Towards Target-Level Testing and Debugging Tools for Embedded Software

Harry Koehnemann, Arizona State University
Dr. Timothy Lindquist, Arizona State University

Abstract

The current process for testing and debugging embedded software is ineffective at revealing errors. There are currently huge costs associated with the validation of embedded applications. Despite the huge costs, the most difficult errors to reveal and locate are found extremely late in the testing process, making them even more costly to repair. This paper first presents a discussion of embedded testing research and practice. This discussion raises a need to improve the existing process and tools for embedded testing as well as enable better processes and tools for the future. To facilitate this improvement, architectural and software capabilities which support testing and debugging with minimal intrusion on the executing system must be developed. Execution visibility and control must come from the underlying system, which should offer interfaces to testing and debugging tools in the same manner it offers them to a compiler. Finally we propose extensions to the underlying system, which consists of additions to both the architecture and run-time system that will help realize target-level tools.

1. Introduction

Software validation involves many activities that take place throughout the lifecycle of software development. A substantial portion of the validation process is software testing, which is the development of test procedures and the generation and execution of test cases. Notice we are not only concerned with the generation of a test case, but are also concerned with how that test is executed. Therefore, a test case is not simply composed of inputs to a system, but also includes any environmental factors. Other research has examined the issues behind test case selection, but few are addressing the problems that surround the execution of those test cases. The goal of this paper is to identify the problems associated with test case execution for embedded systems and to propose solutions for making embedded testing more effective at revealing errors.

1.1 Testing and Debugging Process

Many of the activities, tools, and methods used during software testing are shared by software debugging. Software testing is concerned with executing a piece of software in order to reveal errors, while software debugging is concerned with locating and correcting the cause of an error once it has been revealed. Though these two activities are often referenced separately, their activities are tightly coupled and share many common features.

During debugging, a developer must recreate the exact execution scenario that revealed the fault during testing. Not only must the code execute the same instruction sequences, but all environmental variants must be accounted for during the debugging session. In addition, the tools assisting in the debugging process must providing a developer with a certain degree of execution visibility and control while not impacting the execution behavior of the program.

1.2 Embedded Systems

The testing and debugging process is greatly restricted by embedded systems. Embedded applications are among the most complex software systems being developed today. Such software is often constrained by

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

Each of these properties of embedded software severely restrict execution visibility and control, which consequently restricts the testing and debugging process.

Our current methods and tools for software testing and debugging require a great deal of computing resources. Such resources are not available on the target environment. Therefore, a large gap exists between the methods and tools used during evaluation on the host and those used on the target. Unfortunately, many errors are only revealed during testing in the target environment.

Because of the above issues, concerns are raised over the effectiveness of software validation for embedded systems. Embedded applications are responsible for controlling physical devices and their correct execution is critical in avoiding and/or recovering from device failure. Often these physical devices control life-critical processes, making the embedded software a key element of a life-critical system. Software failure can lead to system failure, which in turn could lead to loss of life.

In addition, designers are increasing their use of embedded software to control the physical elements of large systems. This rate of increase is likely to increase as the cost for embedded controllers becomes cheaper and more attractive when compared with other mechanical techniques. Computer networks are fast replacing point-to-point wiring, due to the networks light weight, easy configurability and expansibility, and lower design complexity. The advancement in the complexity of problems addressed by software in these types of

applications may soon be limited by our inability to satisfy reliability needs and concerns.

2. Software Testing

The software testing phase is concerned with executing a software program in order to reveal errors. Software testing for embedded systems takes place in four basic stages:

1) Module Level Testing
2) Integration Testing
3) System Testing
4) Hardware/Software Integration Testing

The first three testing stages are typical of any software product. Testing begins with exercising each software module and concludes when the software is shown to meet system specifications by passing some rigorous set of system tests. The fourth phase is unique to embedded systems. The software must not only be correct, but must also interface properly with the devices it is controlling.

Testing literature contains countless methodologies, techniques, and tools that support the software testing process. They range from software verification and program proving to random test case selection. All testing methods indirectly apply to embedded systems, as they do all software. Of particular interest to this paper are those techniques that address the problems identified for embedded software - concurrency, real-time constraints, embedded environment, etc. Unfortunately, there exists little research into the unique problems associated with testing embedded software.

2.1 Testing Concurrent Systems

Concurrency increases the difficulty of software testing. Given a concurrent program and some set of input, there exists an unmanageably large set of legal execution sequences the program will take. Furthermore, subsequent execution with the same input may yield different, yet correct results due to differences in the operating environment. This is all complicated by Ada's non-determinstic select construct. Therefore, when testing concurrent software, we are not only concerned with a valid result, but must also be concerned with how the program arrived at that result.

Since multiple executions of a concurrent program may yield different results, it is not enough to ensure that the system produces the correct output for a given input. One must also ensure that the system always produces an acceptable output for each execution sequence that is legal under the language definition. Without sufficient control over program execution, there is no way of ensuring a given test is exercising the code it was intended to test.

Taylor and Osterweil [Tayl80] examined static analysis of concurrent programs. However, this research considered processes in isolation and does not consider interprocess communication. Taylor later extended this work to Ada and a subset of the Ada rendezvous mechanism [Tayl83]. Through this static analysis technique, one could determine all parallel actions and states that could block a task from executing. This method, as with most static techniques, must examine a large set of states and therefore must constrain itself to small, simple programs.

Research in dynamic testing of concurrent Ada programs has largely focused on the detection of deadlocks [Hemb85], the saving of event histories [LeDo85, Maug85], and other techniques that passively watch a program execute then allow the execution sequences to be replayed after a failure has been detected.

Hanson [Hans78] was among the first to discuss run-time control of concurrent programs. In order to regulate the sequences of events, he assigned each concurrent event in the test program a unique time value. He then introduced a test clock that regulated the system during execution. A given event could only execute if it's time was greater than that of the clock.

Tai [Tai86, Tai91] extended Hanson's work to the Ada programming language. His method takes an Ada program P and a rendezvous ordering R and produces a new Ada program P' such that the intertask communication in P' is always R. A similar approach was used in [Koeh89] to apply these techniques to testing and debugging tools. This work addressed the fact that in order to test a specific program state, values in a program may need to be modified during run-time. Modification of the program state is a capability provided by any debugging tool and is a required property of a tool debugging tasked programs. It is important to note that both techniques explicitly perform rendezvous scheduling, removing those decisions from the run-time system and placing control in the hands of the tool.

2.2 Non-intrusive testing

Intrusion plays a significant role in the testing and debugging of embedded software. Any technique used to raise execution visibility or provide for program control must not interfere with the behavior of the test program. Embedded applications have strict timing requirements and any intrusion on a test execution will likely make that test void. Intrusion is typical for host-based testing, but becomes a large problem for target-level testing and debugging activities.

The above approaches address the need for visibility, control, and predictability for testing concurrent software. However, they are all intrusive and use instrumentation (inserting probes into a users program and rewriting certain constructs before submission to the compiler) to gather run-time information and to control program execution. After the probes are added, the user's object code is linked with the rest of the debugging system and then executed under test. This additional code has a serious impact on the execution behavior of the program. Instrumentation is not appropriate for testing real-time, embedded applications.

A non-intrusive debugger for Ada is proposed in [Gill88]. A separate processor executes the testing system and communicates with the target processor through some special purpose hardware. Lyttle and Ford [Lytt90] have also implemented a non-intrusive embedded debugger for Ada. Their tool provides monitoring, breakpoints, and

display facilities for executing embedded applications. While these efforts provide an excellent start towards target-level tools, they do have severe limitations. These implementation do not deal with high level activities such as task interactions and are only concerned with items that can be translated from monitoring system bus activity. As discussed later in Chapter 5, techniques dependent on bus activity will likely fail for future architecture designs. In addition, many of the error detected in the target environment are indeed concerned with high-level activities (process scheduling and interactions, fault handling, interrupt response, etc.).

Other real-time, embedded tools have been proposed for cross-development environments. They can typically be classified into one of the following three categories: 1) ROM monitors, 2) Emulators, and 3) Bus monitors. These types of tools will be further discussed later in this paper.

2.3 Impact of the Underlying System

One of the large problems with testing concurrent systems is dealing with abstraction. The Ada programming language abstracts concurrent activities through task objects [DOD83]. Tasks allow a developer to abstract the concepts of concurrency and interprocess communication and discuss them at a high level. The burden of implementation is then placed on the compiler, and typically the run-time system.

While abstraction is a powerful design tool, it leads to significant problems during the testing phase of software development. Implementation details become buried in the underlying system. At the development level, this high degree of abstraction is appropriate. However, abstraction complicates the testing process. Not only are we concerned with implementation details, but we must also control them to demonstrate that certain properties about a program will hold for every legal execution scenario. Without sufficient control over program execution, there is no way of ensuring that a specific test is exercising the code it was intended to evaluate. In addition, correct operation in one environment (host) does not necessarily imply correct operation in another (target) due to implementation difference in the underlying system.

The underlying system is composed of two parts, the features of the hardware architecture and the operations provided by the run-time system. As language constructs become more abstract, compilers are required to generate more code to implement them. There is no longer a trivial mapping from language construct to machine instruction. Rather, the compiler must provide an algorithmic solution in order to implement these high level constructs. Those solutions exist as operations in the run-time system. Rather than generate code for these constructs, the compiler generates a call to a run-time system operation or service.

As the constructs become more abstract, compilers develop an increasing dependency on the underlying system. This increase in shown in figure 2.1. As new constructs are introduced to programming languages, their increase in abstraction is greater than that of hardware and the run-time system is called upon to bridge the impending gap. No argument is made as to the rate of increase identified by the line slopes; nor is an argument made that these increases are even linear.

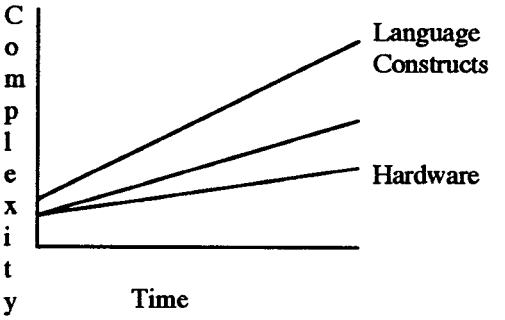

Figure 2.1

3. Embedded Testing

Embedded systems raise many problems for software testing and debugging. Such software typically must deal with concurrency, real-time constraints, an embedded target environment, distributed hardware architectures, and a great deal of hardware-software interfaces for controlling external devices. These issues alone do not provide a complete view of the problems encountered by embedded testing. Embedded systems are typically developed on custom hardware configurations meaning that each system introduces it's own unique problems. Tools and techniques that apply to one are not generally applicable on another, which leads to ad hoc approaches to integration and system testing of embedded software. The program is executed for some length of time and continually bombarded with inputs in an attempt to show it adheres to some specification.

3.1 Current state of embedded testing

As described earlier, the testing process for embedded systems consists of 4 phases that conclude with Hardware/Software (H/S) Integration During H/S integration testing, device and timing related errors are revealed. These errors encompass problems such as:

- incorrect handling of interrupts
- distributed communication problems
- incorrect ordering of concurrent events
- resource contention
- incorrect use of device protocols and timing
- incorrect response to failures or transients

These errors are often extremely difficult problems to fix and often require significant modifications to the software system. In addition, software is forced to conform to custom hardware that may itself have errors. As stated above, H/S integration is the last phase of testing for an embedded system. Since errors are much cheaper to fix the earlier they are revealed,. why would one wait until the last phase of product development to reveal the most difficult to

locate, costly errors to fix? Our goal should be to reveal these errors as early as possible. Unfortunately, target level testing tools have yet to become a reality.

The target processor of an embedded computer is typically minimal in function and size. It is only a small portion of a larger system, whose goals are to minimize cost and space. Therefore, target hardware of an embedded systems will not support software development nor any development tools. To resolve this problem, the source is developed on a larger host platform and cross compilers and linkers are used to generate code and download it to the target processor.

Consequently, two environments exist in our development process, the host environment and the target environment, each having completely different functionality and interface to a user. 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.

3.2 Current Solutions

Approaches to dealing with the above problems 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. A bus monitor gains visibility of an executing program by observing data and instructions transferred across the system bus. With a ROM monitor, debugger code is placed into ROM on the target board. When a break point is encountered, control is transfered to the debug code which can accept commands from the user to examine and change the program's state. Finally, an in-circuit emulator connects with a host system across an ethernet connection. At the other end, a probe replaces the processor on the target board. The emulator then simulates the behavior of the processor in (ideally) real-time, which allows the emulator to tell the outside world what it's doing while it's doing it.

The hardware solutions have minimal 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. That mapping is the implementation strategy chosen by a given compilation system and becomes severely complicated for abstractions such as tasks Maintaining an understanding of the mapping is extremely difficult and cumbersome.

The software solutions can be viewed as attempts to reduce the tremendous costs of testing on the target. Several factors determine how a piece of software is tested:

1) Level of criticality of software module
   Each software module is assigned a different level of criticality based on it's importance to the overall operation of the system.
2) Test platform availability
   Typically, there will exist several test environments available to test a piece of software, each providing a closer approximation to the actual target environment:
   - Host-based source level debugger
   - Host-based instruction set simulator
   - Target emulator
   - Integrated validation facility
3) Test Classification
   The tests to be performed can be categorized to determine what they are attempting to demonstrate. The goal of a test plays a large role in determining the platform on which it will execute. Some examples are shown below.
   - Inter-module
   - Intra-module
   - Inter-cabinet
   - Algorithmic
   - Performance
   - H/S integration

Each of these factors play a role in assigning program modules to the various test platforms based on some criteria that might contain the following:

- Type of software
- Hardware requirements
- Test classification
- Platform availability
- Coverage requirements
- Test support software availability (drivers, stubs)
- Certification Requirements
- Level of effort required for test

This criteria takes into account the 3 factors discussed above as well as additional ones.

The software solutions are an attempt to minimize the time spent testing in the target environment. Validation facilities are expensive to build and time utilized for testing is expensive. This is due to the fact that target level testing occurs extremely late in the development lifecycle and only a small window is allocated for H/S integration testing. However, the target is the only location that can reveal certain errors. It is ironic that our current solutions attempt to reduce the amount of target testing, but will likely lead to extensive modifications and therefore extensive retesting.

4. Problems with Embedded Testing

The solutions proposed above are not effective at revealing errors. Effective implies that a technique reveals a high percentage of the errors and that it does so in a cost-efficient manner. Instead, what the above tools provide is a minimal, low-level view of the execution of a program and those tools become available at a very late stage in development. Below is a list of problems associated with current approaches to embedded testing:

4.1 Expense of Testing Process

Target testing requires expensive, custom validation facilities. The expense of these target facilities is incurred for every project, since little reuse across projects is ever realized. The effort required to build these validation facilities means that every test execution is expensive, making retests extremely costly. Yet, hardware often arrives late and full of errors, forcing software to be

modified and subsequently retested. This late arrival of hardware also impacts the cost of an error, since certain errors are only revealed during H/S integrations testing.

Perhaps the largest factor associated with the high costs of testing will be the questions and concerns that certification processes are beginning to raise about software tools. Typically, development tools have not been required to meet any validation criteria and certainly not the strict criteria imposed on the development system. This luxury may soon disappear as the role tools play in the development process comes under tighter scrutiny. The huge expense of validation facilities will increase dramatically.

4.2 Level of Functionality on Target

The level of functionality found on a target machine is minimal and does not support tools. This lack of functionality greatly limits the effectiveness of testing, since more time and effort is required to locate an error. While a host system provides a high-level interface and discussed software in terms of the high-level language, the target typically deals in machine instructions and physical addresses. Translating these low-level entities requires time and a great deal of tedious, error-prone activities.

4.3 Errors revealed late in development lifecycle

Embedded system designs often incorporate custom ASIC parts that are typically not available until very late in the development process, delaying the availability of any target validation facility. In addition, errors designed into the ASICs are extremely expensive to fix, requiring new masks be created and complete refabrication. Instead, errors in ASICs and other hardware problems are resolved by modifying the software. As stated before, this greatly delays the time which errors are revealed, which in turn increasing the cost of software testing.

4.4 Poor test selection criteria

All to often, tool availability dictates the quality of a testing process. Tests cases and scenarios are determined by what will work on available platforms and which test are schedulable rather than being determined by some theoretical test criteria. A prime example is the FAA's requirements [FAA85] that 1) all testing be done in the target environment and 2) testing include statement coverage. Of course, test coverage is not currently measured on the target.

Unfortunately, it is cheaper for a company to spend it's resources preparing an argument to obtain some form of "waiver" than to actually perform a test. In time, the argument approach will no longer be accepted and the solutions for embedded testing must be in place to accommodate this change. It will only take one implementation that performs statement coverage on the target to force every embedded, real-time software developer to perform statement coverage on the target to meet such a certification requirement.

4.5 Potential use in advancing architectures

Perhaps the largest problem facing embedded testing is that the current solutions cannot be applied to future hardware architectures. Future architectures are proposing:

- wider address spaces
- higher processor speeds
- huge numbers of pins
- internal pipes
- multiple execution units
- large internal caches
- multi-chip modules

Such complexities cast a dark shadow over the hardware solutions previously discussed. With internal caching and parallel activity being done on the chip, one will no longer be able to gain processor state information from simply monitoring the system bus. And 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 [Chil91] an even stronger claim is made that the debugging capabilities provided by the chip will need to become more sophisticated. In future architectures, perhaps the only possibility to view and control the execution of hardware is to gain that information from the hardware itself.

5. Increasing Target Functionality

The previous sections raised issues about the effectiveness of our testing process and claimed that testing is currently being limited by tool functionality. The goal of this paper is to identify shortcomings in the embedded testing process and propose a solution to those problems. The view taken by the authors is that tool support for embedded systems is lacking. Further, those approaches currently used for gaining execution visibility and control will soon be obsolete for future architectures. We propose adding facilities to the underlying system to better support testing and debugging tools for embedded software.

As stated previously, the underlying system is composed of the hardware architecture and the run-time system (RTS). Both are composed of data structures and operations that implement common system abstractions such as processes, semaphores, ports, timers, memory heaps, and faults/exceptions. It should be noted that there is no distinct line between features of hardware and features of the RTS. In fact, as these features and abstractions become more standardized, newer architectures are attempting to incorporate them into their instruction sets [INTE92]. In addition, the implementation,of a feature may span parts of the architecture, RTS, and compiler generated code (i.e. faults/exceptions).

5.1 Model Debugging System

Below is an illustration of a debugging system (Figure 5.1). The data path from the debugging/testing tool represents symbol table information that allows the tool to map machine level information to source level

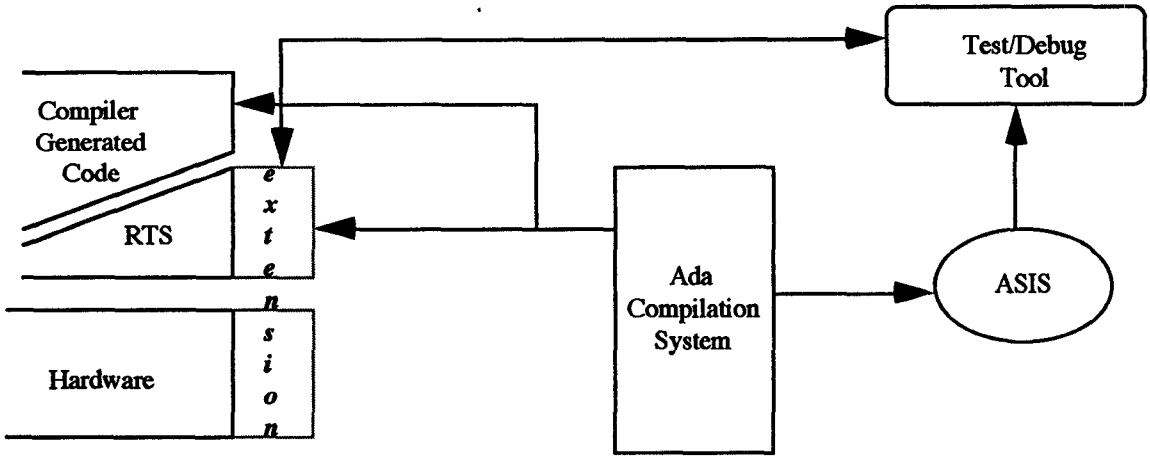

Figure 5.1

constructs. The ASIS toolkit provides easy implementation for this facility. ASIS is a proposed standard interface between an Ada library and any tool requiring compilation information.

Of more interest is the communication path between the target processor and the testing tool. A tool sits external to the rest of the embedded system, while the RTS resides internally on the target board. At first glance, this conceptual path seems rather difficult to realize. However, the implementation becomes easier if thought about as a typical host debugging system. Any debugging system has a least two processes executing, one running the test program and one running the debugger. These two processes share a common physical machine, which allows one process to gain information about the other. The debugger process simple requires data and computation time, which it shares with the test program.

This same scenario is required for embedded debugging, except that the debugger process is split. Part of the debugger process runs on the target machine and part runs on the host. The goal is to minimize the portion that must be run on the target so that it does not intrude on execution of the test program. To realize this non-intrusive execution of the debug software, the target processor can:

1) Execute debug code only at a break point,
2) Run the debugger as a separate process, or
3) Provide a separate execution unit to execute the debugger.

The details of these options are explored in depth later in this paper.

The problem now lies with the interface between the embedded part of the debugger (internal-debugger) and the portion that lies on the host (external-debugger). The solution requires hardware additions that will be discussed later in this paper. A high level view is given in figure 5.2. In this figure, the tool makes logical calls to services provided by the RTS. These calls are actually implemented by the debugging system through data passed between the internal and external debuggers. Hardware additions are required for this physical connection. The next two sections describe the architecture additions and RTS interfaces in more detail.

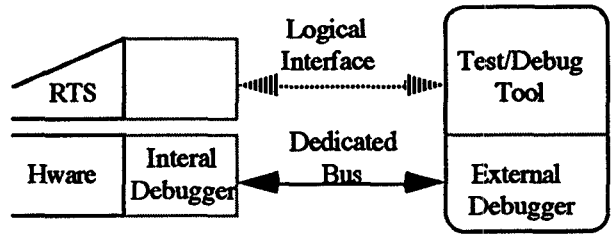

Figure 5.2

6. Architectural Additions

The past decade has seen huge advances in microprocessor designs. Several of these advancements were listed previously and include pipelining and separate functional units. The concept of partitioning a microprocessor in order to perform parallel activities is of great interest to this work.

It was noted earlier that these parallel computations severely restrict current methods for testing and debugging embedded systems, since one must simulate a great amount of computations. However, debugging tools can also use architectural parallelism to their advantage. If a hardware design is partitioned successfully to allow certain activities to occur concurrently, then the testing and debugging methodologies might wish to add

their own computational requirements to the list of parallel activities.

This section will explore additions to hardware architectures. No claim is made as to the costs associated with these features. They assuredly will require space (transistors) and possibly even add to the execution cycles required to implement certain instructions.

6.1 Hardware Partitioning of Memory

One primary concern for industry is reducing the huge volume of retests associated with development. The current testing process ensures that errors are revealed late, which forces retesting large portions of the system. Despite correcting these problems, industry will still be faced with software that is constantly changing. Software is deceivingly easy to change and often the element of a system assigned to unknown or "risky" aspects during design.

Changing software is extremely expensive late in the development for critical systems. Such systems typically have requirement that an error raised in one portion of the system won't interfere with the correct operation of the rest of the system. Current software certification agencies [FAA85] have several software restrictions including:

- Any modification made to a software module forces the retesting of all other modules operating on that same physical device.
- All software on a device must be developed under the highest level of criticality of any module that will execute on the same device.

Without the ability of hardware to guarantee software boundaries, such requirements must be enforced. However, these requirements add a great deal of costs to software development. Consequently, software is often physically partitioned based on critical level, rather than design factors.

Partitioning software modules based on critical levels greatly interferes with the design process. One would rather partition modules based on factors such as processor utilization and inter-module communication requirements. In fact, load balancing and process migration are techniques that would not be usable by embedded system developers unless all software is developed at the highest critical level.

The solution to these issues is hardware partitioning. Each process should have it's own protected address space that is not accessible by any other process. In addition, sets of processes may wish to share memory. The processor should also provide the capability to restrict access to segments of memory based on some criteria.

6.2 Computational Facilities for Debugger.

The debugging system is partitioned into an internal debugger and an external debugger. The internal debugger must physically exist on the target board and communicate with the external debugger through some dedicated medium. The internal debugger will also require execution from the target without interfering with the operation of the application program. There are two possible scenarios:

- The internal debugger runs as a regular process on the processor
- The architecture provides separate facilities to execute the internal debugger code

In either case, control is transfered to the debugger when a breakpoint is encountered.

In the first scenario, the debugger is executed by the processor as any other process. If the debugger executes as a low-level process, it would not interfere with the operation of the rest of the system. However, this is not a feasible approach. Most interesting errors occur during peak system loads, which would mean that the debugger could only execute when the probability of an error occurring was low. Another approach would be to execute the internal debugger as a periodic process of high priority and design the entire system to take this process into account when determining issues such as scheduling.

The second scenario requires the target processor to provide some form of computational facilities. This extra execution will certainly require some amount of utilization of architecture resources such as internal registers and bus accesses. The simplest example of architecture facilities would be a machine that continually dumps some representation of the instruction it is currently executing. This would require a dedicated bus to the external world (proposed later in this section) and that additional circuitry be attached to the computation units to gain access to the current instruction.

The problem with this approach is that the processor is not aware of what data is required by the tools at the other end. Therefore, it must dump everything. At high processor speeds, the amount of information being sent could become overwhelming. However, the data could be filtered and then captured so that a tool could parse it later and recreate an execution history of the program. The hardware required for filtering is not trivial and requires great speed and storage capacity to maintain pace with the target processor.

The next step is to allow software to dictate the information sent by the processor. The functional unit of the hardware sending messages could be implemented as a state machine, emitting different messages based on its current state. The default state would be all processor transactions. Basically, in this configuration, the processor is performing the filtering rather than the external debugger. This addition should not add much in complexity to the hardware architecture and would greatly reduce the complexity of the external debugging hardware.

The final step is to take the (now stateful) functional unit and make it programmable. Instead of a state machine, it now becomes a complete functional unit within the processor itself. The internal debugger code would then be loaded into this portion of the processor at boot time and reside there for the entire execution, transmitting and receiving messages to and from the external debugger.

6.3 Hardware Break Points

Software break points are intrusive and require instructions be inserted into the code of the test program. Conditional break points present a more significant problem, since they require a computation every time they are encountered to determine if the proper conditions are met to halt execution. Such breakpoints are unacceptable for real-time programs.

To resolve this issue, architectures need to provide the capability to set breakpoints in hardware. A set of registers would be classified as BreakPoint Registers (BPR), which the processor would check against the operands for each instruction. Two types of breakpoints are required, data and instruction. Each data BPRs inside the processor would be compared with the address of every data operand for each instruction. Instruction BPRs would be compared with instruction addresses or type. When a match occurs, a breakpoint fault would be raised and control transfered to the internal debugger

Upon returning from a break, the processor is required to restart execution precisely where it had terminated. The state of the processor consists of all it's internal registers, including any pipeline information and cache memory. These values must be saved automatically when a break is encountered.

Another issues is that of conditional breakpoints. Such breakpoints require computations by the processor that run in the background behind the program under test. The evaluation of the conditional expression must begin far enough in advance so that it may complete before the processor has passed the breakpoint location. This evaluation will require memory accesses, raising additional problems. The current value of operands in the expressions must be available to the processor, which might involve accessing it from memory or cache. Any accesses to memory must be scheduled in such a manner that they do not block any resources required by the program under test. Finally, the value used must be valid and not in danger of changing before the breakpoint.

While the problems raised above seem difficult, they are not insurmountable. The external debugger must compile the conditional expression and download the code. At that point it can determine the schedulability of this evaluation by comparing the code for the conditional to the other code that will occur in parallel. The user could then be notified of problems with their conditional breakpoint. The hardware is responsible for detecting any collisions in parallel activity and must not assume the debugger is always accurate. Any debugger activity intruding on the behavior of the test program is important information and must be flagged by the processor.

The primary additions required for hardware break points are additional registers from the architecture and the logic necessary to compare them with the operands of the current instruction. To support conditional breakpoints, the processor must provide background computational support. This support could come from a portion of the processor dedicated to conditional breakpoints, or the code could be downloaded to the internal debugger, given the internal debugger support described previously.

6.4 Architectural Support for Abstractions

As common programming paradigms become more refined, architectures will begin to incorporate them into their instruction sets. It would be unlikely that the only abstractions supported by architectures would remain simple data types (integer, real) and their associated operations (add, subtract, convert). Other abstractions such as processes, semaphores, ports, timers, memory management, and faults that are found in typical applications should be supported as well, along with associated operations on those abstractions.

Moving hardware to another level of abstraction provides huge advantages for testing tools. As stated earlier, the architecture must be the basis for emulation capabilities and providing execution visibility. As the hardware becomes more aware of programming elements, it gains the ability to send more meaningful messages to the external world. A context switch between processes could be sent with a single message, rather than the hundreds of machine instructions it takes to implement the switch.

As the processor becomes the single point of visibility, awareness of the programming environment becomes important. A processor with a high-level understanding of program entities can emit fewer, more meaningful messages than a processor that only comprehends low-level instructions.

6.5 Dedicated Bus

Embedded testing and debugging require an interface that allows the processor to communicate with the external world without interfering with the behavior of the system under test. This physical connection should reside on the target and interface externally through some detachable mechanism. The separation technique is important, since the external debugging system will be detached from this connection once the system is placed into operation. The execution behavior of the program should be independent of whether or not any external tool is attached.

Assuming an adequate physical connection, the next determination is the protocol across it. The following issues must be addressed:

1) At what rate will messages need to be sent?
   Processor speed raises interesting problems, since future speeds might be too quick for external processing techniques. A solution to this problem was discussed previously where the processor became aware of high-level program elements. The goal is to decrease the number of messages required relative to the number of machine cycles.
2) How much data is associated with a message?
   If an architecture is required to emit large volumes of data for messages, there may be instances where the processor must be suspended to allow the internal debugging hardware to catch up to the current processor state. Higher level messages may compound the problem, since more meaningful messages might require more

information. There is likely a tradeoff between message level and data volume.

3) Is the connection bi-directional?
Visibility concerns dictate that state information travel out of the processor. However, methods requiring control of the executing program require that state information travel the other direction. Protocols must be in place to handle contention across the bus and those must be extremely well defined, due to the extreme data rate that could will be encountered across the bus.

4) Who is the active element in sending messages?
Either the processor or the RTS must determine the information sent from the processor. The processor cannot provide all the state information needed, while the RTS will likely not be able to maintain adequate speeds for sending messages.

These questions play a role in determining the interface between the internal and external debuggers. A likely solution would be a master-slave relation, where either the internal or external debugger regulated the other. This scenario does not seem likely, since each has such critical processing concerns. Therefore, each will likely execute independently, while communication is handled via some bus and protocol.

There does exist a master-slave relationship in respect to the bus, however. During program execution, the internal debugger must "own" the bus, since it's processing concerns are the greatest. It must meet the message sending deadlines without altering computations in other parts of the system. There are points during execution where the external debugger must seize control. If the internal debugger cannot allocate the bus to meet the demands of the external debugger, the user must be notified that their requested operation cannot be accomplished during a real-time execution.

The final determination is that of the active element within the processor. There are two basic approaches to determining control of the internal debugging activities. In the first the processor is active and becomes responsible for sending messages to the external debugger. The second approach uses a special debugger portion of the RTS to emit messages, which is loaded into a dedicated functional unit within the architecture. Tools require information maintained by both the architecture and the RTS. Perhaps the solution lies between the two where both the RTS and architecture have the ability to dump messages, depending on the current requirements dictated by the external tool.

7. Run-Time System Additions

The RTS requirements describe an interface between a tool and the underlying system. This is a logical interface requiring substantial hardware support as outlined above. An obvious goal is to minimize the required data and computational requirements of the internal debugger as well as the required communications between the internal and external debuggers.

This paper does not address the question of how these interfaces should be utilized. Such answers should be given by methodologies and techniques for detecting and locating errors in embedded, real-time systems. As discussed earlier, the lack of these methods has led to difficulties for determining adequate RTS services for testing and debugging tools, which has forced a different approach to determine the required operations. Since the RTS is in essence offering an implementation of high-level abstractions, services that provide visibility into the implementation of RTS abstractions should adequately fulfill the needs of most testing and debugging techniques.

A standard currently exists for implementing these abstractions in the MRTSI [ARTE89] and CIFO [ARTE91]. In addition, most of the needs for testing and debugging can be fulfilled by these standards. This is not surprising, since our solution is based on implementation visibility, and the MRTSI and CIFO are providing an implementation interface. However, it is important to note that this approach also indicates that implementations that support these standards should require minimal additions to also support testing and debugging tools as proposed by this paper. Below is a small discussion surrounding each of these abstractions and a list of shortcomings in the MRTSI and CIFO for testing and debugging.

7.1 Processes

Concurrency is a common abstraction used in embedded systems. A design can be decomposed without concern for computational resources, which can then be determined by a scheduler during run-time. Ada implements concurrency through tasks and task types. The CIFO and MRTSI provide extensive tasking support including identification, creation and activation, communication through rendezvous, concurrent access to shared entities, and support for scheduling control. Elements of interest that are not provided by the CIFO or MRTSI include:

- Task State - A developer must have the ability to query and modify the task state for each task in their system. However, a modification could leave the RTS in an inconsistent state. For example, changing a task's state from "delaying" to "running" without removing it from the delay queue would place the RTS into a state that could not be achieved through normal execution. However, the same modification ability is available on typical debugging systems and should be offered by embedded debuggers as well.
- Communication and Synchronization - A developer must have the ability to view and modify each entry queue to determine the concurrent state of the system. Again, modifications could leave the RTS in an unobtainable state.
- Scheduling Control - In addition to the extensive operations provided by the CIFO for concurrency control, a developer must have access to the dispatch port (or ports for multiprocessor systems).

7.2 Interrupt Management

One of our criticism of the current approach to embedded testing is that timing errors are revealed late in the development process. Interrupts are very related to timing issues and their correctness is an important element in embedded testing. Therefore, support for interrupts is extremely important to target testing and debugging. Facilities provided through the CIFO and MRTSI would allow developers to bind various interrupt handling routines, enable and disable certain interrupts, mask and unmask interrupts, and generate software interrupts all controlled dynamically during program a program test.

7.3 Time Management

As stated earlier, timing issues are extremely important to target testing and debugging. Therefore, target tools require sufficient control over issues relating to time. Tools must be allowed to view and modify the clock (although such modifications might produce undefined results) and the delay list of waiting processes maintained by the RTS.

7.4 Memory Management

Dynamic memory is not typically used by embedded applications due to difficulties in demonstrating reliability. However, future systems will likely incorporate algorithms that require dynamic storage. In addition, memory management for dynamic allocations is part of a RTS and should therefore be included in RTS visibility and control discussions. A tool will likely require that ability to demonstrate an application programs behavior when memory is exhausted.

The MRTSI would need to be extended to provide operations that resize a collection making it smaller to show execution behavior when memory is exhausted or larger to demonstrate correct execution should a collection be expanded by the developer. Resizing is not cheap and could require a great deal of computation and data transfers, depending on an implementation.

7.5 Exception/Fault Handling

Proper handling of exceptional events is evaluated during hardware-software integration testing. Therefore, tools require a great deal of control over exceptions and recovery mechanisms. One must be able to raise an exception or fault during program execution and also modify handler binding during execution.

Another question of interest might be to locate the handler for a given fault or exception at a given program location. Such information is not easily gained from the underlying system. The compiler is responsible for handling exception propagation [ARTE89], so determining the handler from only RTS information might be an impossibility and is at best resolved uniquely for each compilation system.

8 Conclusions

The goal of this paper is two fold. The first goal is to identify deficiencies in embedded system testing and raise questions about the future of current tools. The second is to propose a solution to these problems through architectural and RTS additions. The architectural additions will certainly be costly in both time and space, requiring space (transistors) on the chip and access to internal registers and busses that could cause contention and slow the execution of other instructions provided by the architecture. However, the RTS additions are minimal. We defined the needs of testing as making the implementation details of common system abstractions visible and then determined the functionality required to view and control them. The ARTEWG's MRTSI and CIFO provided an outstanding basis for this approach.

The RTS additions are admittedly weak. Our initial goal was to have the methodologies and techniques used for testing embedded, real-time systems drive the operations required by the RTS. Unfortunately, such methods do not yet exist. As stated earlier, testing and debugging of embedded, real-time software remains a black art, with ad hoc methods and techniques. While there has been much research into the concurrency and distribution issues, none has examined real-time constraints, embedded environments, and other issues relating to embedded systems Perhaps the MRTSI and CIFO are sufficient for implementing target level testing and debugging tools. However, this question cannot fully be resolved until more formal methods exist.

Our next step is to evaluate the additions and determine their feasibility. Questions relating the cost of these additions to an architecture and RTS in terms of time and space must be answered. Also, a more complete mapping should exist between the added features and the impact they have on the desired features. One can then make a valid comparison between a feature and the costs associated with it.

The embedded controller market is currently huge, but has only begun to require the computational powers associated with microprocessors. Embedded applications have traditional been event driven rather than computation dependent. Due to their light weight, easy configurability and expansibility, and lower design complexity, computers are quickly being chosen over mechanical techniques for controlling devices. As this transition continues, the size and complexity of embedded programs will grow. Controllers will not only have strict timing requirements, but also have significant computational needs as well. This combination requires new approaches to our current testing process for embedded systems and therefore, more effective tools to aid in testing and debugging embedded applications.

References

[ARTE89] Ada Run-time Environment Working Group, "A Model Run-Time System Interface for Ada," Ada Letters, January, 1989.

[ARTE91] Ada Run-time Environment Working Group, "Catalogue of Interface Features

and Options for the Ada Runtime Environment," Special Edition of Ada Letters, Fall 1991 (II).

[CHIL91] Child, Jeffrey, "32-bit Emulators Struggle with Processor Complexities," Computer Design, May 1, 1991.

[DOD83] Department of Defense, Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815a, United States DoD, 1983.

[FAA85] Federal Aviation Association, Software Consideration in Airborne Systems and Equipment Certification, RTCA/DO-178A, 1985.

[GILL88] Gilles, Jeff and Ford, Ray, "A Guided Tour Through a Window Oriented Debugging Environment for Embedded Real Time Ada Systems," IEEE Transactions on Software Engineering, 1988.

[HANS78] Hansen, B., "Reproducible Testing of Monitors," Software-Practice and Experience, Volume 8, 1978.

[HEMB85] Hembold, D. and Luckham, D., "Debugging Ada Tasking Programs," IEEE Software, March, 1985.

[INTE92] Intel Corporation, i960 Extended Architecture Programmer's Reference Manual, 1993.

[KOEH91] Koehnemann, H.E. and Lindquist, T.E., "Runtime Control of Ada Rendezvous for Testing and Debugging," Proceedings of the 24th Hawaii International Conference on System Sciences, Volume II, 1991.

[LEDO85] LeDoux, C. and Parker, D.S., "Saving Traces for Ada Debugging," Ada in Use: Proceedings of the Paris Conference, 1985.

[LYTT90] Lyttle, D. and Ford, R., "A Symbolic Debugger for Real-Time Embedded Ada Software," Software - Practice and Experience, May 1990.

[MAUG85] Mauger, C. and Pammett K., "An Event-Driven Debugger for Ada," Ada in Use: Proceedings of the Paris Conference, 1985.

[TAI86] Tai, K.C., "Reproducing Testing of Ada Tasking Programs," IEEE Transactions on Software Engineering, 1986.

[TAI91] Tai, K.C., Carver, R.H., and Obaid, E.E., "Debugging Concurrent Ada Programs by Deterministic Execution," IEEE Transactions on Software Engineering, January, 1991.

[TAYL80] Taylor, R.N. and Osterweil, L.J., "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," IEEE Transactions on Software Engineering, May, 1980.

[TAYL83] Taylor, R.N., "A General Purpose Algorithm for Analyzing Concurrent Programs," Communications of the ACM, May, 1983.