DIRECT EQUIVALENCE TESTING OF EMBEDDED SOFTWARE

by

Rohit Pagariya

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computing

School of Computing

The University of Utah

August 2011

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Rohit Pagariya**

has been approved by the following supervisory committee members:

John Regehr	, Chair	**10/18/2010** Date Approved
Ganesh Gopalakrishnan	, Member	**10/18/2010** Date Approved
Matthew Might	, Member	**10/18/2010** Date Approved

and by **Al Davis** , Chair of the Department of **Computer Science**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

Direct equivalence testing is a framework for detecting errors in C compilers and application programs that exploits the fact that program semantics should be preserved during the compilation process. Binaries generated from the same piece of code should remain equivalent irrespective of the compiler, or compiler optimizations, used. Compiler errors as well as program errors such as out of bounds memory access, stack overflow, and use of uninitialized local variables cause nonequivalence in the generated binaries. Direct equivalence testing has detected previously unknown errors in real world embedded software like TinyOS and in different compilers like msp430-gcc and llvm-msp430.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am thankful to my advisor Prof. John Regehr for providing timely guidance and support which helped me with my research and this thesis. He has been a great mentor and motivator during the two years that I have spent here at Utah. I would like to thank my committee members, Prof. Ganesh Gopalakrishnan and Prof. Matthew Might for their comments and feedback on this thesis. I would also like to thank my colleagues for the lively discussions during the past two years and the feedback on my work.

I would like to Yang Chen and Xuejan Yang for their help with CIL and randprog hacking. Thanks are also due to Parasaran, Tarun, Suman, Sriram, Rajvarma, Niladrish, Clifton, Subodh and Anh for the various discussions on theoretical computer science, economy, philosophy, and different cuisines between games of Foosball. A big 'Thank you' to Karen Feinauer for answering hundreds of (silly?) questions.

Lastly I would like to thanks my parents and family without whose support this would not be possible.

CHAPTER 1

INTRODUCTION

Embedded systems are becoming an integral part of our lives as we move ahead in the 21st century. Industries like automotive, aviation, defence systems, consumer electronics, medical health equipment, and home automation depend on embedded software more than ever before. Embedded software is used in various safety critical systems like missiles and airplanes for navigation, in automobiles for throttle and brake control, and inside pacemakers. It is also present in a number of gadgets like cellphones, videogames, televisions, and other home appliances. Bugs in embedded software could potentially be fatal. It is imperative to ensure the quality and robustness of embedded software, most of which is written in type unsafe languages like C and C++, using verification and validation.

1.1 Errors in software

An anomaly that causes a computer program to produce wrong or unintended results is a software error. Errors can find their way into software in different ways. Software errors have a negative impact on quality of a product and increase development costs and time to market. This reduces profits and has an adverse impact on productivity. The cost of an error depends on how early it is detected in the development cycle. It is desirable to detect the errors as early as possible. A National Institute of Standards and Technology study estimates that defective software costs the U.S economy alone, around $60 billion per year [21].

1.2 Verification and validation

Testing is an effective way to detect and locate errors in a system. Testing helps validate whether a system performs as per its specifications. Software testing is

the process of executing a program with the intent of finding errors [19]. Software testing is very broad field and different techniques fall into various categories of classification. If access to the source code is used as a category for classification, there is white box and black box testing. White box testing is used to test the internal structures of a program and determine the path and branch coverage of the code. Black box testing is functional testing that tests the system against the specifications and requirements.

If granularity at which software is tested is used as a category, testing can be classified as unit testing, integration testing, and system testing. Unit testing is where each module of the system is tested individually. Integration testing is performed when two or more modules are integrated. System testing is an end-to-end testing method to check if a system conforms to its specifications.

A combination of the above mentioned techniques is available to test embedded software. Despite all the testing efforts, software may still contain errors. Edsger Dijkstra [6] has famously mentioned "Program testing can be used to show the presence of bugs, but never to show their absence!"

Symbolic execution reasons about all possible paths taken by an input in a program. Model checking techniques aim to explore the program state space and find an input which leads to an erroneous program state. Symbolic execution and model checking techniques, effective as they are, cannot be applied to all software as they lead to state explosion. There have been efforts to apply the techniques of symbolic execution [2,5] and model checking [12] for testing of embedded software. These approaches, though powerful and promising, are limited in scalability and applicability due to the inherently nonportable nature of embedded software and interrupt driven concurrency.

1.3 Equivalence testing

Equivalence of two objects means they are essentially same with respect to some characterics. Given two different computer programs, they may produce equal outputs for all valid inputs. This is known as *functional equivalence*. Conversely,

knowing that two programs are equivalent for some characteristic, we can test the programs for violation of equivalence for a range of valid inputs. This is called *Equivalence Testing*.

The technique of direct equivalence testing presented in this thesis is a combination of differential testing and equivalence checking. A valid input, a test program in this case, is presented to several comparable systems, different compilers, and the output, different binaries, is checked for violation of equivalence. Binaries generated from the same piece of code should be equivalent, irrespective of the compiler, or the compiler optimizations, used. The failure of equivalence check indicates an error, either in the test program or in the compiler. The concept of a state of a machine, represented by a snapshot of the memory at specific points during the program execution, is used to determine equivalence.

Compiler optimizations, interrupt driven concurrency, and semantics of the volatile type qualifier pose some of the key challenges for equivalence testing of embedded software. The solutions to these problems are the research contributions of this thesis.

I applied the technique of direct equivalence testing to detect errors in TinyOS, an operating system for embedded sensor nodes, applications as well as cross compilers like msp430-gcc and llvm-msp430, that are used to compile these applications. Direct equivalence testing has been successful in detecting errors in TinyOS applications as well as compilers. It observes the whole system while still obtaining low level information that enables it to detect memory safety errors, compiler errors, and portability errors. It does not depend upon the timing properties of the system and, as such, can not detect synchronization errors like deadlock, livelock, and race conditions. Since it depends on differential testing, an error that makes all the programs under test to behave in a similar way can not be detected. Direct equivalence testing can detect any errors that lead to different values being stored in the memory, across the different binaries. It is a way to determine if the program semantics are changed by different compilers or compiler optimizations.

This thesis claims that the technique of direct equivalence testing, as explained in the subsequent chapters, is effective in testing of embedded software to detect errors in cross compilers as well as application programs.

CHAPTER 2

BACKGROUND

This chapter provides information about embedded software, interrupts, device registers in microcontrollers, sensor networks, and TinyOS. It also talks about the semantics of the volatile type qualifier in C, memory allocation in C, and its significance for direct equivalence testing.

2.1 Embedded systems and software

An embedded system is a special purpose computer that performs one or few dedicated functions. Users do not perceive an embedded system as a computer in the traditional sense. More often than not, embedded systems perform or control an action. The software that runs on these systems is optimized in terms of performance, memory, and power to perform the few tasks well. Examples include music players, cellphones, microwaves, traffic lights, cameras, televisions, MRI scanners, cars, airplanes, missiles, etc. Today, embedded systems are pervasive.

Concurrency, robustness, responsiveness, and heterogeneity are essential characteristics of embedded software. Concurrency in embedded software manifests as interrupt driven concurrency, multiple processes, and threads. A simple example of an embedded system is an air conditioner that maintains the room temperature at a preset level. The basic components of an air conditioner are a temperature sensor to detect the current temperature, a microcontroller to process the input from the sensor, a heater to increase the room temperature, and a fan to cool the room. Embedded software must perform the tasks of reading the room temperature, processing the input, and controlling the actuators, concurrently, using interrupts, processes, and threads. It should be reliable and robust to perform under normal and extreme conditions. In the event of a failure, it should produce notifications

instead of failing silently. Embedded software should be able to respond in real time to the changing conditions in the room. Depending upon the ability of an embedded system to respond to an event, they are classified as hard real-time or soft real-time systems. A hard real-time system is a system that meets all its deadlines. Failure to meet a deadline in a real-time system could be catastrophic. An airbag deployment after the stipulated time can result in a fatal injury to the passenger. A soft real-time system can afford to miss a deadline. There is no fatal damage if the DVD player skips a couple of frames once in a while, though it might not be good for business. Embedded systems are heterogeneous. It is common for embedded software to deal with analog-to-digital converters (ADC), digital-to-analog converters (DAC), system timers, watchdog timers, Serial Peripheral Interface (SPI) bus controllers, Inter-Integrated Circuit (I2C) bus controllers, Controller Area Network (CAN) bus controllers, etc.

The target embedded systems for direct equivalence testing are wireless sensor nodes with low memory (few KB of RAM and few tens of KB of ROM) and low power operation. These systems have very limited debugging or profiling support, and as such, are not easy to test. These systems do not have dynamically allocated memory.

2.2 Interrupts

Imagine you are waiting for a friend at your home and want to know of his arrival. A simple way would be for you to go to the door every couple of minutes and check if he has arrived. Obviously, this wastes a lot of your precious time. A smarter solution would be for you to wait till you hear the door bell, letting you know that your friend has arrived. An interrupt in a microprocessor is like the door bell, an asynchronous signal which informs the processor about an event needing its attention. Meanwhile, the processor can continue with its work in absence of an interrupt.

When a microprocessor receives an interrupt request, after completing the execution of the current instruction, it saves the program counter (PC) and other

registers on the stack, loads the PC with address of the interrupt handler, and starts executing instructions from the interrupt handler. At the end of the interrupt handler, the saved PC and other registers are restored from the stack and the execution of the previously interrupted program resumes. Following are the most common uses of interrupts: timers, network and disk IO, power-off signals, and process context switching in an operating system. Any computer system has multiple interrupts enabled simultaneously that may result in multiple interrupt requests. The processor handles the interrupts based on a fixed priority or as set by the user. Based on the processor architecture, an interrupt handler could be interrupted by a higher priority interrupt handler. This leads to interrupt driven concurrency.

Interrupt driven concurrency is an integral part of all software. The operating system abstracts over interrupt driven concurrency and presents an easy to use abstraction of processes and threads. Small embedded systems, that cannot afford the overhead of an operating system in terms of memory or CPU cycles, have to deal with interrupt driven concurrency.

2.3 Device registers

There are various devices in a microcontroller like analog-to-digital converters (ADC), digital-to-analog converters (DAC), system timers, watchdog timers, Serial Peripheral Interface (SPI) bus controllers, Inter-Integrated Circuit (I2C) bus controllers, Controller Area Network (CAN) bus controllers, radio etc. The programming interface to these devices, as seen by the CPU, is the device registers. The device registers are used to control a device as well as transfer of data across a device. For example, to read a value converted by the ADC, the ADC control register is loaded with calibration bits to start the conversion. The end of conversion is signaled by an interrupt or a bit in the control register. The converted value is then read from the ADC data register.

Device registers can be accessed directly using special IO instructions or can be accessed like memory. Each memory mapped device register is allocated a memory

address and data can be read from/written to these registers by accessing the memory address.

2.4 TinyOS

A wireless sensor node is a microcontroller based device with sensing and networking capabilities. It is a memory constrained device with only few KB of RAM and tens of KB of ROM for code. A sensor node is highly power constrained device in that it may be expected to work for months on a AA battery. Multiple sensor nodes can be used to create a sensor network and monitor physical parameters like temperature, pressure, etc. of an area for weeks or months. TinyOS [24] is an operating system designed for low power wireless sensor networks providing excellent capabilities for networking. It has support for protocols like time synchronization, data collection, and data dissemination.

TinyOS is written in nesC [11], a dialect of C. It features a component-based architecture where a component is data and code coupled together. Components are wired together to form a TinyOS program. It transforms the TinyOS program so that the compiler finds it easy to optimize. TinyOS has a vast component library which enables rapid implementation while minimizing code size as required by the severe memory constraints inherent in sensor networks.

TinyOS has a split-phase event-driven execution model to achieve concurrency. The programming APIs are nonblocking and have completion call backs. To start an operation, a call is made, which returns immediately. End of the operation is signaled by a callback event. For example, to read an ADC value, a call to `ADCRead` is made, which returns immediately. When the value is read, `ADCReadDone` event is fired.

TinyOS is an nonpreemptive, task oriented operating system. A queue for ready tasks is maintained and scheduler executes one task at a time to completion. An application posts a task to perform some work. A TinyOS task runs to completion and can only be preempted by an interrupt handler. A TinyOS task is a function

with no input parameters. This implies the contents of the stack frame are invalid before the start and after the end of execution of the task.

2.5 Volatile qualifier in C

The *volatile* keyword in C is a type qualifier used to declare that an object can be modified by the hardware or a concurrently executing interrupt/thread. The *volatile* qualifier was primarily introduced for accessing the contents of device registers in a C program.

In Section 6.7.3 the C99 standard [14] says:

> An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine . . . .

The semantics of volatile qualified memory locations are such that values stored in them can change without the knowledge of compiler. Thus the compiler should not cache the value of a volatile-qualified object in a register.

A footnote in the same section elaborates:

> A volatile declaration may be used to describe an object corresponding to a memory-mapped input/output port or an object accessed by an asynchronously interrupting function. Actions on objects so declared shall not be "optimized out" by an implementation or reordered except as permitted by the rules for evaluating expressions.

A compiler is not permitted to optimize accesses to volatile qualified locations.

2.6 Memory allocation in C

There are three different ways to allocate memory in C.

Static allocation Memory for objects is allocated at compile time by the compiler. The life time of these objects is the execution time of the binary.

Automatic allocation is implemented in terms of a stack. Objects are allocated memory on the stack at start of execution of a code block. When the control flow exits the code block, the allocated space is automatically reclaimed.

Dynamic allocation Memory for objects is allocated at run-time on the heap using `malloc`. The heap allocated memory is valid until returned using `free`.

Static and automatic allocation and deallocation of memory is taken care of by the compiler, freeing the developer from the task of memory management. On the other hand, dynamic memory allocation and deallocation on the heap has to be managed by the developer.

Automatically and dynamically allocated memory is initialized if the initialization value is specified. Otherwise, it contains garbage values. Reading an uninitialized memory location is an undefined behavior in C. For example, reading a uninitialized variable stored on the stack will return an indeterminate value. Only when the program initializes a variable stored on the stack or writes to it before reading it again, it contains a valid value. This value becomes invalid once the program flow has exited the code block.

CHAPTER 3

DIRECT EQUIVALENCE TESTING

High level languages provide an abstraction over the machine language of a computer. Software developers write programs that conform to this high level abstraction. They assume that the program semantics are preserved by the tools that convert their programs to machine instructions, namely the compiler, assembler, linker, and loader. These tools are not completely error free. Yet it is of vital importance that they be error free, as it potentially affects all the software that uses these tools. Direct equivalence testing addresses these problems.

Equivalence testing observes different versions of the same program under execution and checks for the failure of equivalence. Checking for equivalence can be as easy as comparing the final output value. But embedded software runs continuously for months and there is no final output to be compared. The solution lies in being able to find points of equivalence in the program where the state of the different versions of the program can be compared during execution. The semantics of the C programming language make it easy to find such program points, usually the start and end of a function. If a point A in the program is such a point of equivalence, then the state of the system, when the execution of different binaries reaches point A, should be equivalent. Block of code between two such points then becomes a unit of testing. It is important to find as many units of testing in the code base as possible for effective testing.

Direct equivalence testing does not test for timing equivalence. It needs the code, i.e., the unit of testing to be deterministic. Interrupt driven concurrency makes a system indeterministic. Compiler optimizations and volatile locations introduce

nonequivalence in the system. The rest of the chapter explains the technique of direct equivalence testing and how it handles the above mentioned problems.

Before we proceed, let us define a state and a computation:

State A set of parameters that provides information about the system is called a state. A snapshot of values stored in memory can be considered a state of a computer system.

Computation It is a block of code or a sequence of instructions that operates on a state.

Direct equivalence testing relies on the invariant that *two equivalent computations should produce the same output states for the same input states, for all valid input states.*

For testing the violation of the above invariant, we need to decide what does a state and a computation mean in real terms? A computer's main memory contains most of the information that is needed by a program. A computation can be expressed in high level languages like C, C++ or assembly language or machine language. Irrespective of this, all forms of expression view the state, i.e., main memory, similarly. For example, let variable *foo* in C correspond to memory location `0x12AB`. The value in variable *foo* as interpreted by the C abstract machine is same as that in memory location `0x12AB` as interpreted by assembly and machine language. Let us now define the notion of direct equivalence testing more concretely.

3.1 Machine state

A computer, in its simplistic form, can be thought of as a machine with only memory and an execution unit. Main memory (RAM) is a part of the computer memory while units like instruction decoder, ALU, FPU, shifters are all part of the execution unit. Computer memory is a set of locations that store data. Logically, registers and cache are part of the computer memory, with very fast access times. At the start of the computation under consideration, main memory stores the data related to computation while contents of the registers and cache are assumed invalid.

We can say that the main memory, i.e., the memory locations and the data present in them, completely represents the state of a machine.

Let A be the set of memory locations while B is the set of values that can be stored in a memory location. For example, $A = \{\texttt{0x}00, \texttt{0x}01, \texttt{0x}02, ..., \texttt{0x}FF\}$ is the set of memory locations while $B = \{0, 1, 2, ..., 9\}$ is set of values that can be stored in a memory location. We say that a machine state is a function that maps memory locations to values stored in them.

$$M : A \mapsto B$$

In practice, we say that a snapshot of values stored in RAM is a machine state.

3.1.1 Equivalence of machine states

Two machine states are equivalent if, for all the memory locations in a machine state, the corresponding memory location in other machine state contains same value.

3.2 Computation

A computation is a block of code that performs an operation on a machine state. This may or may not produce a new machine state. A computation, thus, can be thought of as a function that maps a machine state to a machine state or special nontermination state.

$$C : S \mapsto S \cup \{\emptyset\}$$

where S is the set of all machine states. From here on, the machine states at the start and end of execution of the computation are referred to as input machine state and output machine state, respectively.

3.2.1 Equivalence of computations

Two computations are equivalent if, for all valid input machine states, they map to the same output machine state.

The above definition of equivalence requires to map the whole of RAM as a machine state. Embedded systems targeted for direct equivalent testing do not

contain dynamically allocated memory such as a heap. Though automatically allocated memory or a stack is present.

The C standard does not mention the use of a stack frame. This allows the compilers to optimize the contents of a stack frame whenever it can. There is no uniform treatment of the stack frame across different compilers, which becomes a major challenge in checking for equivalence.

The key insight of this thesis is to choose specific points in the program where the contents of the stack frame are invalid. Thus, it becomes necessary to neglect the values stored on the stack as part of the machine state.

Contents of a stack frame are invalid before the control flow jumps to a function and after the function is exited. A value of an object stored on the stack is valid only when the function in which it is declared is under execution. Indeed, reading an uninitialized variable is an undefined behavior in C. This undefined behavior can be easily checked by ensuring that the program never reads a value from stack before writing it. If we choose a piece of code between two points where a stack frame does not exist, as a unit of testing, then a machine state comprises only the statically allocated memory, i.e., global variables.

The start and end of the execution of a TinyOS task is an example. Since the TinyOS runs tasks to completion, stack frame is not valid before the start and after the end of the task. Thus, a TinyOS task is an ideal candidate for selection as a unit of testing. Given an input state, when different compiled versions of the same TinyOS task or multiple execution runs of a single task result in nonequivalent output machine states, we have found an error.

This notion of equivalence of computations can now be used for real world software programs to test if the semantics of a computation are preserved by the compiler during the compilation of the computation, expressed in C, to an assembly language representation.

3.3 Direct equivalence testing

A simple and naive approach to test would be as follows:

- Compile a piece of code using different compilers and optimization flags
- Compare the different compiled versions of the computation for equivalence.

The approach as shown in Figure 3.1, though technically correct, is infeasible in practice due to storage and compute overheads. Any computation reads and writes only a small fraction of RAM. Equivalence testing requires storing a large number of states. Storing all these states that include full contents of machine state increases the storage overhead. Comparing full contents of a machine state makes it difficult to match it with other machine states. A smarter and better approach is needed, known as a memory access trace.

3.4 Memory access traces

Memory access traces are a set of read and write references generated by a computation. The first read reference to a memory location, not previously written

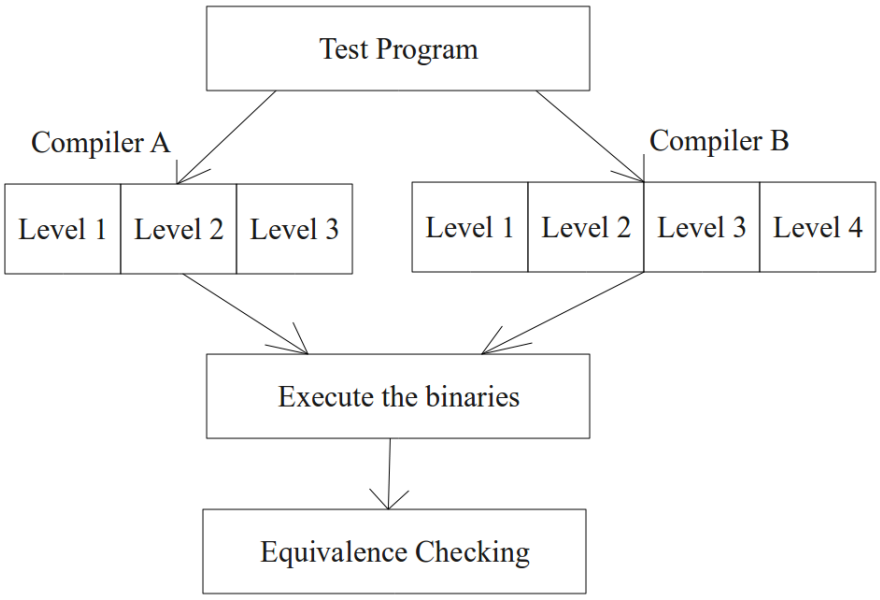

Figure 3.1. Simple testing procedure for equivalence

by the computation, is considered as an input to the computation. A computation does not necessarily read all the memory locations. If a read reference to a location is missing, it means the output of the computation does not depend on the value at that location and it is a do not care input to the computation. The last write reference to a memory location is considered an output of the computation. A missing write reference to a memory location can be inferred as the computation reading the value at the said location and writing it back without changing it.

Figure 3.2 shows a sample memory access trace as generated by a computation along with the input and output machine states obtained using the definition explained above. The memory traces generated by a computation are sufficient to infer about the input and output machine state of a computation.

Different compilers, or the same compiler at a different levels of optimization, place the same variable in the code at different addresses in memory. Compiler A can place a global variable `foo` at address 0x10 while compiler B can place it at 0x20. Obviously, comparing memory addresses to identify the variable `foo` does not work.

A simple solution is to store symbolic information about the memory location like the variable name instead of the memory address. Each memory location,

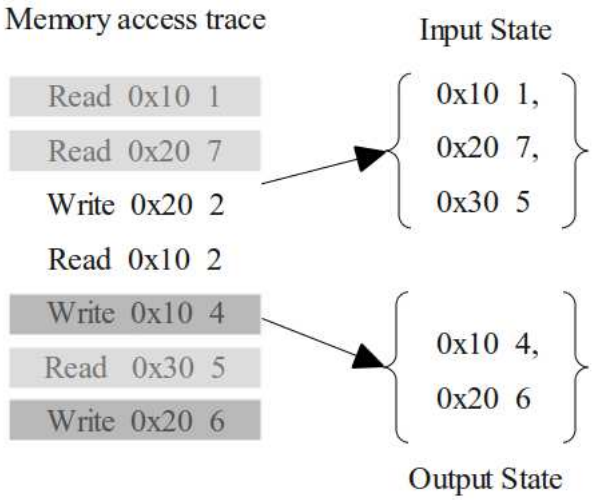

Figure 3.2. Sample memory access trace

except pointers, is assigned a unique name, which is a combination of a variable name and an offset. A value of 25 at address 0x10 can be represented as value 23 at variable `a_0`. The mapping of variable names to their memory addresses and their size in number of bytes is easily obtained from the symbol table stored in the binary. The value stored in a pointer variable is the name of the object it points to, instead of a scalar. For example, if pointer variable `ptr` points to variable `bar` which is located at address 0x100, the value of `ptr` is the variable name string `bar` instead of 0x100. Multilevel pointers are handled by recursively iterating over the symbol table till the pointed-to object is identified.

Consider the computation as shown in Figure 3.3.

Figure 3.4 shows the contents of the memory before and after the execution of each statement in the computation. The snapshot of memory before the start and after the end of execution of the computation represent the input and output machine state for the given computation.

The set represented as $\{(a,4),(a,5),(b,1),(b,6)\}$ is the set of read references and the order they are generated in. Looking at the contents of the memory, we know the actual input machine state is in fact the set $[(a,4),(b,1),(c,3)]$. As per our definition of input machine state which uses extrapolation of memory traces, the input machine state is the set $[(a,4),(b,1),(c,X)]$. Value stored in location c is a do not care input to the computation as it is not read.

Similarly, the set of write traces generated by the computation is $\{(a,5),\ (b,6),\ (a,6)\}$. The actual output machine state as observed from the RAM is the set $[(a,6),\ (b,6),\ (c,3)]$. Our definition of output state using extrapolation of memory traces tells us that in the output machine state, value 6 is stored at locations a and b. But since there is no write memory reference to location c, we can infer that the value stored in it is unchanged though we cannot tell what it is from the memory

```
a = a + 1;
b = a + b;
a = b;
```

Figure 3.3. A simple computation that adds and copies values

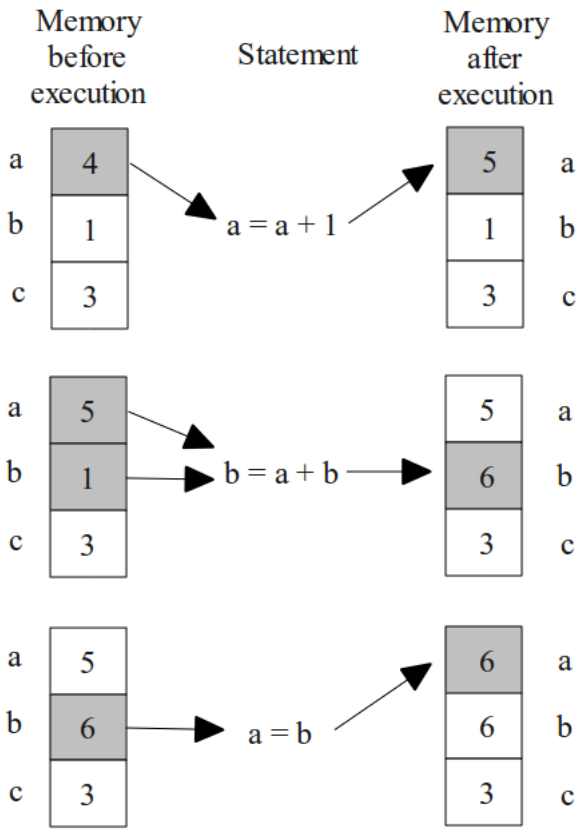

Figure 3.4. An example relating the computation, memory access trace and the machine states

access trace. Assuming that values that can be stored in RAM range from 0 to 9, we get 10 mappings of input to output machine state:

$[(a, 4), (b, 1), (c, 0)] \mapsto [(a, 4), (b, 1), (c, 0)]$

$[(a, 4), (b, 1), (c, 1)] \mapsto [(a, 4), (b, 1), (c, 1)]$

...

$[(a, 4), (b, 1), (c, 9)] \mapsto [(a, 4), (b, 1), (c, 9)]$

The above set of mappings obtained from the memory access traces gives us a corresponding output machine state when presented with an appropriate input machine state. Thus, when an actual machine state $[(a, 4), (b, 1), (c, 3)]$ is given as an input, we can correctly infer the output state $[(a, 4), (b, 1), (c, 3)]$ from memory access traces.

3.5 Compiler optimizations

A straight method to check for nonequivalence of computation is to find matching read reference sets but different write reference sets. This approach does not work in the presence of commonplace compiler optimizations that eliminate read and/or write reference to a location while preserving semantics of the program.

Consider a simple example as shown in Figure 3.5.

In function `foo`, global variable `x` is first written to and then read from resulting in a write reference and a read reference in that order. The compiler can return the cached value of `x` and eliminate the read reference as variable `x` is not `volatile` type qualified. An optimizing compiler can eliminate both write as well as read references if it can prove that `x` is not being written anywhere else in the program.

Figure 3.6 shows valid assembly code for `foo()` along with the read/write reference sets.

Any technique checking for equivalence, when given these read/write reference sets, must detect them as equivalent. The crux of the problem is to distinguish between valid compiler optimizations and incorrect compiler behavior. My solution is explained in the sections that follow.

3.6 Equivalence checking

Given two instances of memory access trace, how to check if they are equivalent?

- Compare the input machine states for equivalence.
- If equivalent, the output machine states are compared for equivalence. If found nonequivalent, an error is reported indicating equivalence check failure.

```
/* Global variable */
int x = 1;
int foo(void)
{
  x = 1;    /* Write */
  return x; /* Read  */
}
```

Figure 3.5. A simple example to demonstrate various compiler optimizations

```
foo:
  mov 1, &x
  mov &x, r15
  ret
R = {(x,1)}   W = {(x,1)}
```

(a) Both read and write present

```
foo:
  mov 1, &x
  mov 1, r15
  ret
R = {}        W = {(x,1)}
```

(b) Read absent

```
foo:
  mov &x, r15
  ret
R = {(x,1)}   W = {}
```

(c) Write absent

```
foo:
  mov 1, r15
  ret
R = {}        W = {}
```

(d) Both read and write absent

Figure 3.6. Possible assembly code for foo() in Figure 3.5

We know that memory access traces represent the same computation as expressed in C. When two input machine states are found equivalent, it is implied that output machine states should be equivalent. If they fail the equivalence check, we have detected an error.

3.6.1 Equivalence of input machine states

In practice, equivalence of two input machine states can be checked by a simple rule: *Two input machine states are equivalent unless there exists a memory location, common to both the states, from which different values are read.* The above rule

handles all possible cases that arise due to compiler optimizations, as discussed in Section 3.5. Furthermore, it is trivial to check in practice.

Note our assumption that absence of read reference from one of the memory access traces does not result in nonequivalence immediately. Absence of a read reference means one of the following two scenarios. The value read from that particular location is not an actual input to the computation, i.e., it is a do not care input. Or the value read is a valid input but wrong code is generated and hence an error must be reported. If wrong code is generated then the corresponding output machine state will be nonequivalent and an error is reported eventually. If no error is reported, it means the particular input was in fact a do not care input.

3.6.2 Equivalence of output machine states

To determine the equivalency of output machine states, we define a rule which says when the two states are *not equivalent*.

Two output machine states are *not equivalent* when any one of the below conditions is satisfied:

- If different values are written to same memory location.
- If there is a missing write to a memory location in one of the memory access traces, except when the missing value to be written is the same as the value that was read from the memory location.

When different values are written to the same memory location, it means generation of wrong code by the compiler or a memory safety error. Absence of a write reference means there is a compilation error except when the value to be written is the same as the value read from that location.

The above two rules for checking equivalence of machine states are sufficient for determining if two memory access traces are equivalent. Also no false positives are reported by these rules. It is important that number of false positives reported tend to zero. False bugs reduce the efficacy of the technique and lead others to think if the particular technique really works.

3.7 Volatile locations

The *volatile* keyword in C is a type qualifier used to declare that an object can be modified in the program by the hardware or a concurrently executing interrupt/thread. The *volatile* type qualifier was primarily introduced for accessing the contents of device registers in a C program. For the purposes of testing, we consider all memory mapped hardware registers as volatile qualified memory locations. The semantics of volatile qualified memory locations are such that values stored in them can change without the knowledge of compiler. Thus the compiler should not cache the value of a volatile-qualified object in a register. A compiler is not permitted to optimize accesses to volatile qualified locations.

The semantics of volatile type qualifier present a unique challenge for direct equivalence testing. While checking memory references to volatile qualified memory locations for equivalence, the following invariants must hold:

- Order of memory references (read or write) to memory locations must be the same across two equivalent memory traces. This means that not only should the order of references to variable `A` be same but the order of references to variables `A` and `B` should also be the same across equivalent memory traces. A compiler is allowed to interleave references to volatile locations only between sequence points. But the code that exercises such behavior is rare and probably unintended.

- No memory reference (read or write) should be added or deleted.

As defined in Section 3.4, only the first read reference is part of input machine state while last write reference is part of output machine state. The above mentioned invariants related to the volatile type qualifier mean that all memory references, not just first read and last write, should be checked. Direct equivalence testing handles the above invariants as follows:

- All the read references to the volatile qualified memory locations are considered as part of the input machine state but as separate inputs. Multiple read references are enumerated to differentiate them from one another.

- All the read and write references are considered as part of the output machine state. Whenever a memory location is accessed, that reference is appended to a list of references to the particular memory location. While checking for equivalence, the list of references is compared for equality, instead of just the last value. For example, when a program reads the volatile variable `volatile_data` twice, then value of `volatile_data` in the output machine state is {(Read, X), (Read, Y)} where X and Y are the values read. This ensures that we check for all the references as well as their order.

A missing write or read reference is detected when the list of references is compared, leading to failure of equivalence check.

3.8 Interrupt driven concurrency

Interrupt driven concurrency is a characteristic feature of embedded software. When an interrupt fires, control jumps from the executing program to an interrupt handler. The interrupt handler performs some work and control goes back to the interrupted program.

There is a possibility that interrupted program and the interrupt handler share some data and may also modify the shared data. For example, let us assume that the variable `shared_data` is written an interrupt handler as well as an interrupted program. Consider the scenario when the interrupt fires right after the program writes 10 to `shared_data`. Now the interrupt handler updates the `shared_data` to 20. If `shared_data` = 10 is the last write to `shared_data` in the program, the output machine state should reflect the value of `shared_data` as 10. But the interrupt handler has updated `shared_data` to 20. This is the actual value reflected as part of the output machine state and incorrectly leads to non equivalence of output machine states. The modification of value in `shared_data` by the interrupt handler happens outside the knowledge of the program. This leads to false positives being reported due to the aliasing by the interrupt handler.

We solve the problem of interrupt driven concurrency by detecting this aliasing by the interrupt handlers and discarding the tainted memory access traces. We

consider the interrupt handler as a separate computation, different from the interrupted computation. This helps in testing the code in the interrupt handlers as well. If the interrupting computation writes to a memory location that was either read or written by the interrupted computation, the memory access traces from that execution run of the interrupted computation are discarded. Discarding a few memory access traces does not affect our testing because if there are errors in the discarded traces, these errors will be detected in other memory access traces, not tainted by interrupt driven concurrency. An error, if it exists, is very likely to manifest itself while checking different memory traces of the same computation. Thus discarding a few traces out of thousands doesn't affect the efficacy of the technique.

Table 3.1 shows a summary of the different problems faced during the testing and the solutions to overcome them.

Table 3.1. Summary of the problems faced and their solutions

Problem	**Solution**
Non equivalence due to presence of a stack frame	Choosing points during program execution when contents of stack are invalid
Interrupt driven concurrency	Discarding the tainted memory trace
Compiler optimizations	Rules for equivalence of input and output machine states
Device registers and volatile locations	Checking all memory references for exact match.

CHAPTER 4

EXPERIMENTS AND RESULTS

I have applied the technique of direct equivalence testing to detect errors in TinyOS applications as well as various compilers. Below is a quick overview of the tools and software I used for testing purposes.

4.1 Tools

4.1.1 Simulators

MSPSim It is a Java-based instruction level emulator of the MSP430 microcontroller series by J. Eriksson et al. [9]. It supports emulation of sensor networking platforms such as Contiki [7], and also TinyOS [24] to some extent. It supports emulation of various peripherals on the sensor boards and enables profiling, debugging and instrumentation. MSPSim was modified to recognize the start and end of a TinyOS task or an interrupt handler and to monitor a specified range of memory addresses. Memory references generated before the start of the `main`() were discarded during random testing of compilers.

COOJA It is a sensor network simulator primarily designed to enable cross-level simulation: simultaneous simulation at many levels of the system. It is a part of the Contiki sensor network OS [7] and depends on MSPSim for low-level simulation of sensor node hardware. COOJA is flexible and extensible in that all levels of the system can be changed or replaced: sensor node platforms, operating system software, radio transceivers, and radio transmission models [22]. COOJA was modified to use the hacked version of MSPSim and to log memory access traces for a single sensor node during the network simulation.

4.1.2 Compilers

I have tested following compilers using direct equivalence testing:

- msp430-gcc
  - mspgcc-3.2.3 dated 7th June,2005.
  - mspgcc-3.2.3 SVN head.
  - mspgcc-4.3.4 dated 8th September, 2009.
  - mspgcc-4.4.2 dated 25th October, 2009.
- llvm-msp430 SVN head.

4.1.3 Software

TinyOS is an open-source operating system designed for wireless embedded sensor networks [24]. TinyOS is a nonpreemptive, task oriented operating system. A queue for ready tasks is maintained and scheduler executes one task at a time to completion. TinyOS has a split-phase event-driven execution model to achieve concurrency. Some of the applications that were tested include MultihopOscilloscope, BaseStation, BlinkTask, MViz, Oscilloscope, RadioCountToLeds, RadioSenseToLeds, Sense, TestAdc, TestFtsp, TestOscilloscopeLQI and RadioStress for TinyOS 2.1 release.

Randprog is a random C program generator by Eide et al. [8]. Randprog generates nearly strictly conforming random C programs. Randprog supports various options to control the structure of the generated C programs. Following are some of the parameters that were varied for testing:

- Program size
- Number of functions
- Presence of structs
- Presence of arrays
- Presence of pointers

- Pointer depth
- 64-bit arithmetic

CIL C Intermediate Language is a high level representation and a set of tools for analysis and source-to-source transformation of C programs [20]. CIL was used to analyze the source code and identify the the global variables, volatile variables, and pointers in the test program.

4.2 TinyOS testing

I tested various TinyOS applications with msp430-gcc. The goal was to detect memory safety errors in TinyOS applications as well as to determine if TinyOS applications were miscompiled.

4.2.1 Unit of testing

Every single TinyOS task as well as an interrupt handler is considered a unit of testing.

4.2.2 Test procedure

The TinyOS scheduler code was modified to let the simulator know about the start and end of a TinyOS task and interrupt handler. Different binaries for a given TinyOS application are obtained by using different compilers and optimization flags. These binaries are executed in MSPSim along with COOJA network simulator, collecting a log of memory traces for each run. The simulator annotates the memory traces according to the task or the interrupt handler to which it belonged. The annotated traces are then parsed and separated according to the computation. These traces are checked for the violation of equivalence. If a violation is found, information about computation, compiler, optimization flags, memory traces and source of violation are displayed.

Our aim was to test as many TinyOS applications as we could in the hope of detecting previously unknown errors in TinyOS applications as well as the compilers used in build process.

4.2.3 Results

Table 4.1 lists the errors we found using direct equivalence testing that affect the TinyOS applications.

Some of the errors may not be termed as memory safety errors but they are all undefined behaviors in C and must be avoided. Next, we present detailed description of some of these bugs.

4.2.3.1 Out of bounds memory access

An out of bounds memory access error was discovered while testing the MultihopOscilloscope application of TinyOS.

As shown in Figure 4.1, values stored in variable `reading` range from 0 to 5. `Timer.fired()` event handler is executed periodically after `Read.readDone()` and resets `reading` to 0 when its value reaches 5. A synchronization error in MultihopOscilloscope application occasionally prevents `Timer.fired()` to be executed between two calls to `Read.readDone()`. In such a scenario, a call to `Read.readDone()` happened when `reading` = 5, leading to an out of bounds memory access. Memory location represented by `readings[5]` is a padding byte of a struct. At different levels of optimization, different values were written to the memory location `readings[5]`.

When checking computations at different levels of optimization for equivalence, input machine states were found equivalent for code in Figure 4.1. But output machine states failed equivalence check as some computations contained a write to the memory location with address same as `readings[5]`. The source of the error

Table 4.1. Bugs affecting TinyOS applications

Component affected	Type of error
MultihopOscilloscope Oscilloscope TestOscilloscopeLQI	Out of bounds array access
ADC	Portability Error
MViz	Read of uninitialized local variable

```
typedef struct oscilloscope {
  uint16_t version;
  uint16_t interval;
  uint16_t id;
  uint16_t count;
  /* 5 element array, indexed from 0 to 4 */
  uint16_t readings[5];
} oscilloscope_t;

/* Global instance of struct oscilloscope_t */
oscilloscope_t local;

/* Values range from 0 to 5 */
uint8_t reading;

void Read.readDone(error_t result, uint16_t data) {
    if (result != SUCCESS) {
      data = 0xffff;
      report_problem();
    }
    local.readings[reading++] = data;
}
```

Figure 4.1. Out of bounds memory access in MultihopOscilloscope application in TinyOS

was traced to `Read.readDone()`. This error had been reported and confirmed by TinyOS developers.

C being an type unsafe language, there is no language provided mechanism to detect and report a memory safety error at runtime. Instead, the machine state is mutated leading to corruption of data. This causes nonequivalence of computations and can be detected by direct equivalence testing.

4.2.3.2 Portability error

Every microcontroller is unique in its architecture with respect to accessing different regions of memory like hardware registers (memory mapped IO), RAM, and ROM. These regions of memory have different constraints on how data can be read or written. One such example is the ADC12 peripheral memory in the msp430F1611

microcontroller, the specification [13] for which states that, "*Addresses in this module should be accessed with word instructions. If byte instructions are used, only even addresses are permissible, and the high byte of the result is always 0.*"

The C99 standard [14] does not address such hardware specific concerns making it unclear if it is compiler's or user's responsibility to ensure compliance. We define a portability error as one that lies in this grey area that is outside the purview of the standard. Below is the detailed description of one such portability error that manifested itself in core TinyOS ADC component code.

Figure 4.2 shows how ADC12CTL0 register is accessed as an adc12ctl0 structure in the TinyOS code base.

```
/* The adc12ctl0 struct corresponds to the ADC12CTL0 register
 * with the struct member bitfields corresponding to bits that
 * make up the ADC12CTL0 register.
 */
typedef struct __nesc_unnamed4254 {
   volatile unsigned
   adc12sc : 1,
   enc : 1,
   adc12tovie : 1,
   adc12ovie : 1,
   adc12on : 1,
   refon : 1,
   r2_5v : 1,
   msc : 1,
   sht0 : 4,
   sht1 : 4;
 } __attribute((packed))  adc12ctl0_t;

/* Memory address of ADC12CTL0: 0x1A0 */
volatile unsigned int ADC12CTL0 __asm ("0x01A0");

/* Reading ADC12CTL0 as adc12ctl0 structure */
static inline adc12ctl0_t getCtl0(void )
{
  return * (adc12ctl0_t *)&ADC12CTL0;
}
```

Figure 4.2. ADC12CTL0 register access code from TinyOS ADC component

msp430-gcc compiles this code to the assembly shown in Figure 4.3.

According to the msp430F1611 specification [13], byte accesses to odd addresses in 16-bit peripheral modules are not allowed. The msp430-gcc compiler generates code that reads an odd address with a byte access which clearly violates the alignment as well as access rules as mentioned in the specification.

A discussion ensued between the compiler writers and TinyOS developers concluding that the compiler was correct in generating byte accesses to the structure. `getCtl0` type casts ADC12CTL0, a volatile unsigned int, to a packed structure of type adc12ctl0. Thus, information about the original type (int) and qualifier (volatile) is lost. Because of the packed attribute, the compiler may not assume that the structure will be word aligned. Also, since the return type is not int, the compiler is not required to generate word access. This error is interesting in that it remained undetected for two major releases of TinyOS.

4.2.3.3 Reading an uninitialized local variable

Reading an uninitialized variable is an undefined behavior in C. We found an instance of such undefined behavior in MViz application of TinyOS. As shown in Figure 4.4, `val` is an uninitialized local variable. It is passed by reference to `getEtx()` and `getParent()` to get the result back in `val`. Both these functions have error paths when the function returns without writing to `val`. When `val` is read in this context, it results in a read of an uninitialized local variable. The garbage value present on the stack is stored in the fields `local.link_route_value` and `local.link_route_addr` of the global struct `local`. This results in a failure of equivalence check as garbage values written are different at different optimization

```
getCtl0:
  mov.b &0x01A0, r14
  mov.b &0x01A1, r15
  swpb  r15
  bis r14, r15
  ret
```

Figure 4.3. Assembly code for getCtl0() as compiled by msp430-gcc

```
typedef struct {
   uint16_t reading;
   uint16_t link_route_value;
   am_addr_t link_route_addr;
 } mviz_msg_t;

 mviz_msg_t local;

 void Read.readDone(error_t result, uint16_t data) {
     uint16_t val;
     if (result != SUCCESS) {
       data = 0xffff;
       report_problem();
     }
     local.reading = data;
     call CtpInfo.getEtx(&val);
     local.link_route_value = val;
     call CtpInfo.getParent(&val);
     local.link_route_addr = val;
 }
```

Figure 4.4. Reading an uninitialized local variable

levels. The compiler, when passed the *-Wall* option that checks for uninitialized variables, did not generate a warning.

4.3 Random testing

To detect bugs in compilers, various phases of a compiler must be exposed to programs with varied and diverse C constructs as specified by the C standard [14]. The programs should be strictly confirming C programs. If a violation of equivalence is detected, then we can be sure that it is due to a compiler error. We use *randprog* to generate random C programs which are used for direct equivalence testing. We have tested various versions of msp430-gcc and llvm-msp430 with random programs.

4.3.1 Unit of testing

The whole random program, i.e., the `main()` is considered a unit of testing. The purpose of random testing is to test the compiler and as such, the random programs do not contain any interrupts. There are no behaviors in a random program that force us to choose equivalent points at a finer granularity.

4.3.2 Test procedure

Random programs are annotated to indicate the start and end of the computation to the simulator. The differential testing procedure, as used for TinyOS testing, is followed: binaries from different compilers and optimization levels, executing them in a simulator to obtain memory traces and then equivalence checking to find errors, if any.

I have tested the various versions of msp430-gcc and llvm-msp430 against thousands of random test cases generated by randprog, manual tuning various options in randprog to exploit the buggy phases of the compilers. My aim for TinyOS testing was to detect application and compiler bugs in TinyOS while Random testing was mainly used to expose bugs in compilation toolchain.

4.3.3 Results

Using direct equivalence testing and conforming random C programs generated by randprog, I was able to detect various correctness as well as volatile related errors in msp430-gcc, then stuck at version 3.2.3. Nature of random testing is such that a few typical compiler bugs manifest themselves repeatedly unless fixed. msp430-gcc was not being actively maintained and various bug reports went unheeded, in turn reducing the motivation to file more bug reports. These factors led to the discontinuation of random testing of msp430-gcc for the time being.

In the meantime, LLVM team [17] decided to work on a port for msp430 microcontrollers and I have been involved in the testing effort for the developmental llvm-msp430 compiler. Direct equivalence testing was quickly finding bugs in llvm-msp430 and the developers were responsive in fixing them. Next, I have

outlined some of the bugs that were found using direct equivalence testing in msp430-gcc and llvm-msp430.

4.3.3.1 Correctness error in msp430-gcc

msp430-gcc support for 64-bit arithmetic is error prone. Every test case generated for random testing involved a fair amount of 64-bit arithmetic and every case would fail equivalence check. Hence, I had to turn off 64-bit arithmetic support in the test cases generated by randprog.

Figure 4.5 shows the C source code, containing the bitwise right shift construct, that was miscompiled in every test case.

The miscompiled statement contains a 64-bit global variable `crc` that is bitwise shifted to right by 40 and the last 8 bits of the result are stored to `crc` again.

Figure 4.6 shows the assembly code generated by msp430-gcc version 3.2.3 at level -O1. A shift of 40 bits is broken into a shift of 32 bits and a shift of 8 bits. The compiler generates wrong code while shifting the result by 8 bits. As seen in Figure 4.6, content of register r12 is not shifted to right. This results in a wrong value being stored in `crc`. Interestingly, this incorrect behavior is exhibited by the compiler only at -O0 and -O1 levels of optimization. It generates correct code at higher levels of optimization.

```
/* Global variable */
int64_t crc = 0;

int main()
{
    ....

    /* Bitwise right shift a 64-bit variable */
    crc = (crc >> 40) & 0xff;

    ....
}
```

Figure 4.5. Sample C code that causes msp430-gcc 3.2.3 to emit wrong code in turn producing wrong results

```
/* Registers in msp430 micro controllers are 16 bits wide.
 */

/* Move the value of variable ‘crc’ present in
 * registers r8 - r11 to temporary registers r12 - r15
 */
mov r8,  r12
mov r9,  r13
mov r10, r14
mov r11, r15

/* Bitwise right shift of 40 is broken into 2 parts:
 * a shift of 32 bits and a shift of 8 bits. The code
 * below is for bitwise right shift by 32.
 */
mov r14, r12
mov r15, r13
clr r14
clr r15

/* The code below is for bitwise right shift by 8.
 * The correct code would have right shifted both
 * the registers r13 and r12. Instead only register
 * r13 is shift to right. Register r12 is never shifted
 * to right.
 */
clrc
rrc  r13
rra  r13
rra  r13
rra  r13
rra  r13
rra  r13
rra  r13
rra  r13
mov.b r12, r15
```

Figure 4.6. Incorrect msp430 assembly code generated by msp430-gcc at level -O1

4.3.3.2 Wrong code bug in llvm-msp430

The msp430 user guide [13] specifies seven addressing modes for 'br' instruction. The relevant ones are:

BR #LBL Branch to label LBL or direct branch (e.g., #0A4h). This instruction is implemented as `MOV @PC+,PC`.

BR LBL Branch to the address contained in LBL or an indirect branch. This instruction is implemented as `MOV X(PC),PC`.

BR &LBL Branch to the address contained in the absolute address LBL or an indirect address. This instruction is implemented as `MOV X(0),PC`.

llvm-msp430 emitted the assembly code containing the branch instruction `BR` using the indexed addressing mode. The branch instruction `BR` does not support indexed addressing mode in msp430 instruction set. The correct solution would be to use direct addressing mode for the `BR` instruction, as was intended.

As shown in Figure 4.7, the control flow is to be transferred from label `.L_A` to label `.L_B`.

The correct instruction would be

`BR #.L_B` instead of `BR .L_B`

using the direct addressing mode with the branch instruction `BR`.

The incorrect code, which used the indexed addressing mode, changed the semantics of the intended operation.

Desired behavior : PC = PC + offset

Actual behavior : PC = memory[PC + offset]

As such, control flow of the program changed resulting in differing outputs. These different outputs, when stored to the memory, led to failure of equivalence check.

```
/* Let there be a branch from label A to label B.
 * The msp430 assembly code would be as described below.
 */
.L_A
  br .L_B

...

/* Some code that does not fit in the offset range of
 * absolute jump. Hence branch instruction is used.
 */

...

.L_B
  mov #0, r15
  ret
```

Figure 4.7. msp430 assembly code for a large function that causes llvm-msp430 compiler to emit incorrect code with the branch instruction `BR` in it

4.4 Advanced compiler optimizations

Direct equivalence testing is effective in detecting bugs when common optimizations like function inlining, loop unrolling, constant propogation, subexpression elimination and others are used. Interprocedural optimizations that extend outside the computation under consideration limit the effectiveness of direct equivalence testing. For example, consider an interprocedural optimization that tries to reduce the code size of two TinyOS tasks. For the testing purposes, two TinyOS tasks are considered separate computations. This renders direct equivalence testing ineffective in this scenario, leading to false positives. If the scope of the interprocedural optimizations is within that of the computation, direct equivalence testing works fine.

For random testing of compilers, the whole program is considered a computation and thus, interprocedural optimizations do not affect the testing. Testing of TinyOS applications was also not affected as no false positives were reported.

4.5 Range of bugs found

Direct equivalence testing is effective in detecting wide array of bugs in application software and the compilation toolchain. It can uncover memory safety bugs like stack overflow, out of bounds access, buffer overflow and use of dangling pointers and wrong code and volatile related bugs in compilers. It can also detect logic errors in software, integer overflow bugs and concurrency bugs in case it leads to storage of differing values in memory.

Direct equivalence testing relies on executing the binaries and collecting the memory trace for equivalence checking. Thus, it cannot detect bugs that crash the compilation toolchain when presented with a valid program. Also bugs like nontermination due to deadlock/livelock, divide by zero, dereferencing a null pointer and latent bugs in the software that do not change the contents of the memory cannot be detected.

To illustrate further, consider the example in Figure 4.8.

The computation under consideration only returns a value of 5. It does not store this value in memory. If a compiler miscompiles the above program to return a value 4 instead of 5, direct equivalence testing would not be able detect this bug as the state of the computation being checked for equivalence remains unchanged.

Direct equivalence testing is effective in detecting any bug that eventually leads the computation under consideration to mutate its state (the contents of memory). Locating the bug in the source code is easy as direct equivalence testing collects low level information about the system. Since the memory reference and the variable name that leads to nonequivalence is known, a glance at the source and assembly code or a breakpoint in the debugger leads to the source of the error. Optimizations

```
int main()
{
  return 2 + 3;
}
```

Figure 4.8. A naive example to demonstrate the limitation of direct equivalence testing

like function inlining and loop unrolling make it more difficult to locate the source of the bug.

CHAPTER 5

RELATED WORK

Direct equivalence testing is an end-to-end technique to test system software for errors using equivalence checking and differential testing. This enables testing of the application software as well as the compilation toolchain.

Software testing is an integral part of the software development cycle. Several software testing techniques have been developed and used [3]. Comparatively, compiler testing, as a research area, has not received similar attention by the academia and the industry. Compilers are large, complex software systems and are not error free [8]. They must be tested aggressively.

McKeeman [18], Sheridan [23] and Eide et al. [8] have shown that *differential testing* with manual and random program generation is quite successful in finding compiler bugs. Differential testing is comparing the behavior of several comparable systems (C compilers) when presented with a series of input. A failure inducing test case is found when the behavior of compilers or the compiled test case differs. McKeeman [18] used randomly generated C programs to uncover bugs in compilers as well as the application software. Sheridan [23] used manual as well as randomly generated test cases to uncover bugs in several open source compilers and assemblers. Eide et al. [8] have used sophisticated test case generation techniques to uncover a special class of compiler bugs related to handling of volatile objects. Lindig [16] uncovered errors in various compilers in the way they handle the C calling conventions.

An important point to note is that all the above techniques used output comparison to determine the behavior of the test cases and were targeted towards desktop platforms like x86 and x64. To my knowledge, direct equivalence testing is the only technique directed towards finding bugs in cross-compilers that are used to compile

embedded software. Direct equivalence testing is similar to the above techniques in that it uses differential testing as the testing methodogy but equivalence checking is used instead of output comparison to determine the behavior of the programs.

Equivalence checking is a widely used technique in the electronic design automation industry to formally prove that two representations of circuit design exhibit identical behavior. The basic techniques used in most of the equivalence checking tools for circuit design are:

- Binary Decision Diagrams [1, 15]
- Symbolic Simulation [4]

There are have been efforts to verify the equivalence of low level embedded software by using the techniques of symbolic execution [5] and cutpoints [10].

Currie et al. [5] used symbolic execution to verify if two small blocks of assembly code were equivalent. Given two blocks of assembly code, symbolic expressions were built for each of them. Then they used decision procedures to determine if the two symbolic expressions were equivalent. Their technique could not handle sophisticated control flow analysis. By their own admission, their technique could not handle interrupts, was inscalable beyond a couple of hundred lines of C code and produced false positives, though they were fairly successful in verifying the equivalence of the small blocks of assembly code for DSP and Pentium processors.

Feng et al. [10] applied the concept of cutpoints for formal equivalence verification of combinational circuits to embedded software. The idea of cutpoints is very simple. Given two combinational circuits whose equivalence is to be verified, we look for points where the two circuits can be proved equivalent. The preceding equivalent logic in the circuit is replaced by a new primary input. When this process is applied recursively to reach the outputs, the two circuits have been proved equivalent. The authors used symbolic simulation to determine cutpoints in the software. Though they were able to speed up the simulation over the existing techniques, it inherited all the drawbacks of using symbolic execution, namely, lack of scalability and failure to address interrupt driven concurrency.

In comparison, direct equivalence testing addresses the problems of interrupt driven concurrency, compiler optimizations and semantics of volatile type qualifier while being able to test all the programming constructs in C language. It is also scales well for small to medium sized embedded systems.

CHAPTER 6

CONCLUSION

Embedded software applications as well as compilers used to compile them are not error free. This can cause the reliability and robustness of real time and safety critical systems to be suspect. Testing is an effective way to ensure quality and reliability of the software. I present Direct Equivalence Testing, a framework for detecting compiler and application errors in embedded software. The results indicate that the new technique is indeed effective in finding hitherto unknown compiler as well as application errors. Direct equivalence testing can potentially detect any error — application or compiler — that directly or indirectly results in different values being stored to the monitored section of the memory (Data/BSS sections) for equivalent computations. I have sucessfully detected following errors using direct equivalence testing:

- Compilation Errors like wrong code and volatile related errors in compilers like msp430-gcc and llvm-msp430.
- Programming Errors like out of bounds access, stack overflow and use of uninitialized local variables in TinyOS applications.
- Portability Errors in TinyOS applications.

The research contribution of this thesis is to apply the technique of direct equivalence testing to embedded software while solving the problems posed by interrupt driven concurrency, compiler optimizations and semantics of volatile type qualifier in C.

In theory, direct equivalence testing could be applied to software of any size, but there are practical scalability issues when the code size exceeds few thousand

lines. This is due to the overhead of instrumenting the large binaries and saving, parsing and comparing millions of memory traces. Future ideas to explore would be better representation of structures that scale, using symbolic execution to determine equivalence of machine states to reduce storage and compute overhead and test the systems that include dynamically allocated memory such as a heap.

REFERENCES

[1] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[2] Tamarah Arons, Elad Elster, Shlomit Ozer, Jonathan Shalev, and Eli Singerman. Efficient symbolic simulation of low level software. In *DATE '08: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 825–830, New York, NY, USA, 2008.

[3] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[4] Randal E. Bryant. Symbolic simulation—techniques and applications. In *DAC '90: Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 517–521, New York, NY, USA, 1990.

[5] David Currie, Xiushan Feng, Masahiro Fujita, Alan J. Hu, Mark Kwan, and Sreeranga Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *International Journal of Parallel Programming*, 34(1):61–91, February 2006.

[6] Edsger Dijkstra. *Technical Report: Notes on Structured Programming*. Department of Mathematics, Eindhoven University of Technology, Eindhoven, Netherlands, April 1970.

[7] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *EMNETS '04: Proceedings of the 1st IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004.

[8] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *EMSOFT '08: Proceedings of the 8th ACM International Conference on Embedded Software*, pages 255–264, New York, NY, USA, 2008.

[9] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik Osterlind, and Thiemo Voigt. MSPsim – an extensible simulator for MSP430-equipped sensor boards. In *EWSN '07: Proceedings of the European Conference on Wireless Sensor Networks*, Delft, The Netherlands, January 2007.

[10] Xiushan Feng and Alan J. Hu. Cutpoints for formal equivalence verification of embedded software. In *EMSOFT '05: Proceedings of the 5th ACM International Conference on Embedded Software*, pages 307–316, New York, NY, USA, 2005.

[11] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 1–11, New York, NY, USA, 2003.

[12] Elsa Gunter and Doron Peled. Model checking, testing and verification working together. *Formal Aspects of Computing*, 17(2):201–221, 2005.

[13] Texas Instruments Inc. Msp430x1xx family user's guide (rev. f). `http://www.ti.com/litv/pdf/slau049f`, February 2006.

[14] International Organization for Standardization. *ISO/IEC 9899:TC2: Programming Languages—C*, May 2005. `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`.

[15] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *DAC '97: Proceedings of the 34th Design Automation Conference*, pages 263–268, New York, NY, USA, 1997.

[16] Christian Lindig. Random testing of C calling conventions. In *AADEBUG '05: Sixth International Symposium on Automated and Analysis-Driven Debugging*, pages 3–11, September 2005.

[17] LLVM Team, University of Illinois at Urbana-Champaign. The LLVM compiler infrastructure project. `http://llvm.org/`, 2008.

[18] William McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100 –107, December 1998.

[19] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[20] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002.

[21] National Institute of Standards and U.S. Dept of Commerce Technology. The economic impacts of inadequate infrastructure for software testing. May 2002.

[22] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-level sensor network simulation with COOJA. In *SenseApp '06: Proceedings of the 1st IEEE International Workshop on Practical Issues in Building Sensor Network Applications*, Tampa, Florida, USA, November 2006.

[23] Flash Sheridan. Practical testing of a C99 compiler using output comparison. *Software: Practice and Experience*, 37(14):1475 – 1488, November 2007.

[24] tinyos.net. `http://www.tinyos.net`.