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
Chapter 2

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.
The target processor of an embedded computer is typically minimal in function and size. For consumer and portable applications, a combination of cost, size and power consumption considerations may result in the quantity of memory.
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 a 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 \( \text{var} \), and a location \( \text{loc} \) 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 \( \text{var} \), a location \( \text{loc} \), 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. Decomposition [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.

![Diagram](image)

**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 interleave 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\text{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 inbuilt 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 footprint, 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
Chapter 2

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
Chapter 2

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].
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 (Single 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 \textit{compiler known functions} or \textit{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.