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
Introduction

Cochin University of Science and Technology

This is then assembled by the cross-assembler/compiler running on the host computer. If the programmer has access to a simulator then one may choose to test the program by simulation. This is likely to lead to program errors being discovered, which will require alteration to the original source code. When satisfied with the program, the developer will then download it to the program memory of the microcontroller itself, using either a standalone programmer linked to the host computer or a programming facility available in the embedded system itself. Then the program running in the actual hardware is tested. Again, this may lead to changes being required in the source code.

The various stages and their functions are shown in Fig. 1.1.

Fig. 1.1 Possible stages in the development process for the program of a simple embedded system project.

The different software tools are usually bundled together into what is called an Integrated Development Environment (IDE). It contains all the software tools...
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.