in the intraprocedural sequence, loops and interprocedural routines in the application program are illustrated. A prototype based on PIC16F87X microcontrollers is described and the experimental results obtained with sample programs are also presented.

6

CONCLUSIONS

6.1Contributions 151
6.2 Highlights of the Work 153
6.3 Merits and Demerits 154
6.4 New Research Directions 158
6.5 Summary 158

This chapter consolidates the algorithms, features, contributions, results and applications of the validation, error localization and optimization techniques developed. The enhancement methods for the improved performance of the tool developed are proposed. The extension of this work to other applications and possible future research are also examined.

6.1 Contributions

This research work is concerned with the static analysis of machine code for embedded system software debugging. An approach towards architecture oriented code validation and optimization with respect to RISC microcontrollers, at the level of machine instruction stream, to make it more effective at revealing errors as well as redundancy is described. The major contributions of this thesis are as follows.

- Proposes an architecture oriented code validation technique assisting the embedded system software designer in debugging, on a host machine, to make the debugging more effective at early detection of errors.
- Prescribes more than one hundred governing rules of inferences to validate the code and hence to make code inspection automatic.
- Codifies the formulated rules in proportional logic formulae for testing their compliance.
- Proposes a relation matrix assisting the validating tool to identify the active memory bank state of each code in the instruction stream.
- Developed an algorithm to eliminate the redundant memory bank switching instructions in embedded system software.
- Developed a static tool to analyze the intraprocedural sequence, loops and interprocedural routines in an application program.
- Proposes an architecture oriented code optimization technique through static analysis of machine code.
- Proposes a technique for optimum data allocation to banked memory resulting in minimum number of bank switching code.
- Pioneered a useful tool in steering novices towards correct use of difficult microcontroller features in developing embedded systems.
- Identifies program states mainly with machine code pattern reducing the state space creation contributing to an improved state-of-the-art model checking.
- Proposes a validation and optimization tool that can be integrated to the system development environment for the early detection of logical errors prior to the system realization and validation phase.

6.2 Highlights of the Work

The approach towards the code validation and optimization of RISC microcontrollers, presented in this thesis has explored the following .

(i) A new application domain for the static analysis of machine code; a code validation technique to assist a programmer in developing error free and reliable embedded software which would reduce the development time as well as improve the quality of the software has been presented. In order to achieve the early detection of bugs in programs being developed, more than hundred rules of inferences based on the instruction set and architectural features of the target processor have been formulated. Codification of the formulated rules has been done in proportional logic. The construction of the CFG from the machine codes and identification of all possible execution paths required for the analysis also have been incorporated.

(ii) The feasibility of the approach has been verified on systems based on PIC16F87X series of microcontrollers. The algorithm can encompass a wide range of RISC processors, once appropriate rules are available for such processors. The pattern of the code sequence used for the evaluation of the governing rules has been identified. A method to automatically localize faults, diagnose, and correct if possible, has been proposed to enhance the debugging process. The validation tool that is developed has been tested with sample programs and the discrepancy in the instruction sequence made by the programmer in configuring the CPU and integrated peripherals functioning in different cases have been identified.

(iii) A relation matrix has been formulated which assists the code analyzer in identifying the active memory bank associated with each code in the

instruction stream. A state transition diagram that shows the memory bank state transition from PAMB to AMB corresponding to each bank selection instruction has been used for the optimization.

(iv) An optimization algorithm has been developed and implemented for a static machine code analyzer which helps to eliminate the redundant data as well as program memory bank switching instructions in partitioned memory architectures. A compiler strategy which utilizes the algorithm developed to determine the optimum data allocation to the available memory bank, resulting in the minimum number of bank switching code has been proposed. The feasibility of the approach has been verified on systems based on PIC16F87X series of microcontrollers. Optimization of the code based on the architectural features of the target processor considered is also included. Validation and optimization technique in the intraprocedural sequence loops and interprocedural routines in an application program are considered. The tool has been tested with sample programs and the results of the analysis have been studied.

6.3 Merits and Demerits

A code validation and optimization technique assisting the embedded system software debugging to make it more effective at revealing errors and redundancy is proposed. Since the method adopts a static analysis, the tool developed has the merits and demerits of static analysis. Since the analysis is done on machine code this work has got the advantages and disadvantages of machine code analysis.

A static analysis of machine code can provide information which can hardly be discovered by traditional simulation or test techniques. Commercial static

analysis tools typically check for standard deficiencies but cannot identify any logical mistakes. But the technique presented here is based on rules of inferences formulated for the target processor that look into platform-specific properties. This contributes to early detection of software bugs resulting from subtle deviations of the hardware specification that had slipped through conventional testing which would lead to malfunctioning at runtime.

Dynamic techniques are generally limited to finding bugs in the program paths that are actually executed whereas static analysis can find bugs in all possible execution paths. Though dynamic program slicing is useful in debugging of programs, the sizes of dynamic-dependence graphs can be very large and thus it is not possible to keep them in memory for realistic program runs. The validation method adopted in this work identifies the program states mainly with machine code pattern, which drastically reduces the state space creation contributing to an improved state-of-the-art model checking.

Instructions that perform memory operations use explicit memory addresses and indirect addressing, which complicates the task of understanding the overall behavior of the code with static analysis. To be able to analyze the control flow of a program available in machine code the boundaries of the low level instructions with which the program is constructed is to be detected, which can again cause several problems. When a function is called from different call sites the determination of the return addresses present further problems. It is very hard to furnish a general solution that handles all the problems associated with the control flow analysis of machine code, but with more information and some architecture specific heuristics the problems become manageable.

The results obtained show that the technique for validation and optimization is independent of the compiler/assembler used for the software development. The

compiler introduced redundancy can also be identified since the proposed approach is realized through the static analysis of machine code. There is no hardware overhead for detection of errors; nor any increase in the code size by this approach, as no signatures or labels are embedded into the program so also no run time overhead. As the architecture advances only the code sequence to be tested need to be varied while the methodology remains the same.

Error checking schemes based on codes usually have limited fault coverage. In this work the coverage of faults depends on the rules governing the validation of codes. Applying rules to validate the code and thus making code inspection automatic eliminates intensive manual effort. Finding and eliminating dead code and inefficient code can help ensure that the software uses the full potential of the hardware. Since the analysis is performed on the low level language the range of types of analysis that can now be performed on the code is significantly reduced. To make the system interactive a disassembler/ decompiler is to be incorporated.

The technique presented in this work achieves optimization of bank switching instructions without much computational burden by analyzing the machine code with a comparatively simple algorithm. Analysis of a high level program cannot easily determine the current memory bank state in partitioned memory architectures. But with a static analysis of the machine code, the state transitions at each bank switching instruction can be easily determined. Redundant data memory bank selection instructions in the intraprocedural sequence, loops and interprocedural routines in the application program can be eliminated. The relation matrix assists the code analyzer in identifying the active memory bank associated with each code in the instruction stream. *Unsound* techniques may identify some inaccurate warnings unless proper care is taken for suppression of false warnings. The suppression of false warning is done by considering the transparent nodes

which is a node without any bank switching instructions and also by considering the union of the active memory bank associated with the incoming edges of a subprogram for interprocedural analysis.

Though the technique described is general, the implementation is architecture oriented, and hence the feasibility study is conducted on PIC16F87X microcontrollers. This method scales well into large number of memory blocks as well as other architectures, once appropriate information is available. The proposed technique can be treated as an extension of conventional debuggers and can be incorporated as part of the Integrated Development Environments resulting in improved software quality and reduced debugging time. Automatic analysis of the machine code with rules proposed here can eliminate many of the iterations like edit, compile, assemble, link and download needed for debugging.

Instructions managing the program memory pages of processors in an application program also can be optimized with a similar technique. The evaluation of compiler optimization in banked memory architecture can be made by switching on and off the optimization of the compiler and counting the number of elimination done with this analysis. Using this tool the optimizing performance of different compilers for partitioned memory architecture can be evaluated. High performance compilers can integrate this technique for better performance and reduced code size. With this post pass optimizer the compiler/programmer is deprived of the complicated analysis for minimal placement of bank selection instructions.

Instruction reordering without affecting the program within the basic blocks can further improve the bank selection optimization. The execution frequency of a node (instruction) is not considered, nor is the run time optimizing attempted. The error location and identification are automatic while error repair needs human intervention.

6.4 New Research Directions

Integrating bug-finding tools into the development process by making them part of Integrated Development Environments is an important direction for future research. Methods for identification and suppression of false warnings can be developed. Improving the scalability of the analysis is another direction for future research. This work can also be extended as a debugging tool suitable for SoC/FPGA system development incorporating embedded processors. Improving the Precision of the analyses by employing both static and run-time checks can be explored.

Elimination of redundant bank switching instructions can be used for a processor core based system to select the number of data memory banks and the size of each bank resulting in the optimized design instead of using a single scratchpad RAM. Incorporating execution profile for the estimation of run time optimization offers further scope for research.

6.5 Summary

This thesis proposes a static analyzer for embedded system software, which is close to a target level testing tool that is portable. Primary goal is to develop techniques that can be implemented in tools that are useful for embedded software developers for the early validation and optimization of their code by conducting a static analysis on the machine code. The focus of this work is to develop methods that automatically localize faults and thus enhance the debugging process as well as reduce human interaction time without software or runtime overhead. Analysis is

done on machine code rather than source code because this eliminates the requirement of knowledge of the semantics of high level language/assembly language; it is also independent of the compiler, developers get the freedom to change compilers or compiler versions.

APPENDIX- A

QUANTITATIVE ANALYSIS

This thesis concentrates on validation of the embedded system codes by way of detection and localization of errors if any and identification of the cause of the error, by conducting a static analysis on machine codes thereby helping the developer to optimize the system performance.

The work suggests a systematic and structured approach for the formation of a rule based code validation, error detection and localization scheme; a system has been developed and realized for a popular microcontroller series PIC16F87X. As a case study more than one hundred rules have been formulated by detailed study/analysis and experimenting with the system. Method/scheme for incorporating more rules for improving the performance and even extending to other processors and families has also been described in sections 4.1.5, 4.4.1 and 5.2. In addition to this a plausible situation which drastically increases the code size has been identified and to solve this problem, an algorithm has been realized to improve/optimize the code by considering the same series of processors as a case

study. Table 5.3 shows the analysis summary of the optimization done for sixteen different programs developed using various compilers.

Efficiency of the proposed system depends on how many rules are applied to validate the code, the way in which the rules are formulated and codified and whether the soundness of the rules are checked in all possible execution paths of the program considered. The total number of rules that can be formulated for a given family of processors tends to be dynamic. As the processor architecture as well as the instructions are examined deeper and deeper and by considering the various ways in which a programmer might use the instructions in a program, more and more rules are likely to be formulated. Most of the rules formulated in this work have been codified and tested. Our intent was to demonstrate the feasibility as well as the efficacy of the proposed mechanisms. The analysis time for a few hundred program words is typically below one second.

Unlike in code optimization, validation is done based on rules which restrict the analysis to perform as a qualifier, indicating whether the rule tested is passed or failed which leads to error detection. If the rules are applied exhaustively the error detection is 100 percent. Due to the discreteness in the results the accuracy is implied. The table A.1 shows the results of the analysis using programs developed for various embedded systems. The tool validated all the working programs but when stuffed with errors and tested, it pin pointed the errors giving the cause of each error irrespective of the compiler adopted. Improvement in the quality of the code has been achieved by optimization. When extended to other family of microcontrollers, the algorithm remains valid, only the code sequence to be tested varies.

Table A.1 Evaluation of programs developed using different compilers/assemblers

Filename	*Compiler/ assembler used*	*Code size (program words)*	*Errors detected, localized and identified*	*bank switching codes present*	*redundant bank switching codes*
DAS_ok	HI-TECH C	84	nil	18	8
DAS _err	HI-TECH C	89	5	18	8
DAS_ok	mikroC	56	nil	10	2
DAS _err	mikroC	56	4	10	2
DAS_ok	WIZ-C	136	nil	11	1
DAS _err	WIZ-C	136	5	11	1
DAS_ok	MPASM	81	nil	9	1
DAS _err	MPASM	79	2	9	1
Traffic_ok	HI-TECH C	79	nil	12	6
Traffic _err	HI-TECH C	74	2	12	6
Traffic_ok	mikroC	72	nil	10	2
Traffic _err	mikroC	72	2	8	2
Traffic_ok	WIZ-C	128	nil	11	1
Traffic _err	WIZ-C	134	2	11	1
Traffic_ok	MPASM	43	nil	14	6
Traffic _err	MPASM	41	2	14	6

Section 2.2.2.5 of the thesis describes the techniques proposed to find bugs in embedded software with static analysis. The types of bugs detected in the work presented by Regehr [23, 24] and Venkitaraman [30] are not comparable with the bugs detected in this work. Bastian Schlich in his thesis [29] describes a new approach for model checking software for microcontrollers [mc]square, which uses assembly code as input. The aim is to verify the correctness of the program under consideration by model checking it with respect to a specification given by a temporal formula. The errors detected include compiler errors, reentrance problems, stack overflows, and unintended use of microcontroller features. His approach needed a special simulator to build the state space whereas the work presented in this thesis doesn't require a simulator. It requires tracking a large state space, which limits the analysis to certain code sizes. But in the proposed validation tool the program states are identified mainly with machine code patterns

which drastically reduce the state space creation contributing to an improved state-of-the-art model checking.

The analysis done by Fehnker et al. given in [188] checks properties that are specific for microcontroller programs such as use of reserved registers, interrupt behavior and timer usage using a source code static analysis tool and a symbolic model checker as the underlying engine. Finding appropriate CTL formula and extending the parser to accept a specific ANSI dialect of C are the main challenges and the work doesn't present any quantitative analysis. Coding the rules in propositional logic is easier in the proposed work and can identify more types of processor specific errors as the compliance checking of each rule results in the detection of one or more errors. To the best of author's knowledge no other architecture oriented approach has been reported for the validation of embedded software; a comparative performance evaluation in the strict sense therefore remains out of bounds of this work. Nevertheless salient features of the proposed tools have been compared in sections 4.5, 5.1 and 6.3.

REFERENCES

[1] Knight J. C., "Issues of Software Reliability in Medical Systems", *Proceedings of third annual IEEE Symposium on Computer-Based Medical Systems*, Chapel Hill, USA, IEEE CS Press, New York, pp. 153-160, 1990.

[2] Kenneth Kennedy, "An Exploration of the Issues Affecting the Development of Software-Based Safety-Critical Systems", http://www.clicktoconvert.com/, 2005.

[3] Gergely Pinter, *Model Based Program Synthesis and Runtime Error Detection for Dependable Embedded Systems*, Ph.D Thesis, Budapest University of Technology and Economics, 2007.

[4] Wilmshurst T. '*An Introduction to the Design of Small-Scale Embedded Systems'*, Palgrave, ISBN 0-333-92994-2, 2001.

[5] David. E. Simon, '*An Embedded Software Primer'*, Addison Wesley Longman Inc., USA, 1999.

[6] Arnold S. Berger, '*Embedded Systems Design: An Introduction to Processes, Tools, and Techniques'*, CMP Books, USA, 2002.

[7] Jean Labrosse et al., '*Embedded Software: Know It All'*, Elsevier Inc., 2008.

[8] P. Riachi, "HI-TECH OCG Compiler Support for Microchip's PIC10/12/16", *Embedded Systems Conference*, San Jose, California, April 15, 2008.

[9] Sudheendra Hangal and Monica S. Lam, "Tracking down Software Bugs Using Automatic Anomaly Detection", *International conference on software Engineering*, pp. 291-301, May 2002.

[10] A. Avizienis and J.C. Laprie, "Dependable computing: From concepts to design diversity," *Proe. IEEE,* Vol. 74, pp 629-638, May 1986.

[11] Murugesan S., “Dependable Software through Fault Tolerance”, *In Proceedings of the IEEE TENCON*, pp. 391-399, 1989.

[12] Mark Weiser, “Programmers Use Slices When Debugging”, Communications of the ACM, 25(7), pp. 446-452, July 1982.

[13] Cifuentes C. and Fraboulet A., “Intraprocedural Static Slicing of Binary Executables”, In *Int.Conf. on Softw. Maint.*, pages 188–195, 1997.

[14] Kiss A., Judit J., Lehotai G. and Tibor, G., “Interprocedural Static Slicing of Binary Executables”, *Proceedings of the third IEEE international workshop on Source Code Analysis and Manipulation*, pp. 118-127, September 2003.

[15] Balakrishnan G. and Reps T., “Analyzing Memory Accesses in x86 Executables”, In ‘*Comp.Construct’*., Lec. Notes in Comp. Sci., pp. 5–23. Springer-Verlag, 2004.

[16] Zhang X. and Guptha R., “Whole Execution Traces and Their Applications”, *ACM Transactions on Architecture and Code Optimization*, Vol. 2No. 3, September, pp. 301-334, 2005.

[17] B. Scholz, B. Burgstaller, and J. Xue, “Minimal Placement of Bank Selection Instructions for Partitioned Memory Architectures,” *ACM Trans. on Embedded Computing Systems (TECS)*, vol. 7, Issue 2, pp.1-32, February 2008.

[18] Application notes, AN586, Macros for Page and Bank Switching, Microchip Technology Inc, http:// www.microchip.com, 1997.

[19] P. Cuenot, B. Tavernier and J.M. Talbot, White Paper: “Embedded Software Validation and Verification using Virtual Platforms for Powertrain Applications”, ACES 09.

[20] D. Lioupis , A. Papagiannis , D. Psihogiou, “A systematic approach to software peripherals for embedded systems”, *Proceedings of the ninth international symposium on Hardware/software codesign*, Copenhagen, Denmark, pp.140-145, April 2001.

[21] Tankut Akgul, Pramote Kuacharoen, Vincent J. Mooney and Vijay K. Madisetti, “A Debugger RTOS for Embedded Systems”, *27th Euromicro Conference 2001*, A Net Odyssey (euromicro'01), Warsaw, Poland, September 04 – 06, 2001.

[22] Tom Erkkinen, “Production Code Generation for Safety-Critical Systems”, *SAE world congress International*, 2004.

[23] John Regehr and Alastair Reid, "HOIST: A System for Automatically Deriving Static Analyzers for Embedded Systems", *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, Boston, MA, USA, October 07-13, 2004.

[24] Regehr J., Reid A. and Webb K., "Eliminating Stack Overflow by Abstract Interpretation", *ACM Transactions on Embedded Computing Systems (TECS)*, Volume 4, No. 4, pages 751-778, November 2005.

[25] NASA report "Independent Verification and Validation Of Embedded Software", Marshall Space Flightcenter, Practice No. PD-ED-1228, NASA, Feb. 1999.

[26] Sterling N., "WARLOCK — a static data race analysis tool". *In Proc. USENIX Annual Technical Conf.*, winter, pp. 97–106, 1993.

[27] Ball T., Rajamani S. K., "The SLAM project: Debugging system software via static analysis", *In Proc. 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, pp. 1–3, January 2002.

[28] Flanagan C., Leino K. R. M, Lillibridge M., Nelson G., Saxe J. B., and Stata R., "Extended static checking for Java", *In Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Berlin, Germany, pp. 234–245, June 2002.

[29] Schlich B., *Model Checking of Software for Microcontrollers*, Dissertation thesis, RWTH Aachen University, 2008.

[30] Venkitaraman R. and Gupta G., "Static program analysis of embedded executable assembly code", in *proceedings of the international conference on Compilers, Architecture and Synthesis for Embedded Systems archive,* pp. 157 – 166, 2004.

[31] Y. Mengting, W. Guoqing, and Y. Chao, "Optimizing Bank Selection Instructions by Using Shared Memory", *Proceedings of the International Conference on Embedded Software Systems (ICESS)*, pp. 447-45, 2008.

[32] M. Li, C. J. Xue, T. Liu and Y. Zhao, "Analysis and Approximation for Bank Selection Instruction Minimization on Partitioned Memory Architecture", *ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES'10)*, Stockholm, Sweden, pp. 1-8, April 13–15, 2010.

[33] Q. Li, Y. He, Y. Chen, W. Wu and W. Xu, "A Heuristic Algorithm for Optimizing Page Selection Instructions", *Proceedings of the IEEE 2nd International Conference on Software Technology and Engineering(ICSTE)*, pp. v2-143 to v2-148, 2010.

[34] L. Tiantian, M. Li, and C. J. Xue, "Joint Variable Partitioning and Bank Selection Instruction Optimization on Embedded Systems with Multiple Memory Banks", *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC 2010)*, pp. 113-118, 2010.

[35] Tim Wilmshurst, '*Designing Embedded Systems with PIC Microcontrollers-Principles and applications*', Newnes, Elsevier, London, UK, 2007.

[36] R. Kamal, '*Embedded Systems-Architecture, Programming and Design*', McGraw-Hill, New Delhi, 2008.

[37] Harry Koehnemann and Timothy Lindquist. "Towards Target-Level Testing and Debugging Tools for Embedded Software", *In Conference Proceedings on TRI-Ada*, pages 288 - 298. ACM, September 1993.

[38] Rainer Leupers, '*Code Optimization Techniques for embedded processors-Methods, Algorithms and Tools*', Kluwer Academic Publishers, 2000.

[39] Aho A.V. and Ullman J.D., '*Principles of Compiler Design*', Addison-Wesley/Narosa, New Delhi, 1985.

[40] S. S. Muchnick, '*Advanced Compiler Design Implementation*'. Morgan Kaufman Publishers, San Francisco, CA, 1997.

[41] A. Sudarsanam and S. Malik, "Memory Bank and Register Allocation in Software Synthesis for ASIPs", *Int. Conf. on Computer-AidedDesign (ICCAD)*, 1995.

[42] M. Saghir, P. Chow and C. Lee, "Exploiting Dual Data-Memory Banks in DigitalSignal Processors", *7th International Conference on Architectural Support forProgramming Languages and Operating Systems*, 1996.

[43] David Hovemeyer**,** William Pugh, "Finding bugs is easy"**,** *ACM SIGPLAN Notices***, v**olume 39, Issue 12, pp. 92 - 106, December 2004.

[44] William H., Mangione-Smith, Brad Hutchings, David Andrews, André DeHon, Carl Ebeling, Reiner,et. al., "Seeking Solutions in Configurable Computing", *IEEE Computer*, pp. 38-43, December 1997**.**

[45] Zainalabedin Navabi, '*Digital Design and Implementation with Field Programmable Devices*', Kluwer Academic Publishers, Boston, 2005

[46] Christophe Bobda, '*Introduction to Reconfigurable Computing-Architectures, Algorithms, and Applications*', Springer, Netherlands, 2007.

[47] Ville-Veikko Helppi and Colin Walls, **"**Prevention is Better Than the Cure:Compiler Run-Time Error Checking" Embedded Systems White Paper, www.mentor.com/embedded**,** June 2009.

[48] B. G. Kolkhorst, A. J. Macina, "Developing Error-Free Software", *IEEE AES Magazine*, pp. 25-31, November 1988.

[49] Lerie Kane, "Creating High Performance Embedded Applications Through Compiler Optimizations", Technology@Intel Magazine, March 2005.

[50] Elliot J. Chikofsky and Burt L. Rubenstein, "CASE: Reliability Engineering for Information Systems", *IEEE Software*, pp. 11-15, March 1988.

[51] Peter S. Gilmour, "How to Select Tools for Microcontroller Software", *IEEE Spectrum*, February 1991.

[52] B. Hailpern and P. Santhanam, "Software Debugging, Testing, and Verification", *IBM Systems journal, special issue- Software Testing and Verification*, Volume 41, Number 1, 2002.

[53] Kapur S., Sriprasad C., "Software development tools for embedded systems", *Digital Avionics Systems Conference*, 14th DASC Volume, Issue, 5-9 Page(s):331 – 335, Nov 1995.

[54] Jens Palsberg and Mayur Naik, "ILP-Based Resource-Aware Compilation", In Ahmed Jerraya and Wayne Wolf, editors, '*Multiprocessor Systems-on-Chips',* Morgan Kaufmann, 2004.

[55] Zoltan Somogyi, Fergus J. Henderson, and Thomas Conway, "Mercury: an efficient purely declarative logic programming language", *In Proceedings of the Australian Computer Science Conference*, Glenelg, Australia, pages 499-512, February 1995.

[56] Chirs Inacio, "Software Fault Tolerance", 18-849b *Dependable Embedded Systems*, Spring 1998,
http://www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/

[57] Stefan Gossens and Mario Dal Cin," Structural Analysis of Explicit Fault-Tolerant Programs", *Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE'04)*, 2004.

[58] Zaipeng Xie, Hongyu Sun, Kewal Saluja "A Survey of Software Fault Tolerance Techniques"; found at http://homepages.cae.wisc.edu/~ece753/INFO.html, 2008.

[59] A. Avizeinis, "The N-Version Approach to Fault-Tolerant Software", *IEEE Transactions of Software Engineering*, Vol. SE-11, No. 12, pp. 1491-1501, December 1985.

[60] D. J. Holding, "Software Fault Tolerance", *IEE Colloquium on Fault Tolerant Techniques,* pp: 6/1-6/9, 11 May 1990.

[61] J. Gray and D. P. Siewiorek, "High-Availability Computer Systems," *IEEE Computer*, 24(9):39-48, September 1991.

[62] http://msdn.microsoft.com/en-us/vstudio/default.aspx, 2009.

[63] R. Abreu, *Spectrum-based Fault Localization in Embedded Software*, Ph D. thesis, Delft University of Technology, October 2009.

[64] Ehud Y. Shapiro, *Algorithmic Program Debugging,* PhD thesis, Yale University, New Haven, Connecticut, 1982.

[65] Scott Renner, "Location of Logical Errors on Pascal Programs with an Appendix on Implementation Problems in Waterloo PROLOG/C", *Technical Report UIUCDCS-F-82-896*, April 1982.

[66] Peter Fritzson, Tibor Gyimothy, Mariam Kamkar, and Nahid Shahmehri, "Generalized algorithmic debugging and testing", In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Canada, pp. 317–326, June 26–28 1991.

[67] Mireille Ducasse and Anna-Maria Emde, "A Review of Automated Debugging Systems: Knowledge, Strategies, and Techniques", In *Proceedings of the 10th International Conference on Software Engineering*, Singapore, pp. 162–171, April 1988.

[68] Rudolph E. Seviora, "Knowledge–based Program Debugging Systems", *IEEE Software*, 4(3):20–32, May 1987.

[69] Mark Weiser, "Program Slicing", *IEEE Transactions on Software Engineering*, SE-10(4), pp. 352-357, July 1984.

[70] Mark Weiser and Jim Lyle, "Experiments on slicing–based debugging aids", In Elliot Soloway and Sitharama Iyengar, editors, '*Empirical Studies of Programmers'*, pages 187–197, Ablex Publishing Corp., Norwood, New Jersey, 1986.

[71] James R. Lyle and Mark Weiser, "Automatic program bug location by program slicing", In *Proceedings of the 2nd International Conference on Computers and Applications*, Beijing, PRC, pp. 877–883, June 1987.

[72] H. Agrawal and J. R. Horgan, "Dynamic program slicing", In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, White Plains, New York, June 1990.

[73] Bogdan Korel and Janusz Laski, "Dynamic slicing of computer programs", *Journal of Systems and Software*, 13(3):187-195, November 1990.

[74] H. Agrawal, "On Slicing Programs with Jump Statements", *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pp. 302–312, 1994.

[75] Leon Osterweil, "Integrating the testing, analysis, and debugging of programs", In H. L. Hausen, editor, '*Software Validation'*, pp. 73–102. Elsevier Science Publishers B. V., North–Holland, 1984.

[76] Lori A. Clarke and Debra J. Richardson, "The application of error–sensitive testing strategies to debugging" In *Proceedings of the ACMSIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, California, pp. 45–52, March 1983.

[77] B. Korel and J. Laski' "STAD - A system for Testing and Debugging: User Perspective", *In Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, Alberta, Canada, pp. 13-20, July 1988.

[78] Bogdan Korel, "PELAS – program error-locating assistant system", *IEEE Transactions on Software Engineering*, SE-14(9):1253–1260, September 1988.

[79] Bogdan Korel and Janusz Laski, "Algorithmic software fault localization", In *Proceedings of the Twenty-Fourth Annual Hawaii InternationalConference on System Sciences*, Hawaii, pp. 246–252, January 1991.

[80] James S. Collofello and Larry Cousins, "Towards automatic software fault location through decision–to–decision path analysis", In *AFIPS Proceedings of 1987 National Computer Conference*, Chicago, Illinois, pp. 539–544, June 1987.

[81] Pan H. and E. H. Spafford, "Heuristics for Automatic Localization of Software Faults", in *Proceedings of the 10th Pacific Northwest Software Quality Conference*, pp. 192—209, Oct 1992.

[82] Mayer W., *Static and Hybrid Analysis in Model-based Debugging*, Ph.D thesis, School of Computer and Information Science, University of South Australia, 2007.

[83] P. Zoeteweij, J. Pietersma, R. Abreu, A. Feldman, and A.J.C. van Gemund, "Automated Fault Diagnosis in Embedded Systems", In *Proceedings of the 2nd IEEE International Conference on Secure Systems and Reliability Improvement (SSIRI'08)*, Yokohama, Japan, pp. 103-110, July 2008.

[84] Jones J. A., Harrold M. J. and Stasko J. T., "Visualization of test information to assist fault localization", In *Proceedings of the 22rd International Conference on Software Engineering (ICSE'02)*, Orlando, Florida, USA, ACM Press, pp. 467–477, 2002.

[85] Dallmeier V., Lindig C., and Zeller A., "Lightweight defect localization for java", In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, 2005.

[86] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Efficient Debugging With Slicing and Backtracking", *Software Practice & Experience*, 23(6), pp. 589-616, June 1993.

[87] http://www.ghs.com/products/MULTI_debugger.html, 2009.

[88] S. Horowitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems* 12, No. 1, 26–60, January 1990.

[89] Binkley, D. & K. Gallagher, "Program slicing", *Advances in Computers*, Vol. 43, pp. 1-10, 1996.

[90] Karl J. Ottenstein and Linda M. Ottenstein, "The Program Dependence Graph in Software Development Environments", *SIGPLAN Notices,* 19(5), pp. 177-184, May 1984.

[91] Viravan C., *Enhancing Debugging Technology*, Ph.D dissertation, Purdue University, March 1994.

[92] Livadas, P. E. and S. Croll, "Program slicing", *Technical Report SERC-TR-61-F, Software Engineering Research Centre*, October 1992.

[93] Hiralal Agrawal, *Towards Automatic Debugging of Computer Programs*, Ph.D thesis, Purdue University, West Lafayette, 1991.

[94] Tip F., "A survey of program slicing techniques", Journal of Programming Languages, Volume 3, Issue 3, pp. 121–189, September 1995.

[95] Bogdan Korel and Janusz Laski, "Dynamic program slicing", *Information Processing Letters*, 29, 3, 155-163, 1988.

[96] J. Bergeron, M. Debbabi, J. Desharnais, M.M. Erhioui, Y. Lavoie, and N. Tawbi, "Static Detection of Malicious Code in Executable Programs", *Int. J. of Req. Eng.*, 2001.

[97] P. Cousot: Integrating physical systems in the static analysis of embedded control software. K. Yi (Ed.) : Third Asian Symposium on Programming Languages and Systems (APLAS), Lecture Notes in Computer Science (LNCS 3780), pages 135–138, Springer-Verlag, 2005.

[98] S. Johnson Lint, "A C Program Checker", In *UNIX Programmer's Supplementary Documents Volume 1 (PS1)*, April 1986.

[99] D. L. Detlefs, R. M. Leino, G. Nelson and J. B. Saxe, "Extended Static Checking", *SRC Research Reports SRC-159, Compaq SRC*, December 1998.

[100] W. R. Bush, J. D. Pincus and D. J. Sielaff, "A Static Analyzer for Finding Dynamic Programming Errors", *Software Practice and Experience,* Vol. 30, No. 7, pp. 775- 802, 2000.

[101] D. Engler, B. Chelf, A. Chou and S. Hallem, "Checking System Rules Using System-Specific, Programmer- Written Compiler Extensions", In *Proceedings of the Fourth Symposium on Operating System Design and Implementation*, October 2000.

[102] P. Cousot, "Integrating Physical Systems in the Static Analysis of Embedded Control Software". K. Yi (Ed.): Third Asian Symposium on Programming Languages and Systems (APLAS), Lecture Notes in Computer Science (LNCS 3780), pp. 135–138, Springer-Verlag, 2005.

[103] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Min´e, D. Monniaux, and X. Rival, "The Astr'ee Analyser", In M. Sagiv, editor, *Proc. 14th ESOP'2005, Edinburg, UK*, volume 3444 of *LNCS*, Springer, pages 21–30, Apr. 2-10, 2005.

[104] P. Baudin, A. Pacalet, J. Raguideau, D. Schoen and N. Williams, "Caveat: A Tool for Software Validation", *Proceedings of the International Conference on Dependable Systems and Networks*, DSN, 2002.

[105] N. Rutar, C. Almazan, and J. S. Foster, "A Comparison of Bug Finding Tools for Java", In *Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering*, Saint-Malo, France, November 2004.

[106] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns", In *Proceedings of the 12th USENIX Security Symposium,* Washington DC, USA, pp. 169-186, August 4-8, 2003.

[107] G. Balakrishnan, R. Gruian, T. W. Reps, and T. Teitelbaum, "Codesurfer/x86-A Platform for Analyzing X86 Executables" In R. Bodík, editor, CC, volume 3443 of Lecture Notes in Computer Science, Springer, pp. 250—254, 2005.

[108] Intel 64 and IA-32 Architectures Software Developer's Manuals, vol. 2B, 2006.

[109] C. Cifuentes and K. Gough, "Decompilation of Binary Programs", *Software Practice & Experience*, 25(7), pp. 811-829, July 1995.

[110] C. Cifuentes, D. Simon, and A. Fraboulet, "Assembly To High-Level Language Translation", In *Int. Conf. on Softw. Maint.*, pp. 228–237, 1998.

[111] W.C. Hsieh, D. Engler, and G. Back, "Reverse-Engineering Instruction Encodings", In *Proceedings of the USENIX Annual Technical Conference*, Boston, Mass, pp. 133-146, June 2001.

[112] J.T. Giffin, S. Jha, and B.P. Miller, "Detecting manipulated remote call streams", In *Proceedings of 11th USENIX Security Symposium*, 2002.

[113] C. Linn and S. Debray, “Obfuscation of executable code to improve resistance to static disassembly”, In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, pp. 290-299, October 2003.

[114] Christopher Kruegel, William Robertson, Fredrik Valeur and Giovanni Vigna, “Static Disassembly of Obfuscated Binaries”, In *Proc. of the 13th USENIX Security Symposium*, 2004.

[115] R. Wang, Flash In The Pan? *Virus Bulletin*, Virus Analysis Library, July 1998.

[116] Goloubeva O. Rebaudengo M.S., Reorda M.S., and Violante M., “Improved Software-based Processor Control-flow Errors Detection Technique”, In *Reliability and maintainability symposium*, pp. 583–589, January 2005.

[117] N. Leveson and C.S.Turner, “An Investigation of the Therac−25 Accidents”, *IEEE Computer*, Vol. 25, No. 7, July 1993.

[118] J. R. Larus, “Whole Program Paths”, In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA. New York, NY: ACM Press, pp. 259–269, 1999.

[119] T. M. Chilimbi, “Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality”, In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, UT. New York, NY: ACM Press, pp. 191–202, 2001.

[120] M. Burtscher and M. Jeeradit, “Compressing Extended Programtraces Using Value Predictors”, In *International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, IEEE Computer Society, pp. 159–69, 2003.

[121] X. Zhang and R.Gupta, “Cost Effective Dynamic Program Slicing’, In *ACMSIGPLAN Conference on Programming Language Design and Implementation*, Washington, DC. New York, NY: ACM Press, pp. 94–106, 2004.

[122] Q. Jacobson, E. Rotenberg, and J. E. Smith, “Path-Based Next Trace Prediction”, In *IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, IEEE Computer Society, pp. 14–23,1997.

[123] R. Gupta, D. Berson, and J. Z. Fang, Path profile guided partial redundancy elimination using speculation”, In *IEEE International Conference on Computer Languages*, Chicago, IL, Washington, DC, IEEE Computer Society, pp. 230–39, 1998.

[124] B. Calder, P. Feller, and A. Eustace, "Value Profiling", In *IEEE/ACM International Symposium on Microarchitecture*, Washington, DC, IEEE Computer Society, pp. 259–69.1997.

[125] J. Yang and R. Gupta, "Frequent Value Locality and Its Applications", ACM *Transactions on Embedded Computing Systems*, New York, NY: ACM Press, 1(1):79–105. 2002.

[126] T. M. Chilimbi and M. Hirzel, "Dynamic hot data stream prefetching for general-purpose programs", In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY: ACM Press, pp. 199–209. 2002.

[127] C. B. Zilles and G. Sohi, "Understanding The Backward Slices Of Performance Degrading Instructions", In *ACM/IEEE 27th International Symposium on Computer Architecture*, New York, NY: ACM Press, pp. 172–81, 2000.

[128] Y. Sazeides, "Instruction-Isomorphism In Program Execution", Journal *of Instruction Level Parallelism*, 5:1–22, 2003.

[129] Glenford J. Myers, '*The Art of Software Testing',* John Wiley & Sons Inc., 1979.

[130] Child Jeffrey, "32-bit Emulators Struggle with Processor Complexities", *Computer Design*, May 1,1991.

[131] Ted Bapty, "Embedded System Validation for Polymorphous Computing Architectures", *white paper, Institute for Software Integrated Systems*, Vanderbilt University, 2001.

[132] Colin Walls, "Debugging Embedded Systems with a Real-Time Operating System", *ECE Magazine*, pp 35-38, June 2007, http://www.embedded-control-europe.com/ece_magazine.

[133] Christine Peng, "On-Chip System Protection Basics for HCS08 Microcontrollers", Freescale Semiconductor, Application Note AN3305, 2007.

[134] DF6811CPU, 8-bit FAST Microcontrollers Family ver 2.17, Digital Core Design, http://www.DigitalCoreDesign.com, 2009.

[135] Leontie, E., Bloom, G., Gelbart, O., Narahari, B. and Simha, R., "A Compiler-Hardware Technique for Protecting Against Buffer Overflow Attacks", *Journal of Information Assurance and Security*, Vol.5, pp. 001-008, 2010.

[136] Joint Test Action Group (JTAG) web page, http://www.jtag.com/.

[137] Steve Furber, '*ARM system-on-chip architecture'*, Addison-Wesley, March 2000.

[138] E. Miller and W.E. Howden, "Software Testing and Validation Techniques**",** *IEEE Computer Society Press*, 1981.

[139] Tom Erkkinen, "Safety-Critical Software Development Using Automatic Production Code Generation", The *MathWorks, Inc*., 2007.

[140] Brett Murphy, Amory Wakefield, and Jon Friedman, "Best Practices for Verification, Validation, and Test in Model-Based Design", 2008-01-1469, Technical notes, The MathWorks, Inc., 2008.

[141] A. Pnueli, M. Siegel, O. Shtrichman, "The Code Validation Tool (CVT)--Automatic Verification of Code Generated from Synchronous Languages", in: B. Steffen (Ed.), *Proc. of the Software Tools for Technology Transfer* (STTT'98), 1998.

[142] IAR visualSTATE-State Machine Design Automation for Embedded Systems, www.phaedsys.org/principals/**iar**/**iar**data/visualstatre.pdf., 2009.

[143] István Majzik, "Concurrent Error Detection Using Watchdog Processors", *Fault tolerant computing systems*, volume 283, informatik, 1996.

[144] Ragel Roshan G. and Parameswaran S. " Hardware Assisted Pre-emptive Control Flow Checking for Embedded Processors to Improve Reliability**",** *Proceedings of the 4th IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, (CODES+ISSS '06), Seoul, Digital Object Identifier: 10.1145/1176254.1176280, pp. 100-105, 22-25 October 2006**.**

[145] S. S. Yau and F. Chen, "An approach to concurrent control flow checking", IEEE *Trans. Software Eng.*, 6(2), pp.126–137, 1980.

[146] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante, "Soft-Error Detection Using Control Flow Assertions", *Proc. Defect and Fault Tolerance in VLSI Systems*, pp. 581 – 588, 2003.

[147] A.Mahmood and E. J.McCluskey, "Concurrent error detection using watchdog processors-a survey", *IEEE Trans. Computers*, 37(2):160–174, February 1988.

[148] N.R. Saxena, E.J. McCluskey, "Control Flow Checking Using Watchdog Assists and Extended-Precision Checksums", *IEEE Transactions on Computers*, Vol. 39, No. 4, pp. 554-559Apr. 1990.

[149] J.P. Shen and M.A. Schuette, "On-line Self-Monitoring Using Signatured Instruction Streams", *Proc. ITC'83*, pp. 275-282, 1983.

[150] Michael A. Schuette and John Paul Shen, "Processor control flow monitoring using signature instruction streams", *IEEE Trans.on Computers*, vol. C-36, No 3, March 1987, pp.264-275.

[151] Janusz Sosnowski, “Detection of control flow errors using signature and checking instructions”, IEEE *International Test Conference*, pp. 81–88, 1988.

[152] N. Oh, P.P. Shirvani, E.J. McCluskey, “Control-Flow Checking by Software Signatures”, *IEEE Transactions on Reliability*, Vol. 51, No. 2, pp. 111-122, March 2002.

[153] R. van Engelen, D. Whalley, and X. Yuan, “Automatic validation of code-improving transformations on lowlevel program representations”, *Science of Computer Programming*, 52:257–280, Aug. 2004

[154] Heiko Falk; Jens Wagner; Andre Schaefer, “Use of a Bit-true Data Flow Analysis for Processor-Specific Source Code Optimization”, *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, Digital Object Identifier 10.1109/ ESTMED. 2006. 321286, pp. 133 – 138, Oct. 2006.

[155] S. Steinke, L.Wehmeyer, B.-S. Lee, and P. Marwedel, “Assigning Program and Data Objects to Scratchpad for Energy Reduction”, *Design, Automation and Test in Europe (DATE)*, pp. 409–417, 2002.

[156] L. Wehmeyer, U. Helmig and P. Marwedel, “Compiler-optimized usage of partitioned memories”, In *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*, Munich, Germany, pp. 114-120, June 2004, DOI=10.1145/1054943.1054959.

[157] M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, “Dynamic Management of Scratch-Pad Memory Space”, In *Proceedings of the 2001 ACM Design Automation Conference(DAC)*, June 2001.

[158] O. Avissar and R. Barua. “An Optimal Memory Allocation Scheme for Scratch-Pad-Based Embedded Systems”, *ACM Transactions on Embedded Computing Systems,* Vol.1(1), pp. 6–26, November 2002.

[159] J. Cho, Y. Paek, and D. Whalley, “Fast Memory Bank Assignment for Fixed-Point Digital Signal Processors”, *ACM Transactions on Design Automation of Electronic Systems*, vol.9(1), pp.52-74, 2004.

[160] M. A. R. Saghir, P. Chow, and C. G. Lee, “ Exploiting DualData-Memory Banks in Digital Signal Processors”, In *Proceedings of the SIGPLAN’96 International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 234-243, 1996.

[161] Q. Zhuge, B. Xiao, and E. H. M. Sha "Exploring Variable Partitioning for Dual Data-memory Bank Processors", In *Proceedings of the 34th International Symposium on Microarchitecture*, pp. 42–55, 2001.

[162] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient utilization of Scratch-pad memory in embedded processor applications", In *European Design and Test Conference,* March 1997.

[163] Darren Lee Buttle, *Verification of Compiled Code*, thesis*,* University of York, January 2001.

[164] Robert van Engelen, David Whalley, and Xin Yuan, " Validation of code-improving transformations for embedded systems", In *proceedings of the 8th ACM Symposium on Applied Computing SAC 2003*, Melbourne Florida, pp. 684–691, March 2003.

[165] Microchip Technology Inc. "PICmicro mid-range MCU family reference manual," 1997.

[166] Microchip Technology Inc. Data sheet, PIC16F87X, http://www.microchip.com.,1999.

[167] T. Sridhar and S.M. Thatte, "Concurrent checking of program flow in VLSI processors," *Proceedings of the 12th Int. Test Conf.*, pp. 191-199, November 1982.

[168] M.S. Hecht. '*Flow Analysis of Computer Programs'*. Elsevier Computer Science Library: Programming Language Series. North-Holland Publishing Co., 1977.

[169] J. P. Tremblay and R. Manohar, '*Discrete Mathematical Structures with Applications to Computer Science'*, McGraw-Hill, Singapore, 1987.

[170] Noel Jerke, '*Visual Basic 6: the complete Reference'*, Tata McGraw-Hill, New Delhi, India, 2000.

[171] Mariamma Chacko, K. Paulose Jacob, C.S. Sridhar, K.G. Balakrishnan, "A Novel Clustering Approach to Support Software Fault Tolerance", An *International Journal of Information Science and Technology*, vol.3, No.4, July 1994.

[172] Flunn M.J., "Very High Speed Computing Systems", *Proc. IEEE*, Vol. 54, pp. 1901-1909, 1966.

[173] MCS-80/85 family user's manual, Intel corporation, Santaclara, CA-95051, 1986.

[174] Willam R. Simpson and John W. Shepherd, "System Complexity and Integrated Diagnostics", *IEEE Design and Test of Computers*, pp. 16-30, Sept. 1991.

[175] Clarke E. M. and Emerson E. A., "*Design and synthesis of synchronization skeletons using branching time temporal logic*", In '*Logic of Programs*', volume 131 of Lecture Notes in Computer Science, Springer, 1982.

[176] Ball T., Larus J.R., "Efficient path profiling", In *Proc. IEEE/ACM Int. Symposium on Microarchitecture*, Paris, pp. 46-57, 1996.

[177] David M. Alter, "Online Stack Overflow Detection on the TMS320C28x DSP", Texas Instruments, *Application Report*, May 2003.

[178] Edited by Y.N. Srikant and Priti Shankar, '*The Compiler Design Handbook-Optimization and Machine code Generation'*, CRC Press, 2008.

[179] Freescale. *http://www.freescale.com*.

[180] Nazhandali L., Minuth M., Zhai B., Olson J., Austin T., and Blaauw D., "A second generation sensor network processor with application-driven memory optimizations and out-of order execution", In *Proceedings of the international Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'05),* ACM Press, New York, pp. 249–256, 2005.

[181] Hempstead M., Wei G., and Brooks D., "Architecture and circuit techniques for low throughput, energy-constrained systems across technology generations", In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'06)*. ACM Press, New York, pp. 368–378, 2006.

[182] *Data Sheet, PIC18F2455/2550/4455/4550*, Microchip Technology Inc., 2004.

[183] http://www.microchip.com.

[184] Ryser H. J., "Combinatorial Properties of Matrices of Zeros and Ones." *Canad. J. Math*, vol.9, pp. 371-377*.,* 1957.

[185] Ryser H.J., Traces of matrices of zeros and ones, *Canad. J. Math.* vol.12, pp. 463–476, 1960.

[186] Alexander Barvinok, "On The Number Of Matrices and a Random Matrix With Prescribed Row and Column Sums and 0-1 Entries, Journal of Advances in Mathematics - ADVAN MATH , vol. 224, no. 1, pp. 316-339, 2010.

[187] S. E. Wright, "Integer matrices with constraints on leading partial row and column sums", *Elsevier journal of Discrete Applied Mathematics*, Vol. 158, pp. 1838-1847, 2010, doi:10.1016/j.dam.2010.06.010.

[188] Fehnker A., Huuck R., Schlich B. and Tapp M. "Automatic Bug Detection in Microcontroller Software by Static Program Analysis", SOFSEM 2009, Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science, Czech Republic, January 2009, pp. 267-278.

LIST OF PUBLICATIONS

1. Mariamma Chacko and K. Poulose Jacob**,** "Validation of Microcontroller Codes: An Architecture Oriented Approach", accepted for publication in *Emerald International Journal of Intelligent Computing and Cybernetics*, ISSN: 1756-378X, 2011.
2. Mariamma Chacko and K. Poulose Jacob, "A Compiler Integrated Assistance for Optimum Data Allocation in Banked Memory Embedded Processors***",*** accepted for publication *in SERSC International Journal of Software Engineering and Its Applications*, ISSN: 1738-9984, 2011**.**
3. Mariamma Chacko and K. Poulose Jacob, "Optimization of Bank Switching Instructions in Microcontrollers having Partitioned Memory Architectures", *CiiT International journal of Software Engineering and Technology*, Vol.1, No.3, pp. 120-126, June 2009.
4. Mariamma Chacko and Poulose Jacob, "Validation of Embedded Software through Static Analysis of Machine Codes", *in Proc. of IEEE International Advance Computing Conference*, Patiala, India, , pp. 488-493, March 2009.
5. Mariamma Chacko and Poulose Jacob , "Optimization of Bank Switching Instructions in Embedded Systems through Static Analysis of Machine Codes*", in Proc. of IEEE International Advance Computing Conference*, Patiala, India, pp.548-552, March 2009.
6. Mariamma Chacko, James Kurian , P.R.S. Pillai and Poulose Jacob K, "An On Board Operation Support Information System and Data Logger for Sea Going Vessels with an Ethernet Interface", *Journal of Shipstechnic*, Vol XVIII, pp. 85-94, 2002.

7. Mariamma Chacko, James Kurian and Paulose Jacob K., “Design and Implementation of a Microcontroller Based Onboard Cockpit Display and Data Logger for Sea Going Small Crafts”, *Proceedings of the International Conference on Ship and Marine Technology*, p123-130, December 2000.
8. Mariamma Chacko, K. Paulose Jacob, C.S. Sridhar and K.G. Balakrishnan, “A Novel Clustering Approach to Support Software Fault Tolerance”, *International Journal of Information Science and Technology*, vol.3, No.4, July 1994.
9. K.G. Menon, Mariamma Chacko, Babu P. Anto and C.S. Sridhar , “A Microprocessor based position controller for a Laser optical grating”, *TENCON’89 4th IEEE region 10, International Conference*, 1989.
10. Leena Thomas, Mariamma Chacko, Babu P. Anto and C.S. Sridhar, “Hardware Implementation of FFT-8086 based system”, *TENCON’89 4th IEEE region 10, International Conference,* 1989.
11. Rajkumar K., Jayakrishnan V., Lorance K.M., Mariamma Chacko, Babu P. Anto and C.S. Sridhar, “A Universal Fully Programmable 16-Channel data acquisition system”, *National symposium on Instrumentation(NSI-13)*, 1988.

Index

Abstract interpretation 40
A-Cluster 82, 83, 85, 86
active memory bank 14, 98, 101, 123, 127, 135, 136, 139, 152, 153, 156
ADC 18, 94, 95, 100, 102, 107, 108, 109, 114, 118, 121, 139, 140, 148, 150
Address profiles 49
addressing modes 87
algorithm 13, 14, 31, 32, 36, 67, 75, 76, 82, 86, 87, 99, 116, 119, 122, 123, 125, 131, 132, 133, 135, 139, 140, 142, 143, 150, 152, 153, 154, 156
antecedent 74, 91, 95, 108
architecture 2, 10, 14, 22, 25, 47, 53, 56, 57, 66, 68, 71, 77, 80, 89, 97, 110, 118, 120, 122, 127, 128, 143, 151, 152, 155, 156, 157
Artificial intelligence 29
ASIC 13
assembler6, 26, 27, 35, 53, 61, 68, 75, 80, 82, 89, 91, 93, 96, 107, 109, 111, 115, 136, 139, 140, 155
assembly language 2, 5, 11, 15, 21, 25, 26, 43, 47, 67, 94, 112, 123, 159
bank selection 4, 14, 67, 101, 122, 123, 124, 125, 126, 127, 128, 132, 135, 154, 156, 157
Bank switching 4, 129
banked memories 4, 67, 124
B-Clusters 83
bidirectional ports 101, 114
binary executables 38, 45, 69
bugs 3, 5, 18, 31, 33, 35, 37, 39, 40, 41, 42, 47, 48, 57, 96, 99, 109, 118, 153, 155
CASE 22
C-Clusters 84
CFG14, 59, 67, 71, 72, 73, 74, 75, 76, 78, 82, 87, 89, 90, 93, 94, 95, 96, 120, 123, 132, 133, 134, 135, 136, 137, 145, 148, 150, 153
CISC 45, 87
clustering 81, 82
code optimization 2, 3, 6, 11, 62, 63, 68, 81, 122, 127, 132, 152
code portability 25, 26
code redundancy 3, 80, 147
code sequence 14, 73, 74, 75, 80, 82, 90, 93, 95, 106, 109, 112, 116, 120, 129, 131, 132, 148, 153, 156
code validation 2, 5, 13, 14, 80, 81, 88, 90, 94, 99, 110, 111, 117, 118, 120, 151, 152, 153, 154
Codification 71, 73, 79, 80, 90, 153
compilation 11, 12, 23, 36, 53, 61, 68, 69, 125
compiler 2, 4, 7, 11, 15, 22, 26, 27, 37, 39, 40, 44, 48, 57, 60, 62, 63, 64, 65, 66, 68, 75, 80, 81, 87, 89, 96, 98, 116, 119, 121, 122, 125, 131, 139, 141, 143, 146, 148, 154, 155, 157, 159
compliance 74, 80, 90, 95, 152
consequence 53, 95
constant folding 11, 63
Constraints 1, 5, 17, 18
control flow13, 26, 36, 39, 43, 44, 45, 49, 58, 59, 60, 69, 71, 72, 73, 80, 87, 88, 89, 93, 117, 133, 148, 150, 155
conversion time 118
CPU 12, 14, 18, 53, 77, 81, 82, 101, 122, 127, 153

cross compiler 6, 20, 27, 53, 81
data allocation 76, 122, 143, 144, 145, 146, 147, 150
data diversity 28, 29
dataflow analysis 5, 48
deadlock 73, 109, 116, 117
debugging 2, 3, 6, 10, 12, 13, 15, 20, 23, 24, 26, 30, 31, 32, 33, 34, 35, 36, 37, 38, 50, 51, 52, 54, 55, 56, 58, 69, 75, 79, 82, 86, 99, 107, 118, 120, 151, 152, 153, 154, 155, 157, 158
decompilation 44
dependence profile 49
design diversity 28, 29
desktop systems 2
destination register 82, 101, 112
disassembler 38, 43, 44, 45, 46, 80, 93, 118, 120, 156
discrepancy 81, 106, 109, 110, 118, 153
discrete function 129
Dynamic techniques 155
efficient code 11, 63, 144, 148
elimination 9, 11, 14, 63, 66, 90, 119, 122, 157
embedded processor 2, 11, 25, 27, 63, 68, 95, 125, 158
embedded software 1, 5, 6, 9, 10, 13, 14, 15, 18, 20, 21, 23, 34, 41, 47, 48, 50, 51, 69, 79, 80, 122, 132, 153, 158
embedded systems 1, 2, 5, 11, 12, 13, 17, 18, 25, 26, 27, 58, 59, 69, 79, 80, 97, 121, 122, 139, 148, 152
entry nodes 150
error localization 151
exception handling 52
executables 3, 13, 38, 42, 43, 47, 90, 148
fault avoidance 58
fault correction. 99
fault diagnosis 31, 99, 109, 120
fault localization 13, 14, 30, 32, 34, 50, 99, 120
fault removal 58
fault tolerance 3, 28, 58
feasibility 55, 71, 97, 120, 126, 153, 154, 157
flag 81, 83, 84, 85, 86, 101, 111, 112, 148
FLASH program memory 97
flow array 90, 150
FPGA 12, 158
Function inlining 11, 62
functional diversity 28
governing rules 78, 100, 101, 109, 117, 152, 153
hex code 80, 92, 107, 114, 129
high level language 3, 5, 15, 21, 26, 38, 47, 107, 159
HI-TECH C 107, 115, 139
host 2, 7, 20, 22, 27, 30, 52, 53, 54, 55, 56, 80, 152
IDE 7, 31
illegal opcodes 52, 81, 110
in circuit debuggers 5
in-circuit emulator 6, 20, 51, 54, 55
indirect addressing 43, 101, 113, 155
initial node 89, 95, 134
instruction set 30, 61, 64, 73, 81, 83, 87, 90, 98, 100, 116, 126, 153
instruction stream 14, 59, 74, 82, 87, 98, 99, 100, 102, 109, 120, 151, 152, 154, 156
integrated peripheral 73, 81, 97, 120, 153
Intel hex file 14, 72, 89, 90, 96, 107, 120, 132, 143

interprocedural routines 14, 123, 135, 150, 152, 154, 156
interrupt vector 97
intraprocedural routines 76, 134, 136
invalid sequence 93
JTAG 13, 30, 51, 55, 56
knowledge base 82, 85, 99
leaf nodes 88, 135, 138, 150
linear scan 43, 76, 106, 134, 148
load-store architecture 87
logic analyzer 13, 54, 55
logic errors 12
loop unrolling 11, 62
loops 14, 123, 135, 148, 150, 152, 154, 156
malfunctioning 69, 71, 81, 155
malicious code 43, 44, 45, 47, 48
memory bank 11, 14, 67, 73, 75, 76, 98, 111, 122, 123, 124, 126, 127, 128, 129, 130, 131, 132, 134, 135, 141, 142, 143, 144, 146, 150, 152, 154, 156, 158
merge node 72, 74, 76, 88, 89, 90, 93, 94, 95, 132, 134, 148, 150
Microchip Technology 94, 97, 139
mikroC 107, 115, 139, 140
model checking 5, 119, 152, 155
Opcodes 109, 110
optimization 2, 4, 5, 8, 9, 11, 13, 14, 15, 21, 27, 37, 49, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 71, 76, 78, 87, 121, 122, 123, 126, 128, 139, 141, 143, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158
optimum data allocation 15, 76, 143, 144, 146, 152, 154
orthogonal 98
oscilloscope 53
partitioned memory architecture 11, 13, 122, 150, 154, 156, 157
path quantifier 92
permutation 76, 143, 144
precedes 86, 91, 93, 105
premises 74, 91, 92
Previously Activated Memory Bank 75, 127, 129, 130
program analysis 5, 24, 40, 42
program counter 97
program dependence graph 36, 87
program graph 72, 88, 89, 90, 93, 94, 107, 132, 136, 149
Program slicing 31, 36, 43
programming languages 18, 28, 40, 64, 76
propositional logic 74, 80, 82, 91, 95
PROTEUS VSM 77, 139, 141
prototype 6, 14, 31, 33, 44, 47, 74, 82, 99, 120, 150
real-time 9, 10, 11, 21, 22, 41, 54
reasoning 33, 34, 50, 82, 100
recovery blocks 28
redundant codes 4, 15, 119, 122, 125, 135, 136, 139, 141, 144, 145, 146, 148, 149, 150
registers 11, 51, 53, 56, 60, 63, 64, 73, 75, 87, 95, 97, 98, 99, 101, 102, 109, 110, 111, 124, 127, 128, 143, 144, 146, 147, 148
relation matrix 14, 75, 78, 122, 126, 129, 131, 132, 139, 148, 150, 152, 153, 156
reset vector 97
reverse-engineering 44
RISC 2, 13, 14, 45, 71, 79, 87, 97, 110, 116, 119, 120, 128, 143, 151, 153
rules of inferences 74, 80, 81, 90, 119, 153, 155
run time 80, 81, 119, 141, 156, 157, 158

screenshot 136
semantic errors 81
simulators 6, 8, 23, 27, 110
sleep 100, 102, 103, 105, 106, 107, 108
snapshot 31, 99
software development 3, 10, 13, 17, 20, 21, 23, 24, 25, 26, 34, 47, 51, 80, 81, 121, 122, 155
Software fault tolerance 28
software testing 10, 13, 20, 53, 69
Soundness 74, 92
source code 3, 6, 11, 15, 34, 35, 39, 42, 44, 47, 60, 61, 64, 66, 68, 72, 80, 81, 87, 98, 146, 159
source node address 107, 136
Source-level Debugger 23
Special Function Registers 97, 127, 128
STAD 32
state space 93, 98, 119, 152, 155
static analysis 3, 6, 9, 13, 15, 38, 39, 40, 41, 42, 43, 45, 47, 48, 69, 75, 78, 80, 118, 122, 125, 127, 132, 148, 149, 151, 152, 153, 154, 155, 156, 158
static slicing 39, 90
status flags 83
status register 97, 111, 127, 128
stipulated 15, 74, 90, 91, 95
subprogram 36, 73, 76, 88, 89, 90, 93, 95, 132, 134, 135, 136, 137, 138, 157
succeeds 91, 93
symbolic debugger 31, 35
System Dependence Graph 36
system integration 52, 57
System on Chip 1, 12
target processor 4, 19, 27, 51, 53, 55, 61, 71, 72, 73, 74, 80, 82, 90, 91, 94, 97, 98, 106, 119, 120, 126, 131, 141, 144, 153, 154, 155
temporal logic 92, 110
Testing and debugging 1, 34, 79
time critical task 2
tool chain 96
transparent nodes 76, 135, 156
tree 31, 46, 67, 73, 88, 132
universal set 91
validation9, 10, 14, 15, 23, 24, 47, 57, 58, 68, 71, 74, 75, 76, 78, 79, 80, 87, 93, 96, 100, 107, 109, 116, 119, 120, 151, 152, 153, 155, 156, 158
Value profile 49
Variable Partitioning 121, 141
verification 3, 21, 23, 24, 28, 33, 44, 46, 57, 58, 81, 119
vertices 82, 87
VHDL 12
VHSIC 12
Visual Basic 14, 75, 76, 107, 116, 136, 149
warnings 76, 93, 107, 111, 117, 123, 134, 135, 137, 139, 156, 158