12-2014

Efficient and Reliable Simulation, Memory Protection, and Driver Generation in Embedded Network Systems

JIANNAN ZHAI
Clemson University, jzhai@g.clemson.edu

Efficient and Reliable Simulation, Memory Protection, and Driver Generation in Embedded Network Systems

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Jiannan Zhai
December 2014

Accepted by:
Jason O. Hallstrom, Committee Chair
Pradip K. Srimani
Brian A. Malloy
Jacob Sorber

Abstract

Embedded systems are widely used, from consumer electronics, to industrial equipment, to spacecraft. With embedded systems becoming more complex, new challenges are presented to application developers. In this dissertation, we focus on three of the most important: (i) Network simulation tools are widely used for sensor network testing and evaluation. Simulation performance affects the efficiency of the application developers who use these tools. The performance of a single host system represents a performance bottleneck for large-scale network simulation. A distributed simulator offering higher performance is needed to support fast, large-scale network simulation. (ii) Single event upsets (SEUs), which occur when a high-energy ionizing particle passes through an integrated circuit, can change the value of a single bit, causing damage and potentially catastrophic system failures. Modern SEU detection and correction approaches typically introduce additional hardware, increasing execution overhead and cost. Given the nature of resource-lean embedded systems, a software-based protection approach must be lightweight. (iii) Writing device drivers for serial-based peripherals is a repetitive task, given that microprocessors operate most such devices in the same way, issuing commands and parsing corresponding responses. A serial device driver generation tool must be capable of accommodating various microprocessors and devices with varying characteristics (e.g., UART settings, device response times, etc.), while producing drivers that offer performance at least as good as functionally equivalent, handwritten drivers. In this dissertation, we focus on the design and implementation of approaches to distributed sensor network simulation, embedded memory protection, and automated serial device driver generation.

The first challenge is to effectively emulate sensor network systems with high fidelity using a distributed simulation system. This is achieved by developing a distributed version of SnapSim [24], D-SnapSim, which runs on a cluster. D-SnapSim relies on multiple physical systems to achieve enhanced speed and scalability, while providing flexibility to execute on clusters of varying size and

computational power. The performance of D-SnapSim is evaluated as a function of network size, bitrate, and cluster configuration relative to SnapSim.

The second challenge is to protect embedded system memory from SEUs with a software-only approach. Traditional SEU prevention and correction strategies rely on hardware extensions to the target system. We present a software-only approach that detects and corrects SEUs in RAM. This is achieved by extending the AVR-GCC compiler to protect the system stack from SEUs through duplication, validation, and recovery. Four applications are used to verify our approach, and the time and space overhead characteristics are evaluated.

The third challenge is to automatically generate serial device drivers, eliminating the repetitive, error-prone work involved in serial device driver development. We present DriverGen, a configuration-based tool developed to provide automated serial device driver generation. Three applications are used to evaluate the performance of the generated drivers, both in terms of space and execution time. A user study is conducted to evaluate the usability of our tool in comparison with driver development in C.

Dedication

For family

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Jason O. Hallstrom, for his support, patience, and understanding during the past four and half years. Second, I would like to thank my committee members, Dr. Pradip K. Srimani, Dr. Brian A. Malloy, and Dr. Jacob Sorber, for the help and suggestions they provided on my research. Third, I want to thank Yvon, Sally, Hao, and Jit for their friendship and the wonderful memories they gave me. Fourth, I want to thank my friends for giving me so much joy and inspiration. Last but not least, I give my thanks to my parents and sister who are the source of my strength and happiness.

This was a long journey. Thank you Qing! Thank you "Nick"!

My work was supported by grants CNS-0745846 and CNS-1126344.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Embedded systems have become critical components of our daily lives, enabling cell phones, ATMs, car engines, spacecraft, and more. As embedded applications become increasingly complex, these systems present significant challenges. In this dissertation, we focus on three key problems faced by embedded application developers – simulation performance limitations, bit errors caused by single event upsets, and the repetitive, error-prone work involved in serial device driver development.

1.1 Problem Statement

In this section, we discuss the challenges to be addressed in the contexts of simulation, memory protection, and driver generation.

1.1.1 Network Simulation

Network simulation is widely used by developers and researchers in sensor network application development and experimentation. However, the efficiency of traditional single-system simulators, both single-threaded and multi-threaded [28] [59] [53], depends largely on the performance of a single host system, which becomes a performance bottleneck for large-scale network simulations. Due to the limited computational power that can be provided by a single physical system, the performance of sensor network simulation decreases significantly as the size of the simulated network increases, affecting the efficiency of development and experimentation.

1.1.2 Memory Protection

Embedded systems have become crucial components in spacecraft and other safety- and mission-critical equipment, controlling telemetry systems, command systems, attitude control systems, and more [57]. Given the high cost and vital importance of spacecraft rovers and satellites, as well as their increasing functionality and complexity, the hardware and software reliability requirements are stringent. Embedded software failures can cause serious consequences in this context. As an example, in 1996, the Ariane 5 spacecraft, which took 10 years and 7 billion (US) dollars to build, crashed due to the failure of the Flight Control Subsystem [31].

The environment outside the Earth's atmosphere is highly radioactive. The radiation is generated mainly by the sun and other stars and can cause damage to semiconductor devices [57]. One of the most common types of damage caused by radiation is the *single event upset* (SEU). Extremely small electronic components (i.e., tens of nanometers [9]) are used in modern integrated circuitry; the components carry little charge. As a result, one high-energy ionizing particle passing through an integrated circuit can release enough charge to change the state of a binary digit, causing a stored bit to change to its opposite value (i.e., a 0-bit can become a 1-bit, and vice-versa [57]). The damage caused by an SEU can range from system malfunction to system crash.

1.1.3 Driver Generation

Modern embedded systems are widely used in consumer electronics, healthcare equipment, industrial automation, automotives, etc. Over 10 billion embedded microprocessor were sold in 2008, with expected compound annual growth of 6.38% from 2008 to 2013 [56]. In developed countries, the average consumer has about 30 embedded microprocessors inside their cellphones, televisions, washing machines, and other common household devices [7]. Embedded systems have become critical components of almost every aspect of our daily lives, enabling our cellphones, ATMs, car engines, and more. Growth in the embedded systems market has led to the broad availablility of serial peripherals designed to limit the number of microprocessor pins that must be dedicated to peripheral control, increasing the number of peripherals a single microprocessor can use. This reduces the number of microprocessors needed in each system, simplifies the hardware design, and reduces the cost of prototyping and production.

Although serial-based peripherals offer a standardized communication interface and follow

a common command/response pattern, the commands, expected responses, and associated timings vary from one device to another. As a result, writing serial device drivers involves significant repetitive work. Moreover, energy efficiency is a key requirement of many embedded systems, making the timeout parameters associated with each command important factors affecting the amount of time a microprocessor spends in a low power state. Unfortunately, accurately measuring the time associated with sending commands and receiving responses can be time-consuming and error prone.

1.2 Research Approach and Contributions

In this section, we outline our approaches to addressing the challenges, as well as our contributions.

1.2.1 Network Simulation

The main challenge addressed by our work in the area of distributed network emulation focuses on extending SnapSim [24] by enabling execution across multiple physical systems. Our design of the distributed version of SnapSim, D-SnapSim, targets cluster environments and accommodates clusters of varying size and computational power. While clusters may experience significant network delays, increasing synchronization time among simulation processes, each machine offers additional computation capacity, offsetting some of the performance loss caused by those delays. A tool, the Simulation Dispatcher, is developed to automate the simulation process and integrate simulation results across the cluster. The experimental results show that D-SnapSim significantly improves the performance of SnapSim as network size and bitrate increase.

1.2.2 Memory Protection

In the memory protection component of the dissertation, our work focuses on extending the AVR-GCC compiler to protect the system stack from SEUs through duplication, validation, and recovery. Our approach injects assembly code into a target application to achieve memory protection without introducing additional hardware. After injection, each callee computes and saves the checksum of its caller's current stack frame and duplicates the caller's stack frame when the callee enters its function body. Before the callee returns, it verifies the stack frame of its caller using the saved checksum and overwrites the stack frame using the duplicate, if an SEU is detected. Our

approach changes the target system software and does not introduce additional hardware. Since our approach operates at the assembly level, it is language and application neutral. We validate the correctness of our approach and consider the overhead introduced, both in terms of space and execution speed.

1.2.3 Driver Generation

Finally, the driver generation component of the dissertation focuses on supporting automated serial device driver generation for embedded systems. The system runs on a desktop, which simulates the target microprocessor. A USB-to-serial interface chip, the FT232R [16], is used to transmit data between a desktop running the driver generation system and the target device. A configuration file is used to specify the commands to be sent to the device, along with the expected responses. Based on the configuration file, the system issues the specified commands to the target device and receives and parses the corresponding responses, simulating the embedded execution process. A hardware component is connected to the UART communication interface to measure the response time of each command, as well as the inter-byte time in each response. Based on the simulated execution and associated measurement results, the target device driver is generated. Three case studies are considered to validate and evaluate our approach, including a small user study to assess usability.

1.2.4 Contributions

Contribution 1. (i) We develop D-SnapSim, based on SnapSim. D-SnapSim runs on a cluster based on a configuration file, relying on multiple physical systems to achieve enhanced speed and scalability. (ii) We evaluate the performance of D-SnapSim using a series of standard sensor network applications from the TinyOS distribution [27].

Contribution 2. (i) We present an approach that protects the runtime stack by injecting assembly code at the beginning and end of each application function. (ii) We present an implementation of the approach, using the popular AVR architecture as a target. (iii) We verify the protection efficacy of our approach and evaluate performance in terms of space and speed overhead using three applications with different stack usage patterns.

Contribution 3. (i) We present an XML-based serial device driver configuration language

that generalizes the specification of a serial device driver, including the commands to be sent, the responses expected, the allowed response time of each command, and other salient parameters. (ii) We present a regular expression library used by the generated driver, which enables the driver to flexibly match responses, and to save desired information from within the responses. (iii) We present an approach that measures response times with precision on the order of 10 microseconds by monitoring data signals in the communication interface. (iv) We implement DriverGen, a configuration-based tool that accurately measures response time and automatically generates a specified serial device driver. (v) We evaluate DriverGen, considering the performance of generated drivers for three serial devices: an LCD display (WH2004A) [60], a WiFi chip (RN131) [39], and a cellular modem (GM862) [51]. (vi) Finally, we conduct a user study to assess the usability of our approach.

1.3 Dissertation Organization

The remainder of the dissertation is organized as follows. Chapter 2 presents background material related to network simulation, memory protection, and driver generation. Chapter 3 presents the distributed network simulation contributions. Chapter 4 presents the software-only SEU protection contributions. Chapter 5 presents the serial device driver generation contributions. Chapter 6 discusses elements of related work. Finally, Chapter 7 concludes with a summary of contributions.

Chapter 2

Background

In this chapter, we present the relevant background material on network simulation, memory protection, and driver generation. We first present an overview of SnapSim, an optimistic sensor network simulator. Next, we present an overview of the AVR architecture, the associated function call process, and the AVR toolchain. Finally, we provide an overview of modern embedded system peripherals and serial device drivers.

2.1 Network Simulation

In a sensor network, messages transmitted between nodes introduce dependencies between transmitters and receivers. An example of such a dependency is shown in Figure 2.1. A and B are two simulated nodes that transmit at time tx_A and tx_B, respectively. If the process that simulates node B runs faster than the process that simulates node A, B will pass tx_A before A reaches this point. As a result, B will miss the data transmission at tx_A. To avoid such problems, simulation

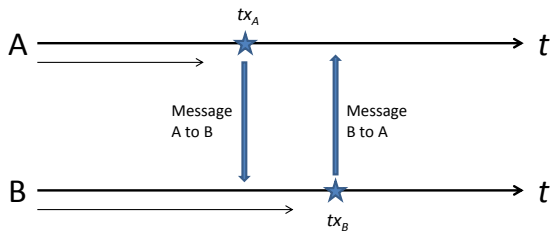


Figure 2.1: Distributed Simulation Dependencies (adapted from [24])

process synchronization is used to ensure the validity of the simulated execution sequence.

SnapSim is based on a probing and backtracking synchronization strategy. For a given simulated time t, the *nearest transmission time* of a node is defined as the earliest simulated transmission time of the node at or after time t. The *global nearest transmission time* is defined as the earliest simulated transmission time across all nodes in the network at or after time t [24]. SnapSim uses probing executions to find the nearest transmission time for each simulated node. To ensure the simulation is valid, the nodes that have passed the global nearest transmission time must be backtracked.

During simulation, once the nearest transmission time is identified, the associated process broadcasts its nearest transmission time and waits for other nodes. Once every simulated node reaches and broadcasts its nearest transmission time, the earliest time is selected as the global nearest transmission time. Each node that passed the global nearest transmission time backtracks to the last global transmission time and re-executes to correct for the missed transmission.

SnapSim is implemented based on the popular Avrora simulator. Results show that SnapSim improves the performance of Avrora by 2 to 10 times for typical sensor network applications [24]. However, SnapSim runs on a single system and relies on multiple threads to achieve parallel network simulation. As simulated network size increases, the limited computational capacity available on a single physical system becomes a bottleneck for simulation performance. We implement D-SnapSim based on SnapSim. D-SnapSim runs on a cluster and relies on multiple physical systems to achieve enhanced speed and scalability.

2.2 Memory Protection

While our approach is architecture neutral, our implementation is based on the Atmel AVR toolchain and focuses on AVR microprocessors. In this section, we survey the AVR architecture, function call process, and the AVR Toolchain.

2.2.1 AVR Architecture

AVR microprocessors are based on a modified Harvard architecture [1], which stores instructions and data in physically separate memories, flash memory and SRAM, respectively. Instructions and data are accessed concurrently through separate memory buses. Flash memory is non-volatile

Figure 2.2: AVR RAM Map

and offers high capacity, but slow access speed, and is used to store executable programs composed of AVR instructions. The SRAM is volatile and offers low capacity, but fast access speed, and is used to store data used by the executable programs at runtime. The ATmega644 includes 64KB of flash memory, 4KB of SRAM, a 16-bit instruction bus, and an 8-bit data bus.

AVR SRAM. The on-board SRAM of the ATmega644 has an address range of 0x0100 to 0x10FF, as shown in Figure 2.2. The SRAM is partitioned into sections, each used to store different types of data. The *.data* section is used to store initialized static variables and global variables. The *.bss* section is used to store uninitialized static and global variables. The pre-allocated SRAM usage is the sum of the sizes of the .data and .bss sections. The remaining space in SRAM is shared by the heap and stack sections. The *heap* section is used to store dynamically allocated memory, e.g. when *malloc()* is called [19]. The heap grows "upward", towards the higher address range. The *stack* section is used to store return addresses, actual parameters, conflict registers, local variables, and other information. The stack grows "downward", from *RAM_END*, address 0x10FF, towards the lower address range.

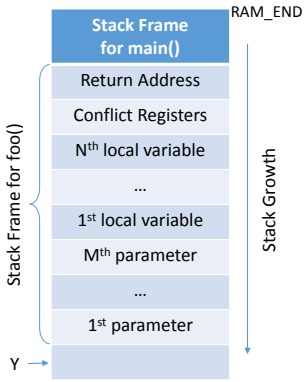


Figure 2.3: AVR Stack Frame

Stack Frame. The stack consists of stack frames, each corresponding to a function call. Each stack frame is created when a function is called, and freed when the function returns. For example, as shown in Figure 2.3, when the *main* function calls the *foo* function, a stack frame will be created for *foo*. First, the return address of *main* will be pushed to the stack, followed by the conflict registers. Next, the local variables and parameters will be pushed to the stack, in reverse order of declaration. The stack frame spans the return address through the first parameter. The stack frame pointer, Y, now points to the next available address in the stack. When *foo* finishes execution, the stack frame will be freed,

and the stack frame pointer will point back to the position where the return address of the previous stack frame was stored.

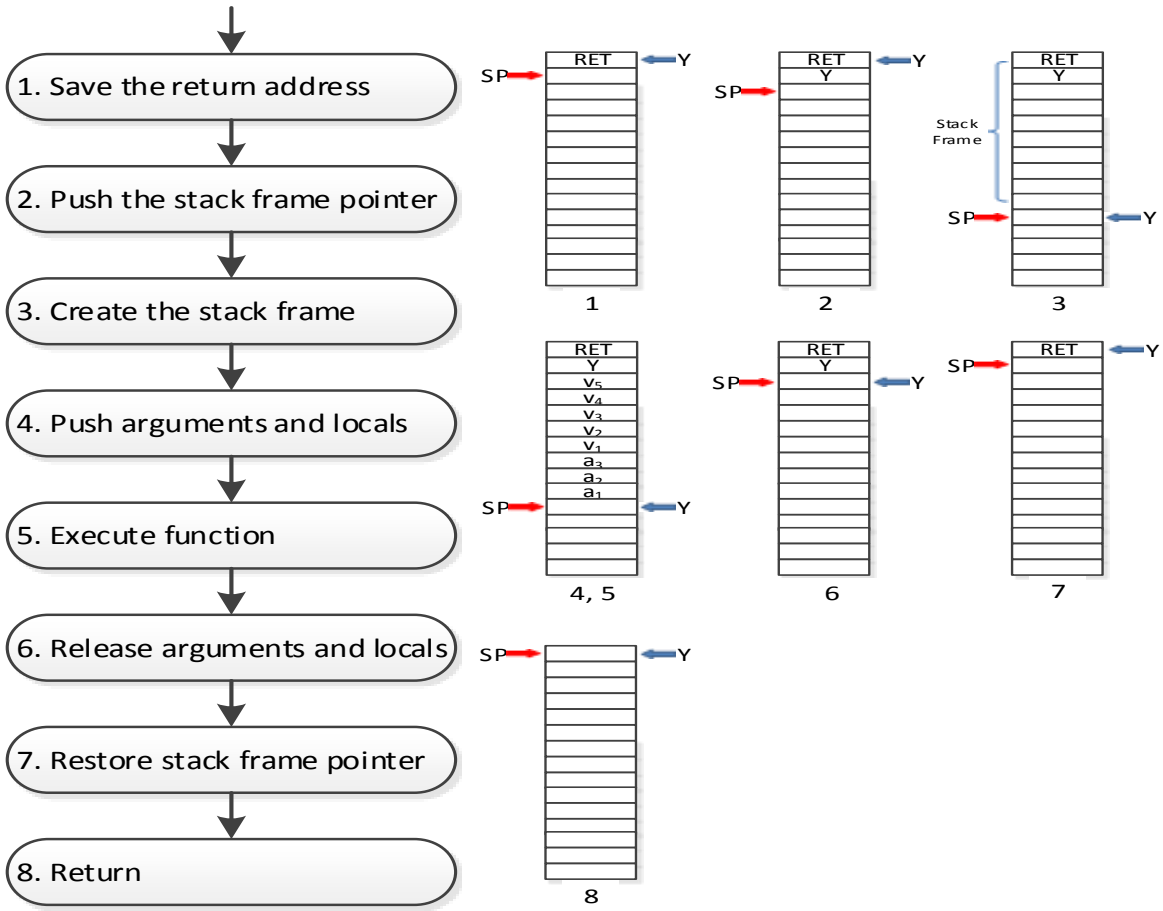
Registers. AVR microprocessors have two types of registers, general-purpose registers and I/O registers.

General-purpose registers are used for arithmetic operations, such as adding, subtracting, and comparing numbers, as well as indexing and setting long jump destinations. The ATmega644 has 32 general-purpose registers, R1 through R32, which are mapped into the first 32 locations in RAM, and can be directly used in assembly commands. Some general-purpose registers are used for special purposes; for example, R29 and R28 store a 16-bit address, the Y pointer, to indicate the top of the current stack frame. (The use of the Y pointer will be considered further in the next subsection). The use of these registers is compiler-dependent. For example, AVR-GCC uses R24 and R25 to store the return value of each function call. We attempt to limit the number of registers manipulated by our approach to reduce the cost of saving and restoring the conflict registers.

The I/O registers are used to control the internal peripherals of the AVR microprocessor. The ATmega644 has 64 I/O registers, mapped into the next 64 locations of SRAM, *0x20* through *0x5F*. Again, some I/O registers are used for special purposes; for example, AVR-GCC uses *0x3E* and *0x3D* as the stack pointer (SP), which indicates the current top of the stack.

2.2.2 Function Calls

All function calls follow the same process and use the system stack to perform most operations, as illustrated in Figure 2.4. Figure 2.4a summarizes the execution process when a function is called, and Figure 2.4b illustrates the associated stack changes after each operation is performed. Each rectangle represents two bytes in the stack. The numbers below each stack denote the operation(s) that changed the stack. `SP` denotes the stack pointer, and `Y` denotes the stack frame pointer. When a function is called, the return address is automatically pushed onto the stack by one of the function call instructions, `call`, `rcall`, or `icall` (step 1). After the stack frame pointer is pushed (step 2), the stack frame of the function is created by changing the stack pointer and stack frame pointer (step 3). The arguments and local variables are then pushed onto the stack (step 4), and the function begins executing (step 5). The arguments and local variables are released after the function finishes its execution (step 6), and the stack frame pointer is restored (step 7). Finally, the



(a) Function Execution Process

(b) Stack Changes

Figure 2.4: Function Execution

function returns (step 8). The return address is popped and used when one of the function return instructions, `ret` or `reti`, is called.

2.2.2.1 Atmel AVR Toolchain

The Atmel AVR toolchain is a collection of tools used to generate executable programs for AVR microprocessors. The toolchain consists of the following tools.

- *avr-gcc*, an extension of GNU GCC, is a cross-compiler, which translates high-level C or C++ code to assembly code for AVR microprocessors.
- *avr-as* is an assembler, which translates AVR assembly code to an object file.
- *avr-ld* is a linker, which uses a linker script to combine object modules into an executable image

suitable for loading into the flash memory of an AVR microprocessor. By using a customized linker script, the default locations and sizes in SRAM can be changed. New memory sections may also be added [5].

- *avr-libc* is a standard C library, which contains standard C routines, as well as additional AVR-specific library functions.

As a matter of convenience, the AVR toolchain can be used to compile, assemble, and link C programs in a single command. However, in order to modify the assembly code of an AVR application and use a customized linker script, these steps are performed individually in our implementation.

AVR GCC provides 5 optimization levels, -O0, -O1, -O2, -O3, and -Os, each enabling different optimization mechanisms. The exception is -O0, which offers no optimization [21]. Our approach is based on modifying unoptimized assembly code generated with the -O0 option. This option makes it more convenient for developing, debugging, and evaluating our approach.

2.3 Driver Generation

The growth in the embedded systems market has led to the broad availability of serial peripherals designed to limit the number of microprocessor pins that must be dedicated to peripheral control, increasing the number of peripherals a single microprocessor can use. This reduces the number of microprocessors needed in each system, simplifies the hardware design, and reduces the cost of prototyping and production.

Serial-based peripherals are among the most popular peripherals used in embedded systems. They offer a simple, standardized communication interface, following a simple common command/response pattern. A microprocessor sends a command to a serial device, which performs operations based on the received command. The serial device then sends a response back to the microprocessor, indicating the result of the command and the status of the device. However, the commands, expected responses, and allowed timings vary from one device to another. As a result, writing serial device drivers involves a significant amount of repetitive work. Moreover, since energy efficiency is a key requirement of many embedded systems, timeout parameters associated with each command are important factors affecting the amount of time a microprocessor spends in a low power state. Unfortunately, accurately measuring the time associated with these communications can be

time-consuming and error-prone.

Motivated by the repetitive structure of most serial device drivers and the importance of accurate timing, our approach automates the driver generation process based on a configuration file. To enable the driver to flexibly match responses, and to save desired information from within the responses, a regular expression library is created and used in the generated driver.

Chapter 3

Network Simulation

In this chapter, we present the design, implementation, and evaluation of a distributed version of *SnapSim*, *D-SnapSim*, which runs on a cluster. D-SnapSim extends SnapSim by enabling execution across multiple physical systems. Threads executing on the same system communicate via shared variables. Communication between processes executing on different systems is handled via Java Remote Method Invocation (RMI). The implementation is designed to balance thread use and distribution to maximize cluster resources. As a result, overall performance is maximized.

3.1 Distributed Execution

D-SnapSim is realized as a network of SnapSim insances. Figure 3.1 illustrates a simulation consisting of 16 simulated nodes; two possible cluster configurations are shown. The first configuration contains 4 physical machines, each executing an instance of SnapSim simulating 4 nodes. This configuration balances the simulation load equally across the cluster. Now consider a scenario in which some machines offer less processing power than others. If the load is balanced, as in the first configuration, faster machines will waste time waiting to synchronize with slower machines, reducing simulation speed. To avoid this, we configure slower machines to simulate fewer nodes. In the second configuration shown in the figure, SnapSim instances 3, 4, and 5 simulate 4 nodes each; SnapSim instances 1 and 2 simulate 2 nodes each.

Each SnapSim instance repeatedly identifies and broadcasts its candidate global nearest transmission time. When a SnapSim instance receives broadcast messages from all other SnapSim

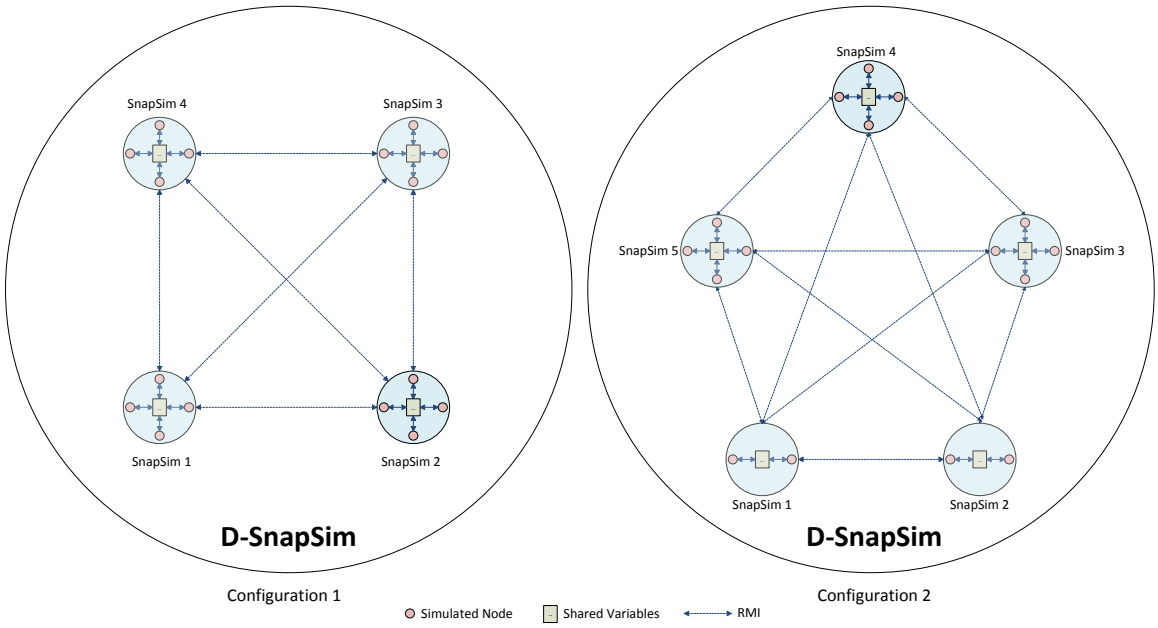


Figure 3.1: D-SnapSim Simulation of 16 Nodes Under 2 Independent Configurations

instances, the lowest value of all the candidate times, including the one discovered in its own search, becomes the global nearest transmission time.

Each broadcast message consists of the candidate global nearest transmission time and the simulated packet that will be transmitted at that time. Once a candidate global nearest transmission time becomes the global nearest transmission time, the packet becomes the global nearest packet, transmitted at the global nearest transmission time.

3.2 Inter-System Communication

An inter-system communication component, implemented based on Java Remote Invocation (RMI), is introduced to handle communication among simulated nodes executing on different systems. In SnapSim, when a node synchronizes with other nodes, it compares its nearest transmission time with the current (candidate) global nearest transmission time and updates the global nearest transmission time if its nearest transmission time is earlier. In D-SnapSim, the inter-system communication component is used when a node synchronizes with nodes executing on other systems. The following key functions are provided by the inter-system communication component.

- *Init*: This function is used to initialize the RMI component, and to create an RMI registry so

that the SnapSim instance executing on a given system is reachable by the instances executing on other systems.

- *Report*: This function is used to handle simulated communication, and also reports the local nearest transmission time of the executing node to the simulated nodes executing on other systems to identify the global nearest transmission time. The packet to be transmitted by the simulated node at its nearest transmission time is also included in the communication.

3.3 Distributed Synchronization

In our first approach to synchronizing nodes executing on different systems, each simulated node reports directly to all other simulated nodes when it reaches a transmission point. After reporting, the node waits until all other nodes have reported. Each SnapSim instance maintains a global variable that keeps track of the number of nodes that have reported. A node stops waiting and continues executing once the report count is equal to the number of simulated nodes, indicating that all simulated nodes have reported. Although this approach is easy to implement, the number of synchronization messages (RMI messages) introduced per synchronization is equal to the number of simulated nodes. Since network speed significantly affects simulation efficiency, requiring each node to synchronize directly with other nodes reduces simulation speed.

In our second approach, each simulated node reports only to the nodes simulated on the same system. Once all nodes executing on the same system have synchronized, the (elected) local nearest transmission time and associated packet are broadcast to all other systems. To achieve this, instead of maintaining a variable that keeps track of the number of nodes that have reported, each SnapSim instance tracks the number of nodes that have reported which are simulated on the instance's system, as well as the number of systems that have reported. Once the number of local nodes which have reported reaches the number of locally simulated nodes, all the local nodes pause execution, and the local (elected) nearest transmission time and associated packet are broadcast to other systems. Once the number of reported systems reaches the total number of systems, all nodes continue executing until the next transmission point is reached. Using this approach, the number of synchronization messages is reduced to the number of systems used in the cluster.

3.4 Simulation Dispatcher

To initialize D-SnapSim on a computing cluster and integrate simulation results across the cluster, a *Dispatcher* application is used. The Dispatcher controls the number of SnapSim instances to initialize, as well as the number of nodes to be simulated within each instance.

When D-SnapSim runs for the first time on a new cluster (or a new version of D-SnapSim is used), a script is used to load the SnapSim instances (jar files) on the cluster. To coordinate startup activities across the cluster, each SnapSim instance reports to the Dispatcher at startup and begins querying the Dispatcher to determine if it has received reports from all other SnapSim instances. The Dispatcher initiates simulation after all SnapSim instances have reported. The Dispatcher also returns a list of the active SnapSim IP addresses to each SnapSim instance. (At startup, the instances are disconnected; they have no knowledge of other SnapSim instances.)

When a SnapSim instance completes its simulation, it reports its completion status and sends simulation results to the Dispatcher, including performance statistics (e.g. simulation time, synchronization overhead time incurred). The Dispatcher terminates the instances after all of the simulation results have been received. Finally, the Dispatcher generates a simulation performance report containing the simulation time, synchronization time, number of packets sent, and number of rollbacks that occurred.

The Dispatcher is not involved in the simulation process unless it receives an exception report. In such a case, the Dispatcher reports the exception and either restarts or terminates the SnapSim instances, as specified in the configuration.

3.5 Evaluation

We now present the experimental results for D-SnapSim. All experiments were conducted on a modest PC cluster, with each machine configured with an Intel Core i7 2600 processor and 8GB of RAM. The JVM heap size was set to 64MB. The cluster was connected using a Cisco 3560G series 1G Ethernet switch. To limit I/O impacts, the data monitors which display simulation results were disabled.

3.5.1 D-SnapSim Performance Evaluation

Simulated applications with a high bitrate require a larger the number of synchronizations. We focus on ***XnpCount***, the application with high bitrate. XnpCount is used in network reprogramming and continuously sends messages to the network. Since typical embedded applications operate at a lower bitrate, the typical performance is better than the presented results. We evaluate D-SnapSim on networks of size 2, 4, 8,..., 1024, for 100 simulated seconds.

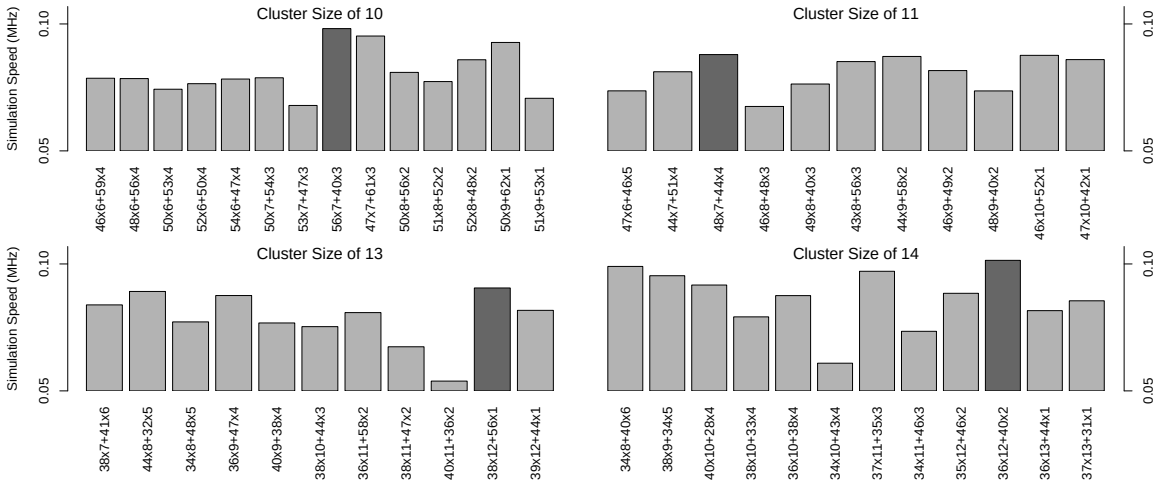


Figure 3.2: Simulation Performance Across Configurations (network size = 512)

For each network, there are multiple configuration options (i.e. cluster size, number of SnapSim instances, number of simulated nodes per SnapSim instance). We group the configurations based on cluster size. For example, a network of size 512 might be simulated using: *(i)* 16 SnapSim instances, each simulating 32 nodes. This configuration is denoted by 32x16. *(ii)* 10 SnapSim instances, 6 simulating 48 nodes, 4 simulating 56 nodes. This configuration is denoted by 48x6+56x4. We first identify the configuration that yields the best performance for each cluster size. That configuration represents the optimal configuration for that cluster size. To accomplish this, we experimentally search the configuration space. Figure 3.2 summarizes the experimental results of a simulated network of size 512, using 10, 11, 13, and 14 physical machines, respectively. Figure 3.3 summarizes the experimental results of a simulated network of size 1024, using 10, 11, and 14 physical machines, respectively. In both cases, the x-axis represents the configurations, and the y-axis represents simulation speed, in MHz. (The speed axis is logarithmically scaled in all graphs). Each highlighted bar represents the configuration that yields the best performance.

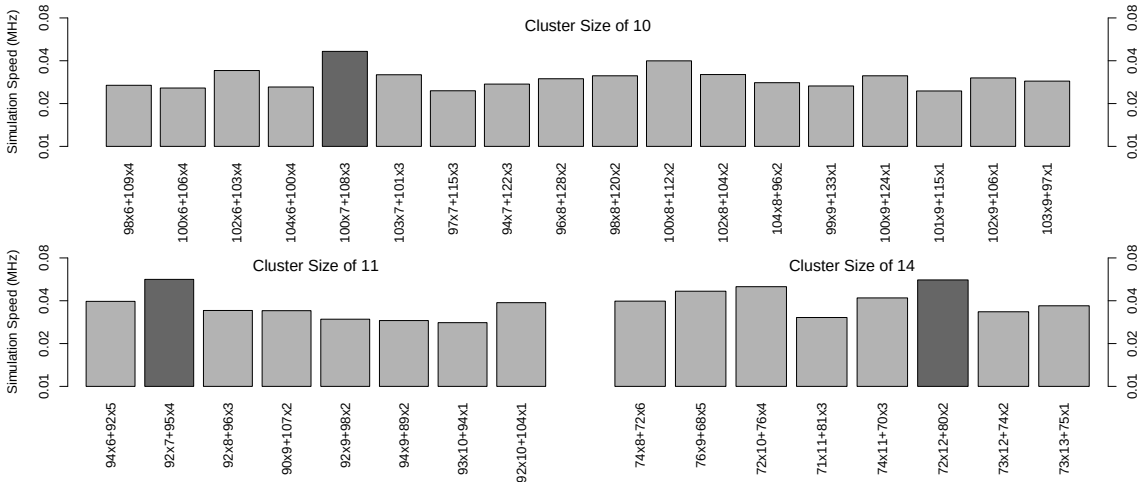


Figure 3.3: Simulation Performance Across Configurations (network size = 1024)

We next identify the optimal cluster size by comparing performance across all cluster sizes based on their optimal performance (calculated in the previous step). The experimental results are summarized in Figure 3.4. The x-axis represents cluster size, and the y-axis again represents simulation speed, in MHz. The "whiskers" represent worst-case configuration performance for each cluster size. The highlighted bar represents a cluster size of 14, yielding the best performance for a simulation of 512 nodes. As the figure illustrates, when the cluster size is small, performance is relatively low due to a lack of processing power. As cluster size increases, the additional machines improve performance. As more machines are added, however, network delays become a significant factor, decreasing performance.

Synchronization delays among threads are less significant than networking delays among processors. Our goal is to simulate as many nodes as possible on each system, while utilizing the cluster's computation capacity. The desired outcome balances small thread synchronization delays with the power of distributed computing.

Based on the identified optimal configuration, Figure 3.5 summarizes the performance of D-SnapSim, using SnapSim as a baseline for comparison, across varying simulation scales. The x-axis represents network size, and the y-axis represents simulation speed, in MHz, again logarithmically scaled. In small networks (<8), SnapSim runs faster than D-SnapSim because the i7's quad core processor is able to simulate 8 nodes in parallel without significant delays. When the network size reaches 8, D-SnapSim starts to outperform SnapSim. Further, its performance declines less rapidly as a function of network size compared to SnapSim. This performance decline is mainly due to

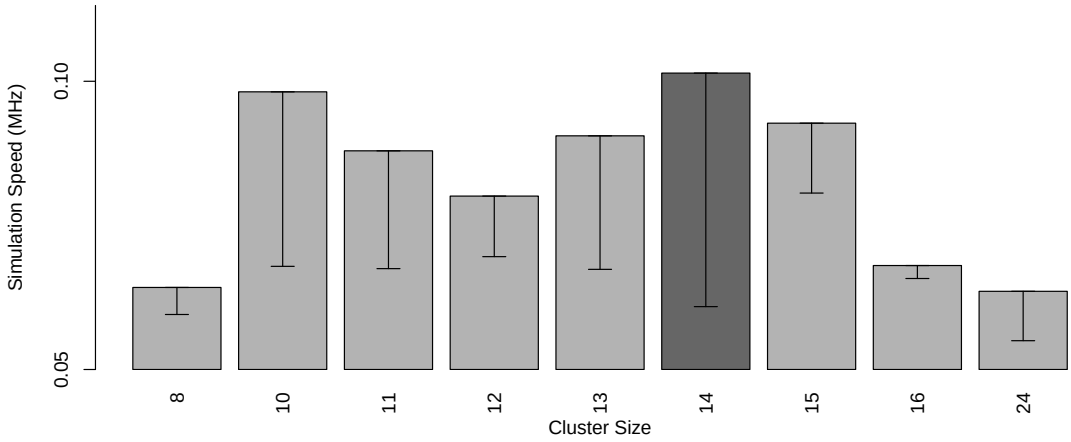


Figure 3.4: Simulation Performance (network size = 512)

an increase in synchronization time among SnapSim instances. Physical machines also experience significant network delays. At the same time, however, each machine offers additional computation capacity, offsetting some of the performance loss caused by those delays.

3.5.2 D-SnapSim Synchronization Overhead

We also evaluate the synchronization overhead, in terms of execution time, incurred by D-SnapSim. Figure 3.6 summarizes the results compared to SnapSim. The x-axis represents network size, and the y-axis represents synchronization overhead, measured in terms of the percentage of execution time spent synchronizing SnapSim instances. The figure shows that SnapSim's synchronization overhead increases significantly, from about 20% to 90%, as network size increases from 8 to 1024. In contrast, D-SnapSim's overhead is less significant and increases at a slower rate. The overhead of D-SnapSim increases from about 55% to 65%, as network size increases from 8 to 1024.

Synchronization overhead in D-SnapSim consists of two components: synchronization among local threads, and synchronization among threads on different systems. The latter naturally consumes more time due to network delays incurred between systems. As a result, the most significant synchronization overhead in D-SnapSim stems from physical delays. As simulated network size increases, physical network delays increase, causing the synchronization overhead to increase.

We now explore the relative difference between local and cluster synchronization overhead

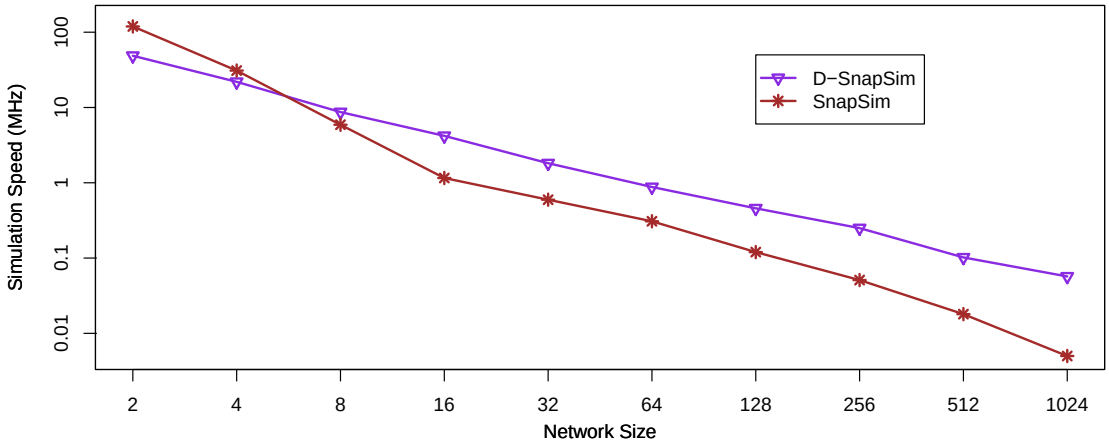


Figure 3.5: Performance vs Network Size (network size = 2, 4,..., 1024)

in D-SnapSim. As network size increases, we increase the cluster size to increase computation capacity. This in turn contributes to increased networking delays, increasing cluster synchronization overhead. Figure 3.7 summarizes the differential impacts on overhead as network size increases. The x-axis represents network size, and the y-axis represents synchronization overhead as a percentage of execution time. The results show that as network size increases, cluster synchronization is the primary bottleneck, as expected. Beyond a network of 32 nodes, cluster synchronization overhead represents most of the overhead incurred during execution.

In total, the results collected from the local cluster reflect 453 hours of physical execution time to simulate 5.42 hours of simulated time across networks of size 2, 4, 8,..., 1024.

3.5.3 Discussion

The results summarized in Figure 3.4 are surprising. We would expect that as cluster size increases, simulation performance would follow the same trend — to a point. After the optimal distribution point is reached, performance should begin to decrease as networking delays have a more significant impact. However, this is not reflected in the summarized results. Between the two peaks at 10 and 14, performance dips. The most likely explanation is that the test cluster was not accessed exclusively; utilization noise was introduced by other users, resulting in performance anomalies.

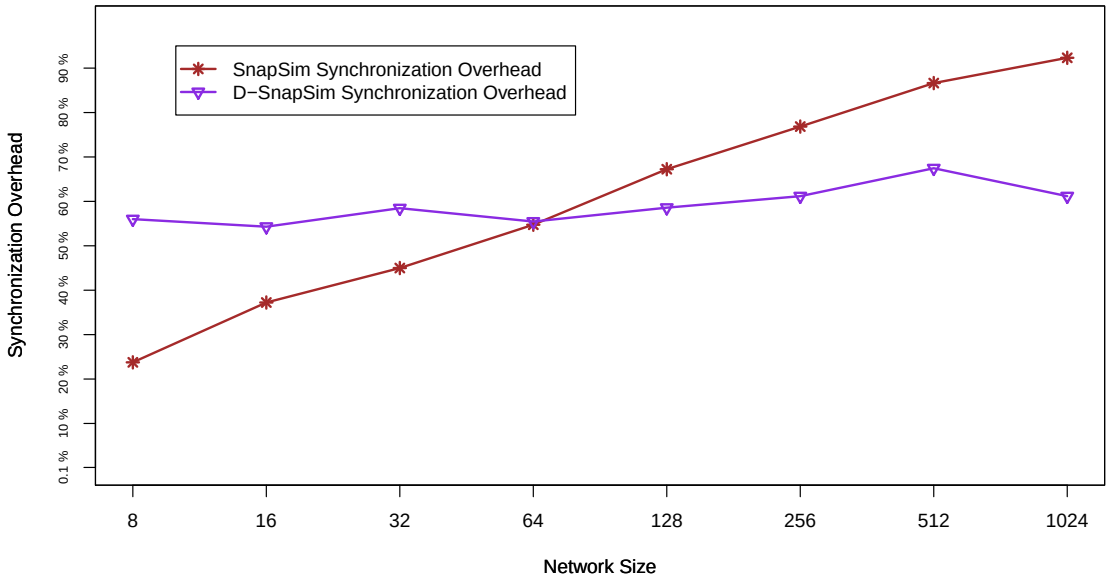


Figure 3.6: Synchronization Overhead

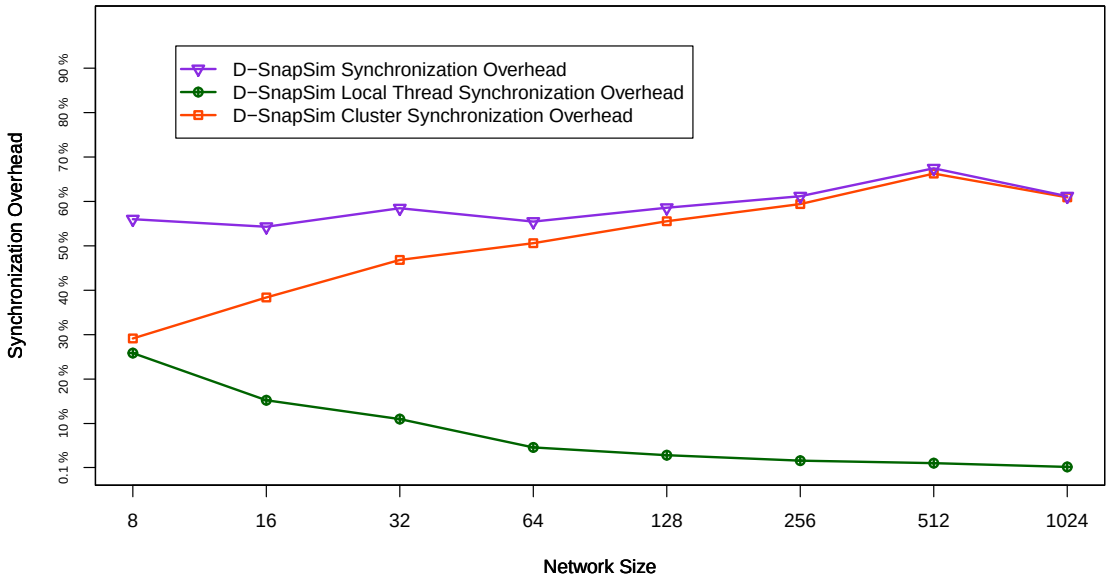


Figure 3.7: Synchronization Overhead

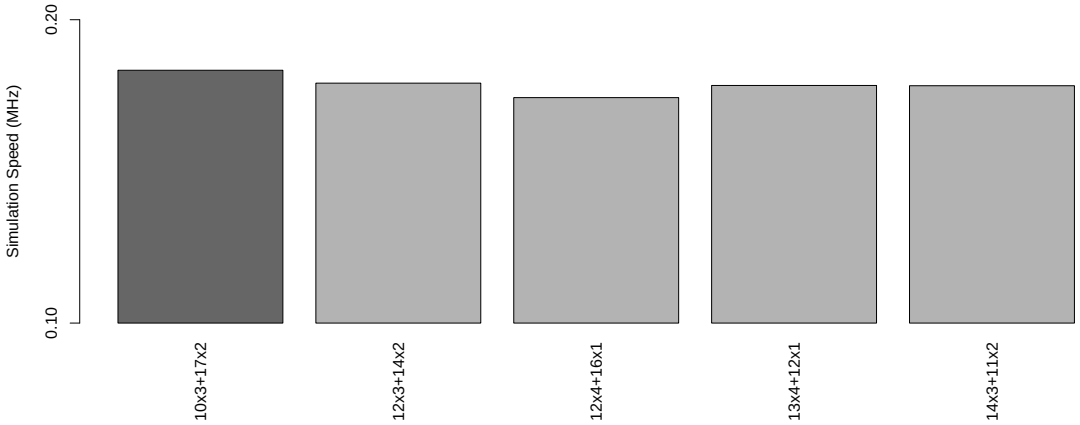


Figure 3.8: FutureGrid Performance (network size = 64)

Environment	***Avg. Simulation Speed***	***Avg. Thread Sync. Overhead***	***Avg. Cluster Sync. Overhead***
Local Lab	2.84 MHz	27%	41%
FutureGrid	0.68 MHz	29%	59%

Table 3.1: Simulation Performance Comparison (network size = 2, 4, 8, 16, 32, 64)

To address this issue, we ran a set of trials on FutureGrid [17], a free experimental cloud service provided to the research community. However, the performance was not what we had hoped. We ran the simulation experiments using the same setup as on our local systems, except using virtual machines created within FutureGrid. As before, we first simulate each network size with different configurations to find the optimal configuration for that size. For example, Figure 3.8 summarizes the results of a simulated network of size 64 using 5 physical machines. The x-axis represents the configurations, and the y-axis represents simulation speed, in MHz. The highlighted bar represents the configuration that yields the best performance for a simulation of 64 nodes. Experimental results for network sizes ranging from 2 to 64 simulated on FutureGrid are summarized in Figure 3.9. The x-axis represents network size, and the y-axis represents simulation speed, in MHz. The performance on FutureGrid decreases as network size increases, similar to the performance on the local cluster, but on FutureGrid, the system runs slower than SnapSim. These relatively poor results are due to a lack of computational capacity available via FutureGrid virtual machines, as well as significant network latency among the virtual machines. Table 3.1 compares simulation speed and synchronization overhead results from experiments conducted on our local lab systems and FutureGrid. These results show that both the local thread synchronization overhead and the

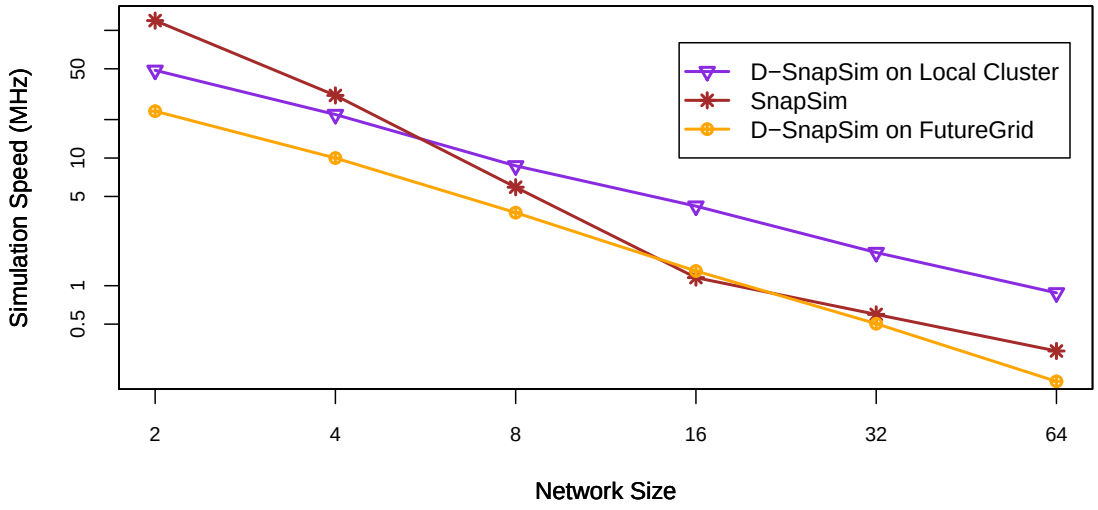


Figure 3.9: FutureGrid Performance (network size = 2, 4, ..., 64)

cluster overhead experienced when using FutureGrid were more significant than when using our local systems. As a result, we have focused the presentation on results collected from the local cluster.

3.6 Summary

In this chapter, we presented an approach to enabling large-scale embedded network simulation on a cluster. We first described the design of D-SnapSim, a distributed simulator that runs on a cluster, emphasizing the distributed execution and synchronization mechanisms. We next described the Simulation Dispatcher component used to automate the simulation process and collect simulation results. Finally, we presented detailed experimental results measuring the performance and associated synchronization overhead of D-SnapSim relative to SnapSim. The results show that D-SnapSim effectively utilizes the computational power of a cluster and significantly improves the performance of SnapSim as network size and bitrate increase.

Chapter 4

Memory Protection

In this chapter, we present the design, implementation, and evaluation of a new approach to protecting embedded system memory from SEUs. We focus on the protection of the runtime stack, and therefore make the following assumptions: i) Flash memory and registers are not affected by SEUs. ii) Only one SEU will occur during a given function execution. This assumption is well-justified. It is rare for more than one bit to be upset simultaneously; this occurs in only 5 to 6 percent of bit flip errors [54].

Our approach is designed to align with the NASA coding standards for C applied in space projects [38]: First, dynamic memory allocation is not allowed, so the heap section in RAM is not used. However, for the sake of completeness, we consider the possibility of a non-empty heap in our approach. Second, the `goto` statement is not allowed. Finally, each function should have fewer than 60 lines of code, making the execution time of each function relatively short.

Our approach protects the system stack by introducing auxiliary assembly code. The new code is injected at both the beginning and end of each function and handles CRC calculation, CRC comparison, and memory duplication. When a function is called, the code injected at the beginning of the call calculates the CRC of the caller's stack frame and saves both the CRC and the stack frame. Before the callee returns, the code injected at the end of the function calculates the CRC of the caller's stack frame again, compares it with the saved CRC, and restores the caller's stack frame if the CRCs do not match.

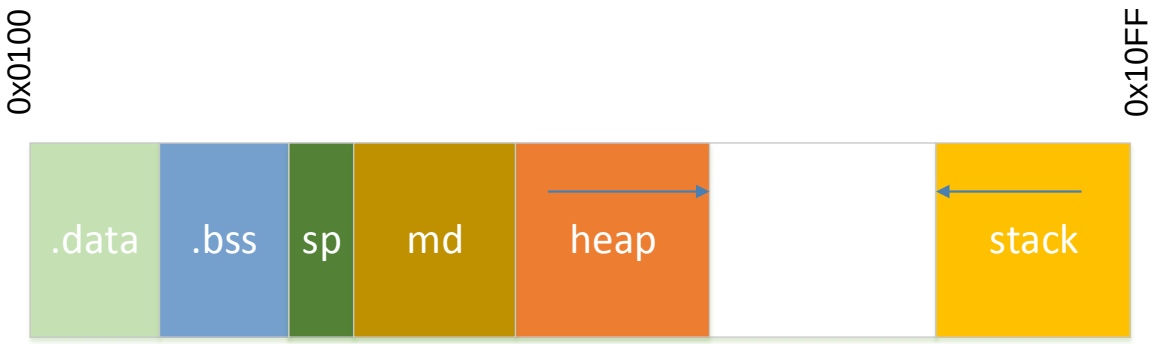


Figure 4.1: Modified Memory Sections

4.1 Supporting Memory Sections

To store the duplicate stack frames, two new sections are created in SRAM, as shown in Figure 4.1, just after the `.bss` section. This is achieved by modifying the linker script [50].

The `md` section is used to store duplicate stack frames. These duplicate frames are referred to as *Stack Frame Snapshots* (SFSs). The heap section grows towards the stack, and of course, the runtime usage of the heap and stack are unpredictable. To prevent the `md` and `heap` sections from colliding, the size of the `md` section is fixed. If the heap is used, the size of the `md` section is set to 1/3 of the available space; otherwise, it is set to 1/2 of the available space. For example, if the `.data` and `.bss` sections require 1 KB in a 4 KB RAM, the space available is 3 KB, so the size of `md` would be set to 1 KB, if the heap were used, and the size of `md` would be 1.5 KB, if the heap were not used.

The `sp` section is used to store the address of the next available memory space in `md`, similar to the stack pointer. This address is referred to as the *Snapshot Top Pointer* (STP). To protect the STP from SEUs, the size of the `sp` section is set to 6 bytes, and 3 STP duplicates are stored in this section. Given that we assume only one SEU will occur during the execution of a given function, only one STP duplicate could be altered by a flipped bit. The altered STP is easily identified and corrected by comparing the values of the three STP duplicates.

4.2 Injected Code Segments

We categorize the injected code based on function. Each contiguous assembly segment performs a set of operations, handling a specific action. These segments are designed to use only

registers, reducing their dependency on RAM. Each segment is assigned a unique ID. Here we summarize each type of code segment.

- *CRC Calculation* (ID: CC): This segment is used to calculate the CRC of a memory region. In our implementation, CRC16-CCITT is used [18].
- *CRC Save* (ID: CS): This segment is used to save the CRC to the stack.
- *CRC Compare* (ID: CM): This segment is used to compare two CRCs. The comparison result indicates whether an SEU is detected.
- *Frame Copy* (ID: FC): This segment is used to copy a stack frame to a given destination, and to save and restore stack frames.
- *Frame Size Save* (ID: FS): This segment is used to save the size of the stack frame for the current function. This will be discussed further in Section 4.3.3.
- *STP Initialization* (ID: SN): This segment is used to initialize the STP so it points to the lowest address of the `md` section.
- *STP Update* (ID: SU): This segment is used to update the STP. First, it obtains the correct STP value by comparing the three STP replicas. Next, the replicas are updated to reflect the addition or removal of a stack frame.

4.3 The ASM Handler

The ASM Handler, written in Java, automates the code injection process. First, the input C code is compiled to assembly using GCC. The ASM Handler then injects the assembly code. The modified code is then assembled and linked into an AVR executable. The Handler implementation consists of three loosely-coupled modules: the Reader, the Scanner, and the Injector, as shown in Figure 4.2. Assembly metadata is generated to assist the code injection process. Here, we describe the metadata creation process and describe each module of the ASM Handler.

4.3.1 ASM Metadata

Each line of assembly code is tagged with metadata that describes the line of code. Each metadata annotation classifies a line into one of three categories, as shown in Listing 4.1. A *directive*

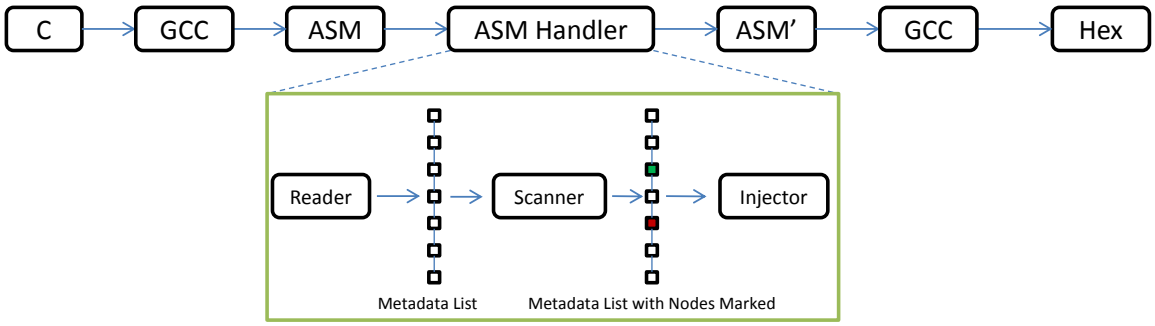


Figure 4.2: Code Injection Process

```
1 .arch atmega644          % directive
2 .text                    % directive
3 .global main             % directive
4 .type main, @function   % directive
5 main:                    % label
6 push r28                 % instruction
7 push r29                 % instruction
8 ...
```

Listing 4.1: Assembly Code Example

is used to specify assembly code information used by the assembler, including the system architecture (line 1), a section declaration (line 2), a label declaration (line 3), a label type (line 4), and other information. A *label reference* is used to identify a location in the assembly code (line 5). In this example, `main` specifies the starting address of the main function. An *instruction* is used to identify an instruction that will be executed by the microprocessor (lines 6-8). All metadata annotations include code injection information, which specifies the corresponding code segments to be injected (discussed in Section 4.2), as well as the position of the injection (i.e., before or after the assembly line).

4.3.2 Reader

The Reader is used to read the assembly file and generate corresponding metadata. It reads each line of assembly and generates a corresponding metadata node (in memory), which is then appended to a metadata list. For example, the Reader generates a list with 7 nodes after it reads the code in Listing 4.1, as shown in Figure 4.2.

4.3.3 Scanner

The Scanner is used to scan the metadata list and mark each metadata node based on the operations performed by the associated code. Marked metadata nodes indicate that new code segments will be injected either before or after the corresponding line of code. The scanner analyzes the assembly code to identify the operations where code segments must be injected. Here, we summarize the key operations.

- The *Stack Frame Establishment* operation is used to establish the stack frame for the current function. This operation is identified by scanning for "`sbiw r28,`n", which is used to establish the stack frame.
- The *Stack Frame Pointer Save* operation is used to copy the stack frame pointer, `Y`, to the stack pointer, `SP`. This operation is identified by scanning for "`out __SP_L__,r28`", which indicates that the required registers are ready, and the function is about to execute.
- The *Function Return* operation is used when a function returns. This operation is identified by scanning for the return instruction, "`ret`".

The Scanner scans each node in the metadata list, checks if the code represented by the node performs one of the key operations, and marks each such node with a list of identifiers from the set {CC, CS, CM, FC, FS, SN, SU, P}, where CC, CS, CM, FC, FS, SN, SU are the IDs of the code segments to be injected, and P indicates the position of the injection. Nodes that do not require code injection are not tagged. For example, the metadata node that represents the function return operation is marked with {CC, CM, FC, SU, P}, indicating that code segments CC, CM, FC, and SU must be injected before the associated line of code.

The Scanner also extracts two information elements from the metadata list: i) The Scanner scans the metadata list and detects if `malloc` is called in the target program, which indicates whether the `heap` section in RAM is used. This information is later used in determining the section size used to store the stack frame duplicates, as discussed in Section 4.1. ii) The Scanner determines the size of each function's stack frame by scanning the assembly code used to establish the stack frame, "`sbiw r28, n`", yielding a stack frame size of $n + 10$. The `n` bytes are used to store the arguments and local variables, and the additional 10 bytes are used to store the return address, CRC, and three copies of the stack frame size, each of which requires 2 bytes.

4.3.4 Injector

The Injector is used to inject code segments into the target assembly code. It again scans the metadata list. When a node is marked, the Injector injects the specified code segments at the position specified by parameter P. Finally, a modified assembly file is generated, which is then assembled and linked to form an executable.

In our initial approach, the code segments were directly injected into the target code, effectively making the code segments *inlined*. The results showed that the ROM overhead was significant. Each function, regardless of its size, was injected with code segments that require approximately 500 bytes of ROM. In our final approach, all the code segments are injected at the end of the target code, and each is labeled with its unique ID. When scanning the metadata list, instead of injecting the code segments into the target code, a function call instruction, `call`, is injected. A function return instruction, `ret`, is added at the end of each code segment. Because each segment was designed to use only registers, only two stack bytes are needed by each segment to save the return address. We discuss the performance of both approaches in Section 4.5.

4.4 Modified Function Execution Process

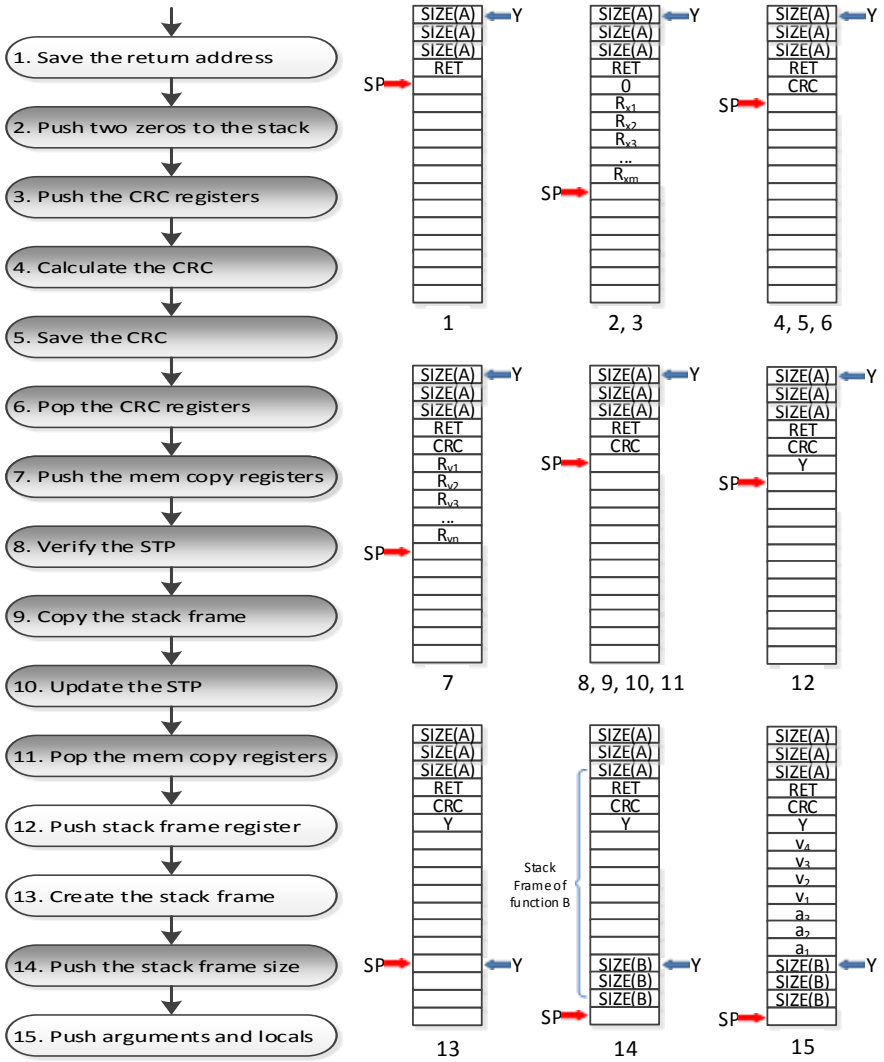
The auxiliary code injected at the beginning and end of each function modifies the function execution process, including the invocation and return sub-processes.

4.4.1 Modified Function Invocation Process

The code segments injected at the beginning of each function are used to calculate a CRC over the caller's stack frame, and to save a duplicate of the caller's stack frame, as shown in Figure 4.3. Figure 4.3a shows the execution process of the pre-invocation code; Figure 4.3b shows the stack changes associated with the pre-invocation code. Each rectangle represents two stack bytes. In the execution process diagram, the white ovals denote unchanged operations performed by the original code, and the shaded ovals denote operations modified or introduced by the injected code. In the stack diagram, `SP` denotes the stack pointer, and `Y` denotes the stack frame pointer. The numbers below each stack identify the operations that changed the stack.

When a function `B` is called by a function `A`, the return address of A is pushed onto the stack automatically by the function call instruction (step 1). To calculate the CRC of the caller's

stack frame, multiple registers are used, so they must be saved before the CRC calculation process and restored when the process is finished. To prevent the calculated CRC from being overwritten when the registers are restored, two bytes (zeros) are pushed onto the stack as a placeholder (step 2) for the CRC result before the registers used to calculate the CRC are saved (step 3). After the CRC of function A's stack frame is calculated (step 4), the CRC result is saved to the placeholder location (step 5). The registers used to calculate the CRC are then restored (step 6). Next, the



(a) Function Invocation Process (b) Stack Changes

Figure 4.3: Modified Function Invocation Process

stack frame of the caller, function A, has to be saved. The registers used in the stack saving process are pushed onto the stack (step 7). Next, the correct STP is selected by comparing the values of

the three STP copies (step 8). Using the correct STP, the stack frame is then copied and saved in the `md` section (step 9). After the STP copies are updated (step 10), the CRC registers are restored (step 11).

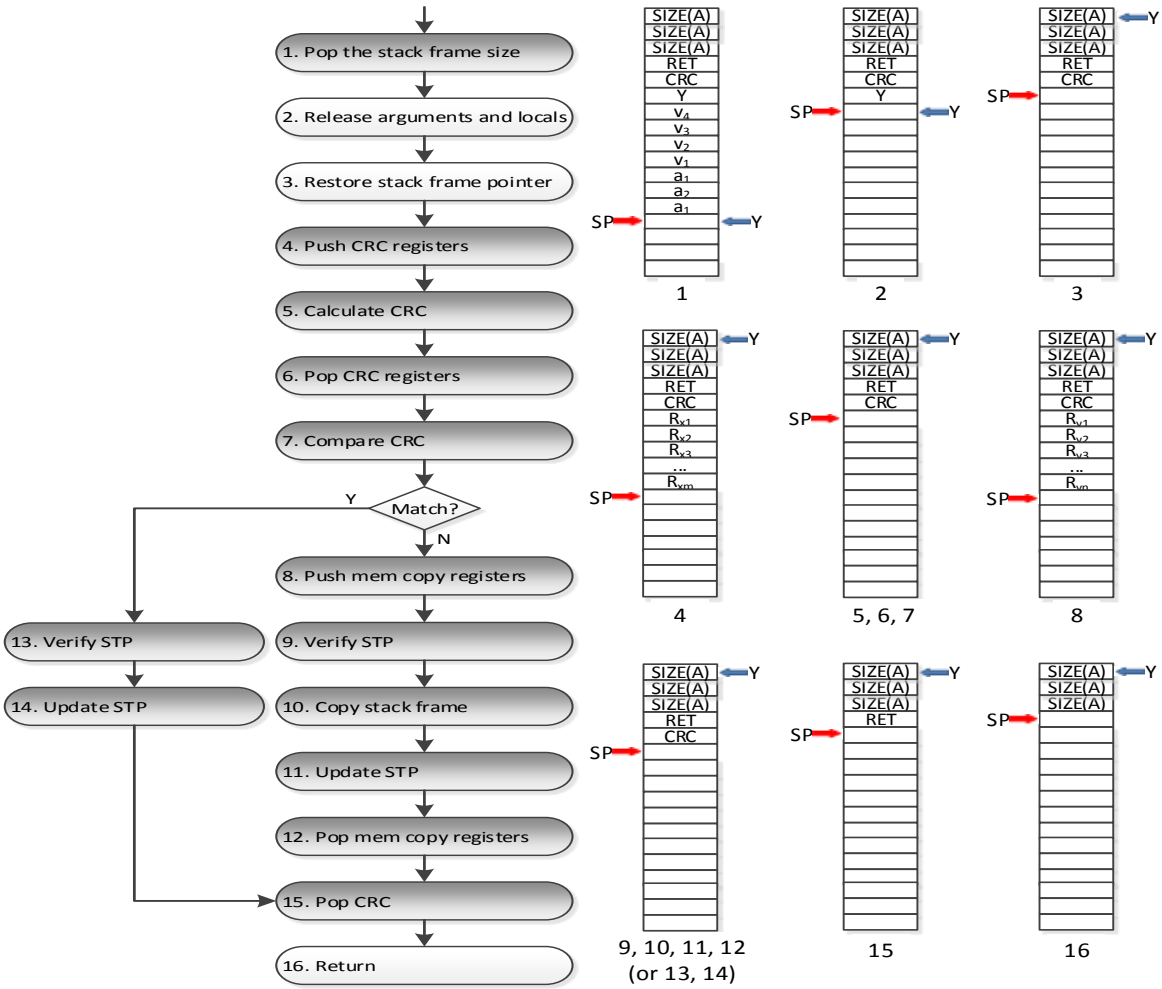
After the stack frame pointer of function `B` is saved (step 12), and the stack frame is established (step 13), three copies of the stack frame size of the callee, function `B`, are pushed onto the stack (step 14), which is a key operation in the injected code.

When a function is called, the return address is pushed onto the stack, and later used when the function returns. However, the callee does not have sufficient context regarding its caller, including the caller's stack frame address and size. It is impossible for the callee to calculate the CRC of the caller's stack frame and to duplicate the stack frame without this information. To solve this problem, each function saves its stack frame size in the stack, which is used by its callee to perform the CRC calculation and stack frame copy. To ensure the correctness of the stack frame size, three copies are saved. A three-way comparison is used to yield the correct value.

4.4.2 Modified Function Return Process

The code segments injected at the end of each function are used to verify the stack frame of the caller function, and to restore the stack frame if an SEU is detected, as shown in Figure 4.4. Each rectangle represents two stack bytes. Again, in the execution process diagram, the white ovals denote operations performed by the original code, and the shaded ovals denote operations modified or introduced by the injected code.

When function `B` returns, it first pops its stack frame size (step 1). After the space used to store the arguments and local variables is released (step 2), the stack frame pointer is restored (step 3). The CRC of function `A`'s stack frame is then calculated and temporarily stored in two registers (steps 4-6). The values stored in these registers are saved before the function return process. Next, the calculated CRC is compared with the CRC saved in the stack (step 7). If the two CRCs do not match, the saved stack frame of `A` is restored, and the STP is updated to release the space used to store the stack frame of `A` (steps 8-12). Again, the stack frame size of function `A` saved in the stack is used to support the CRC comparison and stack frame restoration (if needed). If the two CRCs match, the STP is updated (steps 13-14). After verification of `A`'s stack frame is complete, the CRC is popped from the stack (step 15). Finally, function `B` returns, and the return address is popped automatically (step 16).



(a) Function Return Process (b) Stack Changes

Figure 4.4: Modified Function Return Process

4.5 Evaluation

Here we present our evaluation of the SEU protection approach. We first introduce three test applications with different degrees of stack dynamism. We then validate the correctness of our approach and analyze the relationship between protection efficacy and the SEU injection rate. Next, we consider the overhead introduced by our approach, both in terms of space and execution speed. Finally, we validate our approach on physical hardware. Ubuntu 13.10, with Linux kernel version 3.8, and GCC 4.1.2 are used.

4.5.1 Test Applications

To evaluate our approach under varying stack conditions and SEU injection rates, three AVR applications are considered. The stack usage of each function is collected by repeatedly querying the value of the stack pointer at short intervals (i.e., every 25.6 μs). Given that the address of the "stack bottom" is known, the size of the stack is easily calculated. The stack usage pattern of each application is shown in Figure 4.5. The x-axis represents execution time, and the y-axis represents stack size. Below is a description of each application. The source code of the three applications is presented in Appendix A.

- The `Delay` application repeatedly executes a function that contains a delay of 2,040 clock cycles, implemented using a while loop, yielding low stack variability.
- The `Double Function Calls` application repeatedly executes three functions — function A calls B, and function B calls C — yielding moderate stack variability.
- The `Fibonacci` application repeatedly calculates the tenth Fibonacci number using recursion, yielding significant stack variability.

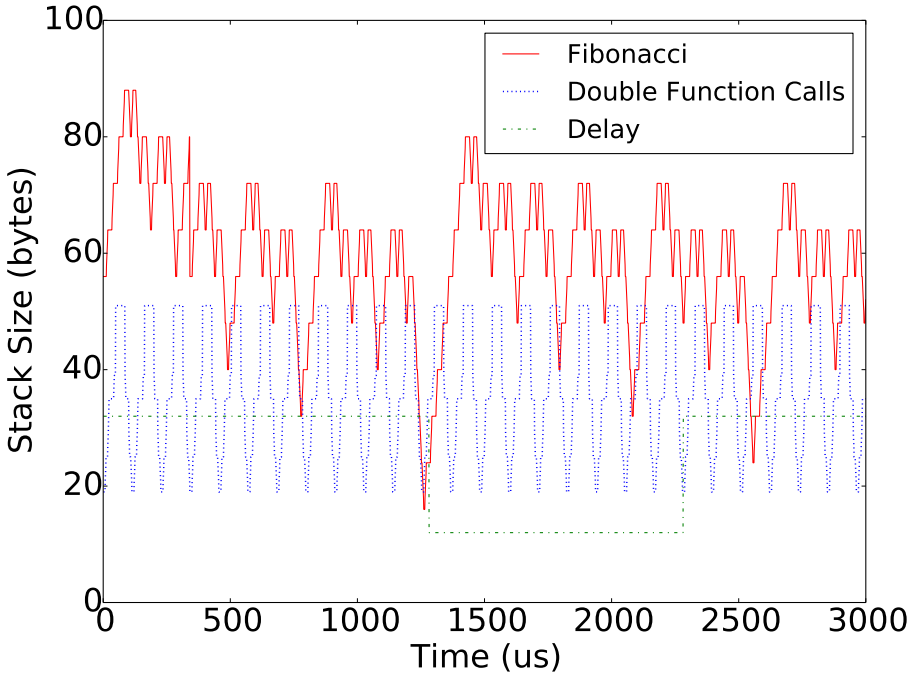


Figure 4.5: Stack Usage of Test Applications

4.5.2 Validation

We first validate our approach and consider the SEU protection efficacy it affords. Recall the modified SRAM partition shown in Figure 4.1. In our analysis, we ignore both the .data and the .bss sections, as well as the heap section. Global variables stored in the .data and the .bss sections can be protected using well-known techniques based on cloning and comparison. Similarly, heap space is often unused in embedded applications due to limited memory availability. We focus our analysis on stack frame protection. Again, the injected code segments used to protect the stack frames are designed to use only registers, and each segment requires only two bytes in the currently executing function's stack frame (for the return address).

We first assume that the currently executing function's frame, which includes the return address of the injected code segment, is not affected by SEUs. We use induction to prove the correctness of our approach. Suppose n is the number of stack frames stored in the stack, excluding the frame for *main*.

Base Case: If $n = 0$, only the stack frame of *main* is on the stack. When *main* calls another function, say *foo*, the stack frame for *foo* is created. According to our assumption, the current stack frame (*foo*'s) will not be affected by SEUs during execution. When *foo* returns, the stack frame of *main* is protected by our approach. So the stack frames of caller and callee are guaranteed to be correct if any function is invoked and returns, when $n = 0$.

Inductive Step: Assume that the stack frames of callers and callees are guaranteed to be correct for $n = k$, where $k \geq 1$. Now consider $n = k + 1$. Assume a is the current function, which calls b. According to our assumption, the stack frame of the executing function, b, is not affected by SEUs. When b returns, the stack frame of a is protected by our approach. So the stack frames of callers and callees are guaranteed to be correct when $n = k + 1$. Given the assumption that the stack frames of callers and callees are guaranteed to be correct for $n = k$, by induction, the stack is guaranteed to be correct, assuming the executing stack frame is never affected by SEUs.

To verify this claim, the AVR Simulator IDE [43] is used to manually inject SEUs, and to observe execution results using two breakpoints added to the callee. The first breakpoint is added at the beginning of the callee body and is used to inject an SEU error. When the program pauses at this breakpoint, we manually flip a bit in the caller's stack frame, simulating an SEU error. The second breakpoint is added right before the callee returns and is used to observe the protection

results. When the program pauses at this breakpoint, we check the caller's stack frame. The results show that each function is able to detect and correct SEUs introduced "beneath" the topmost stack frame.

Note that if the stack frame of the executing function is affected by an SEU, protection is not guaranteed. If the SEU changes key data, such as the return address or stack frame size, the current function will not execute as expected. We assume that only one SEU will occur during a given function execution, and that the SEU is uniformly likely to affect all bits in RAM. The probability of successful SEU protection can be expressed as:

$$p = 1 - \frac{c}{2s + e - c + 6} \tag{4.1}$$

Where p is the probability of successful protection, s is the stack size, e is the size of the unused space in RAM, 6 is the size of the three STP copies, and c is the average size of a stack frame. Since the return address of the injected code segment is stored in the current stack frame, the two bytes for the return address are included in c. The total size of protected memory is $s + e + (s - c) + 6$, where $s - c$ is the size of the stack frame copies stored in the *md* section.

We extend our analysis to cases where more than one SEU may occur during a given function execution. Our approach succeeds when the following conditions are met: (i) the currently executing function's stack frame is not affected (so the return address of the injected code segment is not affected); (ii) at least two of the three copies of the caller's stack frame size are not affected; (iii) at least two of the three copies of the STPs are not affected; and (iv) at least one of the two caller's stack frames (the original and the backup copy saved in the `md` section) is not affected. To simplify the analysis, conditions (ii) and (iii) are strengthened, requiring that all three copies of the caller's stack frame size not be affected, and that all three copies of the STPs not be affected. Since the strengthened conditions slightly reduce the probability of successful SEU protection (only 4 bytes are ignored), the real probability of protection is slightly higher than the presented results.

The probability of successful SEU protection can be expressed as:

$$
\begin{aligned}
p = & (1 - \frac{c}{2s+e-c+6})^n * (1 - \frac{6}{2s+e-2c+6})^n \\
& * (1 - \frac{6}{2s+e-2c})^n * \{(1 - \frac{2c}{2s+e-2c-6})^n \\
& + \mathrm{C}_2^1 * (1 - \frac{c}{2s+e-2c-6})^n \\
& * [1 - (1 - \frac{c}{2s+e-3c-6})^n]\}
\end{aligned}
\tag{4.2}
$$

Where p is the probability of success, s is the size of the stack, e is the size of the unused space in RAM, 6 is the size of the three stack frame size copies (or the three STP copies), c is the average size of the stack frame (including the return address of the injected code segment), and n is the number of SEUs that occur during a function's execution. In equation 4.2, $(1 - \frac{c}{2s+e-c+6})^n$ is the probability that the currently executing function's stack frame is not affected by SEUs. $(1 - \frac{6}{2s+e-2c+6})^n * (1 - \frac{6}{2s+e-2c})^n$ is the probability that both copies of the caller's stack frame size and the three STPs are not affected by SEUs. Within the curly brackets, $(1 - \frac{2c}{2s+e-2c-6})^n$ is the probability that both the original and the copy of the caller's stack frame are not affected by SEUs. $\mathrm{C}_2^1 * (1 - \frac{c}{2s+e-2c-6})^n * [1 - (1 - \frac{c}{2s+e-3c-6})^n]$ is the probability that either the original or the copy of the caller's stack frame is affected by SEUs. So $\{(1 - \frac{2c}{2s+e-2c-6})^n + \mathrm{C}_2^1 * (1 - \frac{c}{2s+e-2c-6})^n * [1 - (1 - \frac{c}{2s+e-3c-6})^n]\}$ is the probability that at least one — the original or the copy — of the caller's stack frames is not affected.

In equation 4.2, the average number of SEUs that occur per function execution, n, can be approximated as:

$$
n = \frac{y * l}{m} * f \tag{4.3}
$$

where y is the average number of clock cycles used to execute each instruction, m is the frequency of the microprocessor, l is the average number of instructions within each function, and f is the SEU injection rate. Most AVR instructions require 2 clock cycles to execute, and the frequency of our ATmega644 is set to 10MHz.

We now consider the relationship between SEU protection probability and SEU occurrence rate. To demonstrate the relationship, we collect the corresponding parameters for the three test applications using AVR Simulator IDE, as shown in Table 4.1. Figure 4.6 plots the change in SEU protection probability as a function of SEU injection rate. The x-axis represents the rate at which

Applications	l	c (bytes)	s (bytes)	e (bytes)
Delay	115	16	30	3022
Double Function Calls	54	9	30	3022
Fibonacci	42	10	60	2992

Table 4.1: Application Stack Characteristics

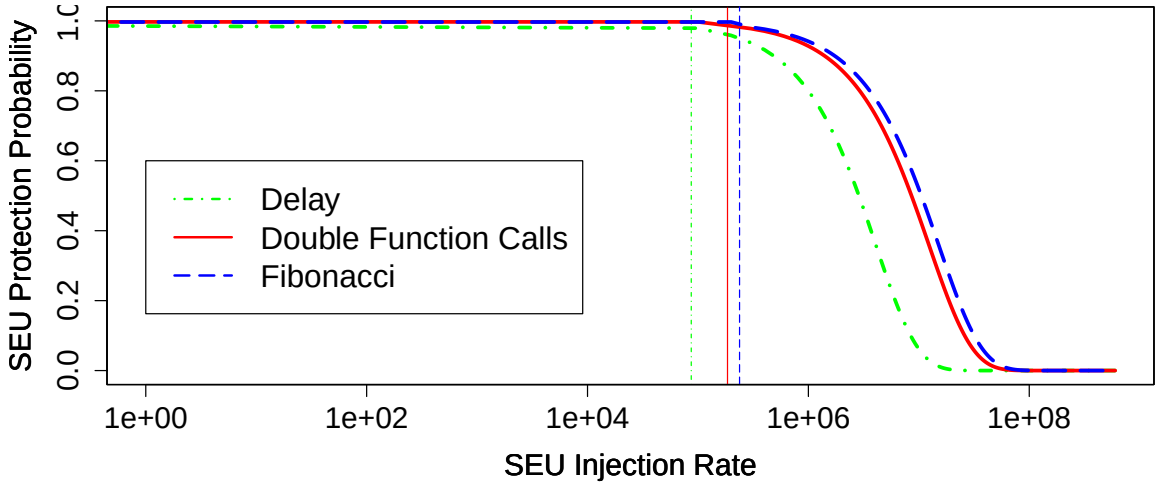


Figure 4.6: SEU Protection Probability

SEUs are injected, and the y-axis represents the corresponding SEU protection probability. Each vertical line marks where the number of SEUs begins to exceed 1 (for each application). When only one SEU occurs during a given function execution (left side of the vertical line), the SEU protection probability is constant (Delay: 99.48%, Double Function Calls: 99.71%, Fibonacci: 99.68%) because the only case the approach cannot handle is when the current frame is affected. When more than one SEU occurs during a given function execution (right side of the vertical line), the SEU protection probability decreases because the SEUs may affect the stack frame of the current function, the stack frame sizes of the caller stored in the stack, the STPs, and stack frame copies stored in the `md` section. As the SEU occurrence rate increases, the SEU protection probability decreases, until it approaches 0. The lower the stack dynamism, the longer the function execution time, which increases the probability of SEU occurrence in the current stack frame. Low stack frame dynamism causes the SEU protection probability for Delay to drop significantly compared to the other applications.

Code Segment	**Number of Executions**	**Instructions**	**Clock Cycles**	**ROM Space**
CRC Calculation	2	24*S+1	27*S+4	50
CRC Save	1	13	26	26
CRC Compare	1	27	52	64
Frame Copy	1 or 2	64+4*S	8*S+128	50
Frame Size save	1	18	34	36
STP Initialization	1	16	28	32
STP Update	2	7	14	14
Total (No SEUs)	-	48*S+154	62*S+304	272
Total (Recovery)	-	52*S+218	70*S+432	272

Table 4.2: Execution Overhead

4.5.3 Performance

Since the same code is injected for every function, the execution overhead is similar for all functions, varying only when an SEU is detected. Table 4.2 summarizes the overhead of each injected code segment. The second column lists the number of times each code segment executes (per function execution), the third column lists the number of instructions executed in each code segment, the fourth column lists the number of clock cycles spent executing each code segment, and the fifth column lists the ROM space overhead for each injected segment. S denotes the size of the (recovered) stack frame. The *CRC calculation* code segment and *STP update* code segment execute twice for each function, and the *frame copy* code segment executes either once or twice, depending on whether an SEU is detected. Each of the other code segments executes once for each function execution. Therefore, the minimum overhead introduced in terms of number of clock cycles is $62 * S + 304$, when an SEU is not detected. The worst case is $70 * S + 432$ clock cycles, when an SEU is detected.

We next evaluate space overhead using the three test applications. The ROM space data was collected using *avr-size*. The results are summarized in Figure 4.7. The y-axis represents ROM size, in bytes. Delay and Fibonacci involve two functions, and Double Function Calls involves four. From Figure 4.7, we can see that the ROM overhead for the Double Function Calls application is twice that of the Delay and Fibonacci applications. ROM overhead is related only to the number of functions in the program.

We next evaluate execution overhead. As shown in Table 4.2, the execution overhead for every function call is determined primarily by the size of the stack frame. We consider the execution

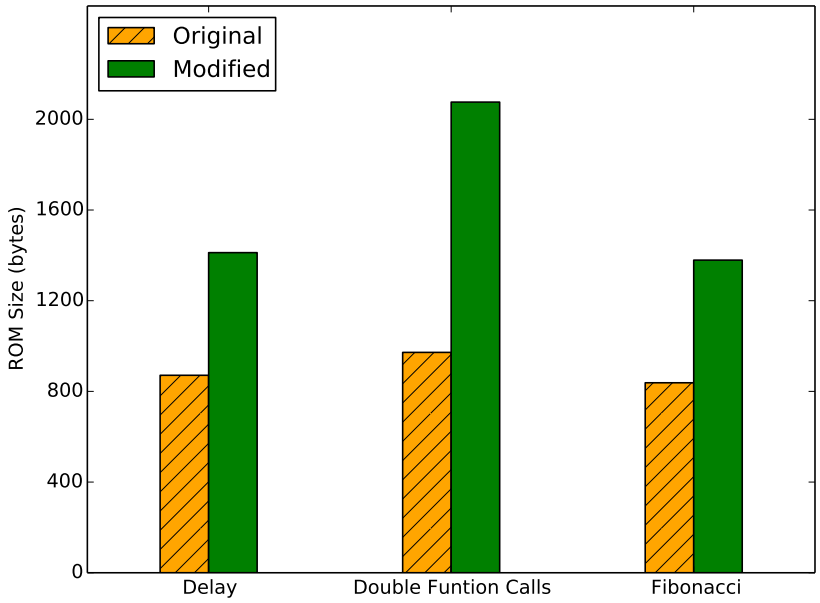


Figure 4.7: ROM Overhead

overhead as a function of the average number of instructions executed between each call instruction (i.e., an inverse measure of call frequency). The execution overhead can be expressed as:

$$e = \frac{l + L}{l} \tag{4.4}$$

Where L is the number of injected machine instructions for each function, and l is the average number of machine instructions executed between two function call instructions. As summarized in Table 4.2, $L = 48 * S + 154$ when no SEUs are detected, and $L = 52 * S + 218$ when an SEU is detected. The average frame size, S, is 20.

The execution overhead is summarized in Figure 4.8. The x-axis represents the average number of instructions executed between function invocations (l), and the y-axis represents execution overhead, measured as the ratio between the execution speed of the original code and the modified code. The figure shows that given the same stack frame size, execution overhead is determined by stack dynamism. The less stack dynamism, the less speed overhead. The explanation is that within a given period of time, increased function calls lead to increased execution of the injected code.

The results reveal an interesting tradeoff among dynamism, protection efficacy, and performance. Increasing dynamism offers better protection, but worse performance; decreasing dynamism offers better performance, but less protection. Knowledge of this tradeoff can be used to inform the

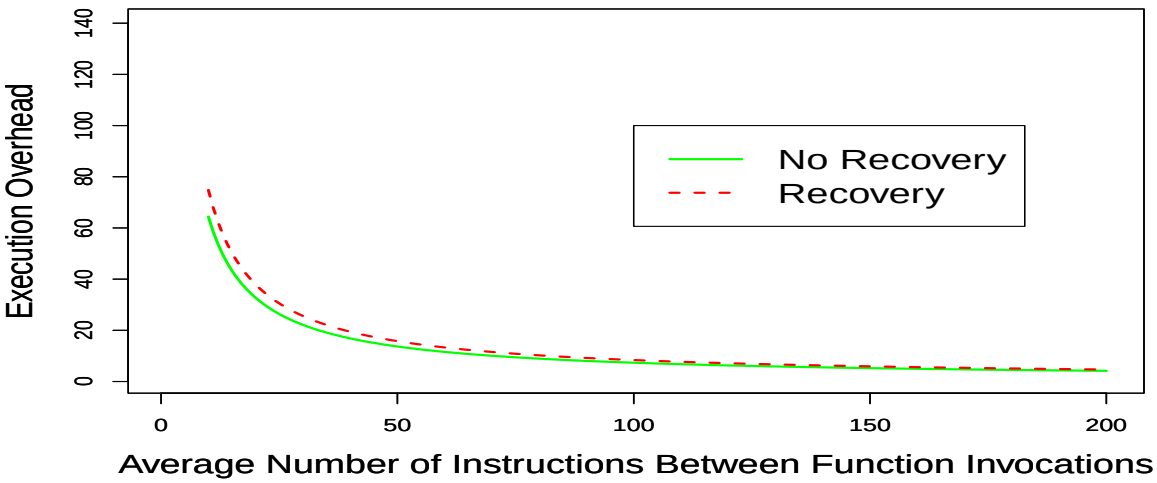


Figure 4.8: Execution Overhead

function decomposition process, enabling embedded designers to appropriately balance protection efficacy and execution overhead.

4.5.4 Physical Hardware

To validate our approach on physical hardware, we emulate the occurrence of SEUs by flipping random bits in the SRAM area. To perform auditable test runs, we developed an AVR application that continuously generates an increasing integer sequence, which is then sent to the UART interface at a speed controlled by a delay function. In our experiments, the delay times are set to 1000 ms, 100 ms, 10 ms, and 1 ms, making the integer generation speed 1/s, 10/s, 100/s, and 1000/s, respectively. A Python program running on a desktop is used to receive the sequence and observe the impact of flipped bits by monitoring the continuity of the sequence. A timer interrupt is used to trigger the occurrence of SEUs. The interrupt service routine generates a random address within the range of the top of the stack and the end of RAM space, excluding the stack frame of the current interrupt, and then flips the bit at this location. The SEU injection frequencies are set to 39062Hz, 4878Hz, 609Hz, 153Hz, and 38Hz[1]. An Atmega644 microprocessor is used in our experiments.

We declare (observable) failure when one of the following two situations occurs: (i) The AVR application stops generating integers; or (ii) the integer sequence received by the Python program

[1]These frequencies are derived from the built-in timer prescaler of the Atmega644 at values of 1, 8, 64, 256, and 1024, respectively

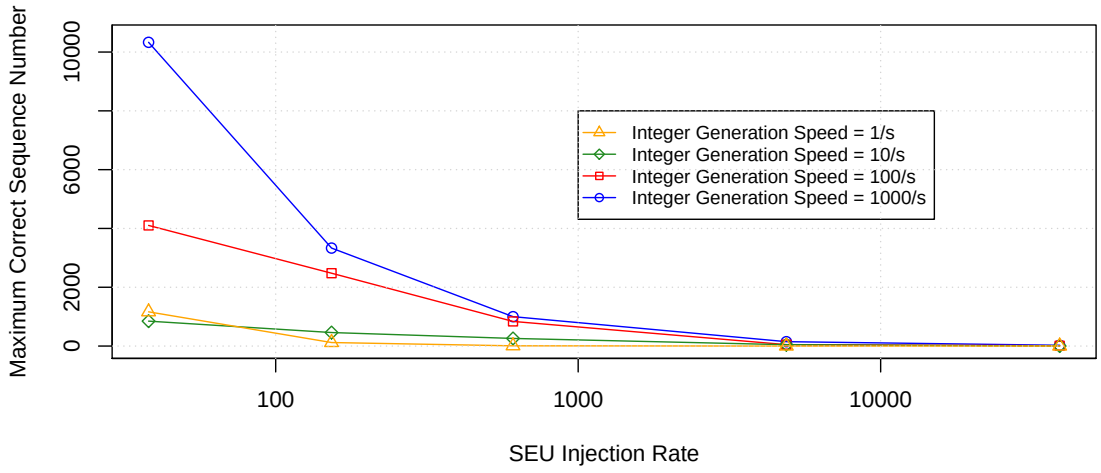


Figure 4.9: Physical Hardware Results

becomes discontinuous. We monitor the integer sequence and record the maximum count before failure. The experimental results are summarized in Figure 4.9. The x-axis represents SEU injection frequency, and the y-axis represents the maximum count received by the Python program. The figure shows that as SEU injection frequency increases, running time to failure decreases. This matches our SEU protection probability analysis, summarized in Figure 4.6, and is again explained as follows: As SEU injection frequency increases, the probability that an SEU occurs in a critical area increases. When the frequency is extremely high (e.g. approximately 10 MHz), the program can hardly send any values. However, the observed SEU occurrence rate in outer space is approximately $10^{-6} SEU/bit\text{-}Day$ [55]. Given that the total RAM size of an Atmega644 is 4K Bytes, the expected SEU occurrence rate for an Atmega644 is 0.0032 SEU/day, which is significantly lower than the lowest frequency (9765.625 SEU/second) that we considered. This situation would be extremely rare in real scenarios.

4.6 Summary

In this chapter, we presented our approach to protecting embedded system memory from SEUs. We first described the code segments used to perform stack duplication, validation, and recovery. We then described a tool used to appropriately inject the code segments into the target

program. We next presented the modified function invocation and return processes. Finally, we presented the evaluation of our approach, including a protection probability analysis, a performance analysis, and a physical hardware test. Results show that our approach achieves a stack protection success rate of over 99% for typical programs.

Chapter 5

Serial Device Driver Generation

In this chapter, we present the design, implementation, and evaluation of an automated serial device driver generation system for embedded systems.

5.1 Serial Device Driver Generation

Developing serial device drivers involves significant repetitive work since most serial devices are operated in the same manner. A microprocessor operates a serial device by issuing device-specific commands. Each command specifies an action expected to be executed by the target device. For each command, a corresponding response will be sent to the microprocessor after execution, indicating the result of the command and the status of the device. Moreover, energy efficiency is a key requirement of many embedded systems, making the timeout parameters associated with each command important factors affecting the amount of time a microprocessor spends in a low power state. Unfortunately, accurately measuring the time associated with sending commands and receiving responses can be time-consuming and error prone. Based on this observation of similarity across drivers and the attendant difficulty of measuring the associated timeout parameters, we present a configuration-based tool that simulates the execution of a specified command set, accurately measures the timeout parameters, and generates the target device driver based on the execution results. Each command sequence is implemented as a function in the driver.

The working process of DriverGen is simple. DriverGen reads, parses, and validates a driver configuration file. Next, the tool issues the specified commands to the target device and receives the

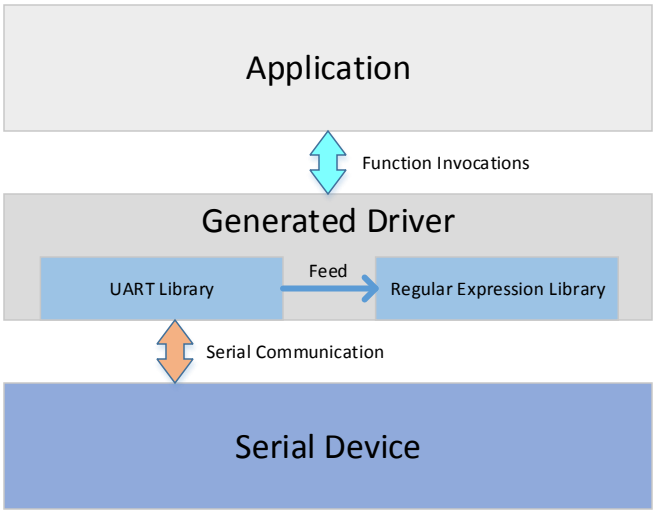


Figure 5.1: Application and Driver Architecture

corresponding responses. Finally, a header file and a body file are generated based on the execution results. To explain our design, we step through the driver generation process for the GM862, a popular cellular modem chip.

5.1.1 UART Library

To handle UART communication, a UART library is included in the generated driver. Key elements of the library header are shown in Figure 5.1. The generated driver uses the UART library to communicate with the serial device. It implements the functions shown in Listing 5.1.

```
1 void uart_init(uart_config_t* config);
2 void uart_wake();
3 void uart_sleep();
4 void uart_send_bytes(uint8_t* data, uint8_t length);
5 bool uart_receive_byte(uint8_t* data, uint16_t timeout_ms);
```

Listing 5.1: UART Library (partial)

- *uart_init*: This function is used to initialize the UART of the microprocessor. The struct argument contains UART configuration parameters, such as baud rate, number of data bits and stop bits, etc.
- *uart_wake*: This function is used to wake the UART from power reduction mode.

- *uart_sleep*: This function is used to put the UART into power reduction mode.
- *uart_send_bytes*: This function is used to send the bytes passed as argument to the target device. The number of bytes is specified in the second parameter, *length*.
- *uart_receive_byte*: This function is used to receive a byte from the target device. If the expected byte is not received within the time specified by the second parameter, *timeout_ms*, the function returns *false*, indicating a timeout error.

5.1.2 Regular Expression Library

Different commands yield different responses, indicating whether the commands were executed successfully. The information in some responses may need to be saved. For example, this is a typical response to the *time query* command when issued to the GM862:

TX: AT+CCLK
RX: +CCLK: "13/05/19,14:15:32"
OK

The string "OK" indicates a successful query; "13/05/19" and "14:15:32" represent the retrieved date and time, respectively. To match the response pattern and save the desired information, we implement regular expression libraries in Java and C, used by DriverGen and the generated drivers, respectively. Both work in the same manner. When the generated driver is initialized, the expected response pattern for each command specified in the configuration is compiled by the regular expression library and saved in RAM. Each compiled expression is assigned an ID for later invocation. This one-time initialization approach requires more space, but avoids the time required to re-compile each expression whenever a command is sent. Each response string is passed to the regular expression library character by character to determine the status of the device (e.g., TCP connection established), and to capture the desired information (e.g., date and time in the GM862 time query response). After a character is passed, the library returns *true* if the target string matches the corresponding regular expression, and *false* if not.

In this example, the regular expression "+CCLK:"(\d2/ \d2/ \d2),(\d2:\d2:\d2)".*\s*OK" is used to match "OK" and save the date and time [20]. "\d2" is used to match a 2-digit integer, and ".*\s*" is used to match any character, including whitespace (i.e., spaces, tabs, and newlines).

The parentheses are used to mark the information to be saved. In this example, the data and time will be saved separately.

Key elements of the header for the regular expression library are shown in Figure 5.1. The library is included in the generated driver to support regular expression matching. The received response strings are processed by the regular expression library byte by byte. It provides the functions shown in Listing 5.2.

```
1 bool regex_compile(char* reg);
2 bool regex_use(uint8_t regex_id);
3 bool regex_feed(char c);
4 void regex_get_saved_string(char* saved_string, uint8_t regex_id);
```

Listing 5.2: Regular Expression Library (partial)

- *regex_compile*: This function is used to compile a given regular expression. The function returns *true* if the passed regular expression is valid, and *false* otherwise. A unique ID is created for each generated regular expression for later invocation.
- *regex_use*: This function is used to set the regular expression used to match a response string. The regular expression is specified by its ID, passed as argument. The function returns *true* if the specified regular expression exists, and *false* if not.
- *regex_feed*: This function is used to pass a character to the regular expression library. It returns *true* if the (current selected) regular expression matches after the character is passed, and *false* if otherwise.
- *regex_get_saved_string*: This function is used to retrieve a string saved in a array created by the library. The first parameter is used to store the retrieved string, and the second parameter is used to specify the position of the saved string in the array. In our example, to retrieve the date and time strings, two calls to this function are needed; the values of the second argument will be 0 and 1, respectively.

5.1.3 Timing

To determine if the target device is responding, or the response is finished, timeouts are used, making accurate timing an important factor in a driver implementation. DriverGen monitors the

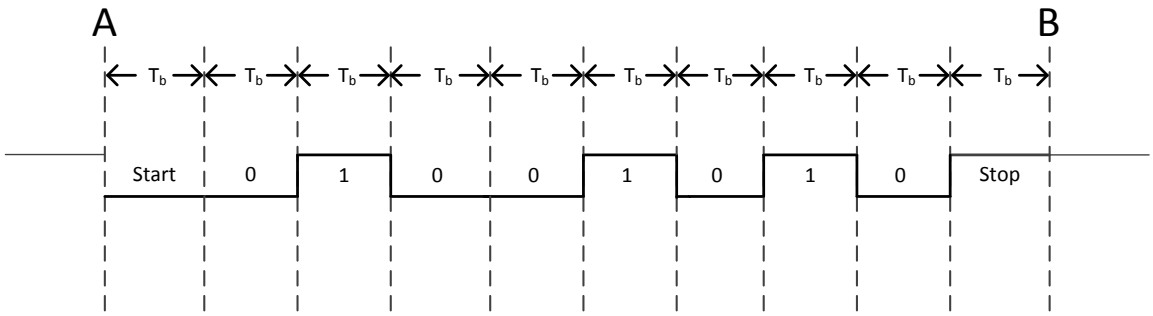


Figure 5.2: UART Transmission Timing

UART communication signals to measure the response time of the target device to each command, as well as the time between bytes in the response. We discuss our approach in the following sections.

5.1.3.1 UART Communication Timing

In standard UART communication, the data line is held high when idle. A data frame begins with a start bit (low), which signals the beginning of the data frame. One or two stop bits (high) are used to signal the end of a data frame. Figure 5.2 shows the data frame of character "J" (point `A` to `B`) using a common UART configuration: 9600 baud, 8 data bits, 1 stop bit, and no parity. `Start` and `Stop` represent the start and stop bits, and numbers `01000101` represent the data bits of character "J". Only the detection of the start bit is needed to detect a data frame. In our example, T_b, the time to transmit one bit, is approximately 0.1 ms. After a start bit is detected at point `A`, the detection process is disabled for $9.5 \times T_b$ to ignore the data bits, since the start bit is the only bit needed for detection. The half bit offset is introduced to ensure that the detection process resumes in the middle of the stop bit to avoid detection errors caused by clock inaccuracies.

5.1.3.2 Response Timing

Figure 5.3 shows the timing properties associated with sending commands and receiving responses. The horizontal axis represents time. The transparent and dark boxes represent command and response bytes, respectively. T_r is the `response time`, which represents the time between the last byte in a command and the first byte of a response. T_{i1}-T_{i4} are the `inter-byte times`, which represent the time between two sequential bytes in a response. For each command with a corresponding response, we measure T_r and T_i^*, the maximum inter-byte time — T_{i2} in this example.

T_r is used in the generated driver to detect if the target device responds to a given command.

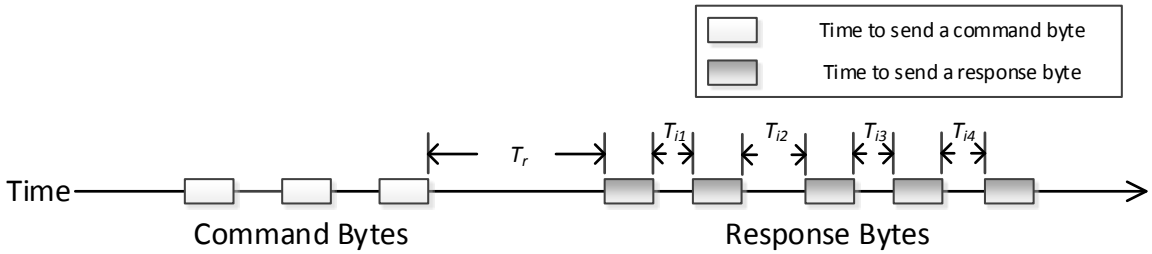


Figure 5.3: Command and Response Timing

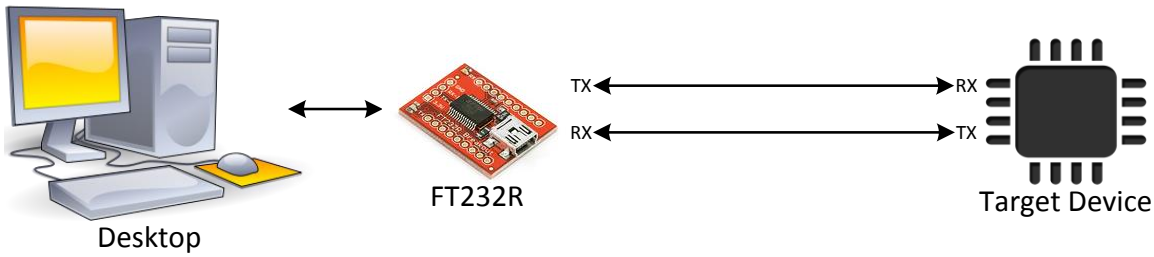
After sending the command to the target device, the driver waits for the first byte in the response, up to $(1+\alpha)\times T_r$. α is a small positive number (0.1 is used in our implementation) to accommodate (unseen) variation in the responses. If the first byte is not received within $(1+\alpha)\times T_r$, the target device is assumed to have ignored the command.

T_i^* is used to detect the end of the response. After receiving the first byte, the driver waits for each subsequent byte for $(1+\beta)\times T_i^*$. Again, β is a small positive number (0.1 is used in our implementation) to accommodate (unseen) variation in the response pattern. If the next byte is not received within $(1+\beta)\times T_i^*$, the response is considered finished.

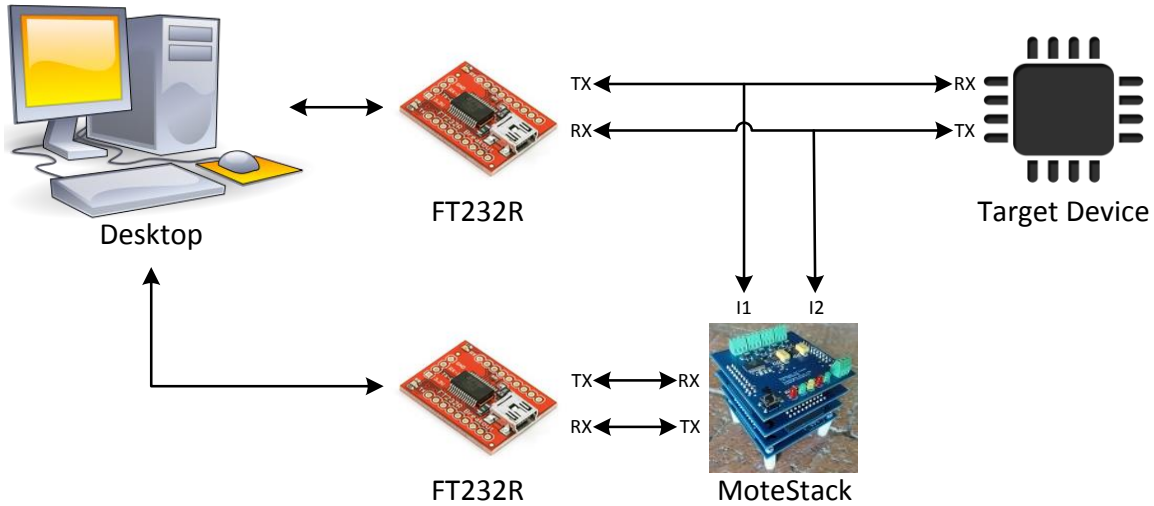
5.1.4 Hardware Setup

In the first version of the hardware design, shown in Figure 5.4a, the DriverGen hardware consisted of a desktop running a Java program, and an FT232R chip used by the desktop to communicate with the target device through the UART. However, because Java is an interpreted language running on the Java Virtual Machine (JVM), timing calls (e.g., `schedule` in class `Timer`) are not accurate or consistent. Further, Java programs do not have direct serial port access, so a third party library was used through the Java Native Interface (JNI) [52] to read and write data over the UART, making the timing even less accurate. For example, in our experiments using the GM862 cellular chip, the measured response time of a simple command that queries the system firmware version varied from about 12 ms to 35 ms.

In our second version of the design, shown in Figure 5.4b, an FT232R and a MoteStack [11] were added to the DriverGen hardware. The FT232R is used by the desktop to communicate with the MoteStack. The MoteStack is used to monitor the UART data signals to measure the response and inter-byte times. The ATMega644 microprocessor in the MoteStack runs at 10MHz and uses an 8-bit timer to keep track of time, yielding interrupt resolution of 25.6 μs. Two external interrupt



(a) First Approach



(b) Second Approach

Figure 5.4: Hardware Setup

pins, `I1` and `I2`, are connected to the UART buses and are configured to trigger on the falling edge (voltage going from high to low) to detect the start bits of UART data frames. By measuring time directly from the UART signals, the accuracy and consistency of measured response and inter-byte times are greatly improved. For example, the measured response time of the version query command falls to approximately 8-9 ms.

5.1.5 Driver Configuration

DriverGen runs based on a driver configuration file. The file is used to configure UART communication, control execution of each command, and generate the target driver. We use XML as the driver configuration format. A popular Java XML library, dom4j [35], is used in our implementation. We provide a portion of the GM862 configuration file in Listing 5.3.

5.1.5.1 Driver Configuration Parameters

The driver configuration parameters are organized into four categories:

```
1  <Configuration>
2     <Target>GM862</Target>
3     <Platform>AVR</Platform>
4     <MCU>ATMega168</MCU>
5     <Interface>UART</Interface>
6     <Version>1.0.0</Version>
7     <Global>
8        <ResponseTimeout>3000</ResponseTimeout>
9        <InterByteTimeout>10</InterByteTimeout>
10       <Retry>5</Retry>
11    </Global>
12    <Uart>
13       <BaudRate>9600</BaudRate>
14       <Data>8</Data>
15       <Stop>1</Stop>
16       <Parity>None</Parity>
17    </Uart>
18    <Function>
19       <Name>wake</Name>
20       <EntryPoint/>
21       <Command>AT&#13;</Command>
22       <ResponseMatch>\s*OK</ResponseMatch>
23       <Delay>1000</Delay>
24    </Function>
25    <Function>
26       <Name>set_new_interface</Name>
27       <Command>AT#SELINT=2&#13;</Command>
28       <ResponseMatch>\s*OK</ResponseMatch>
29    </Function>
30    <Function>
31       <Name>query_time</Name>
32       <Command>AT+CCLK&#13;</Command>
33       <ResponseMatch>
34          +CCLK:"(\d2/\d2\/d2),(\d2:\d2:\d2)".*\s*OK
35       </ResponseMatch>
36       <Dependency>
37          <DName>set_new_interface</DName>
38       </Dependency>
39     </Function>
40 </Configuration>
```

Listing 5.3: GM862 Configuration File (partial)

Basic Information Parameters (Lines 2-6). These parameters are used to specify the driver (i) name; (ii) platform, e.g., AVR, ARM[1]; (iii) microprocessor, e.g., ATMega168, ATMega644; (iv)

[1] AVR is supported in the current version.

communication interface, i.e., UART, I2C, SPI[2]; and (v) version. In this example, the GM862 driver is being developed for the `ATMega168` microprocessor of the `AVR` family. It uses `UART` as the communication interface.

Global Definition Parameters (Lines 7-11). These parameters are used to specify global constants used by DriverGen. The `ResponseTimeout` tag specifies the maximum time before the first response byte should be received, measured in milliseconds. The `InterByteTimeout` tag specifies the maximum time between subsequent bytes. The `Retry` tag specifies how many times DriverGen will retry the execution of each function before it stops and reports failure. A default retry value of 5 is used if another value is not specified.

UART Configuration Parameters (Lines 12-17). These parameters, only needed when the communication interface is configured to UART, include standard UART configuration parameters, including baud rate, number of data and stop bits, etc. In this example, the UART is configured to use 9600 baud, 8 data bits, 1 stop bit, and no parity.

Function Configuration Parameters (Lines 18-39). The function configuration parameters are the core of the driver configuration; they specify the names of the driver functions to be generated, the commands to be sent to the target device, the responses expected, and the delays to be introduced after function execution. These delays are often needed because some operations, e.g., system initialization, need extra time to finish after the responses are received. Note that the hexadecimal numbers in the `command` tags are used to specify characters that are not allowed by the XML standard.

The order of function execution is important because of inherent dependency relationships among commands. For example, the TCP transmission function should execute only after a connection is established by the TCP connection function. To specify the required order of function execution, we introduce `EntryPoint`, `Dependency`, and `DName` tags. Multiple `DName` tags are used when a function has more than one dependency. A function is the first to run if it is configured as the entry point, specifying that this function is a dependency for all other functions. The function that wakes the target device from sleep is usually the entry point of the driver. If function A is

[2]UART is supported in the current version

specified as a dependency of function B, function B will run after function A finishes execution, assuming it is successful.

In this example, lines 18-24 configure the `wake` function. For this function, "`AT<CR>`" is sent to the target device, and "OK" is expected. The response and inter-byte timeouts are set to 3000 and 10 milliseconds (in lines 8 and 9), respectively. This function is also configured to be the entry point of the driver. The `delay` tag in line 23 indicates a 1000-millisecond delay to be introduced after execution. The `set_new_interface` function (lines 25-29) executes after the entry point function `wake` because it does not have `dependency` tags, whereas function `query_time` (lines30-39) must execute after `set_new_interface` because `set_new_interface` is in its dependency list, as shown on lines 36-38. The configuration of functions `set_new_interface` and `query_time` is similar to the configuration of function `wake`.

5.1.5.2 Driver Configuration Parameter Rules

Configuration parameter rules are applied by the parser, discussed in Section 5.1.6.1, to validate the driver configuration. The rules fall into four categories:

Base Information Parameter Rules. These rules specify the validity constraints for the base information parameters. For example, if the AVR platform is specified, the microprocessor parameter must specify a member of the AVR ATMega series, such as the ATMega168, ATMega644, etc.

Global Definition Parameter Rules. These rules specify the validity constraints for the global definition parameters. For example, the response timeout, inter-byte timeout, and retry parameters must be positive integers.

UART Configuration Parameter Rules. If the UART communication interface is selected, these rules specify the validity constraints for the UART configuration parameters. For example, the number of data bits must be either 7 or 8, and the baud rate must be a standard rate, such as 4800, 9600, etc.

Function Configuration Parameter Rules. These rules specify the validity constraints for the function configuration parameters. For example, if a command has a response, the response specified in the configuration must not be empty, and the response and inter-byte timeout values must be

positive integers. Only one function is allowed to be the entry point, and cycles are not allowed in the dependency sequence.

5.1.6 System Architecture

The DriverGen system consists of three modules: Parser, Executor, and Generator, as shown in Figure 5.5.

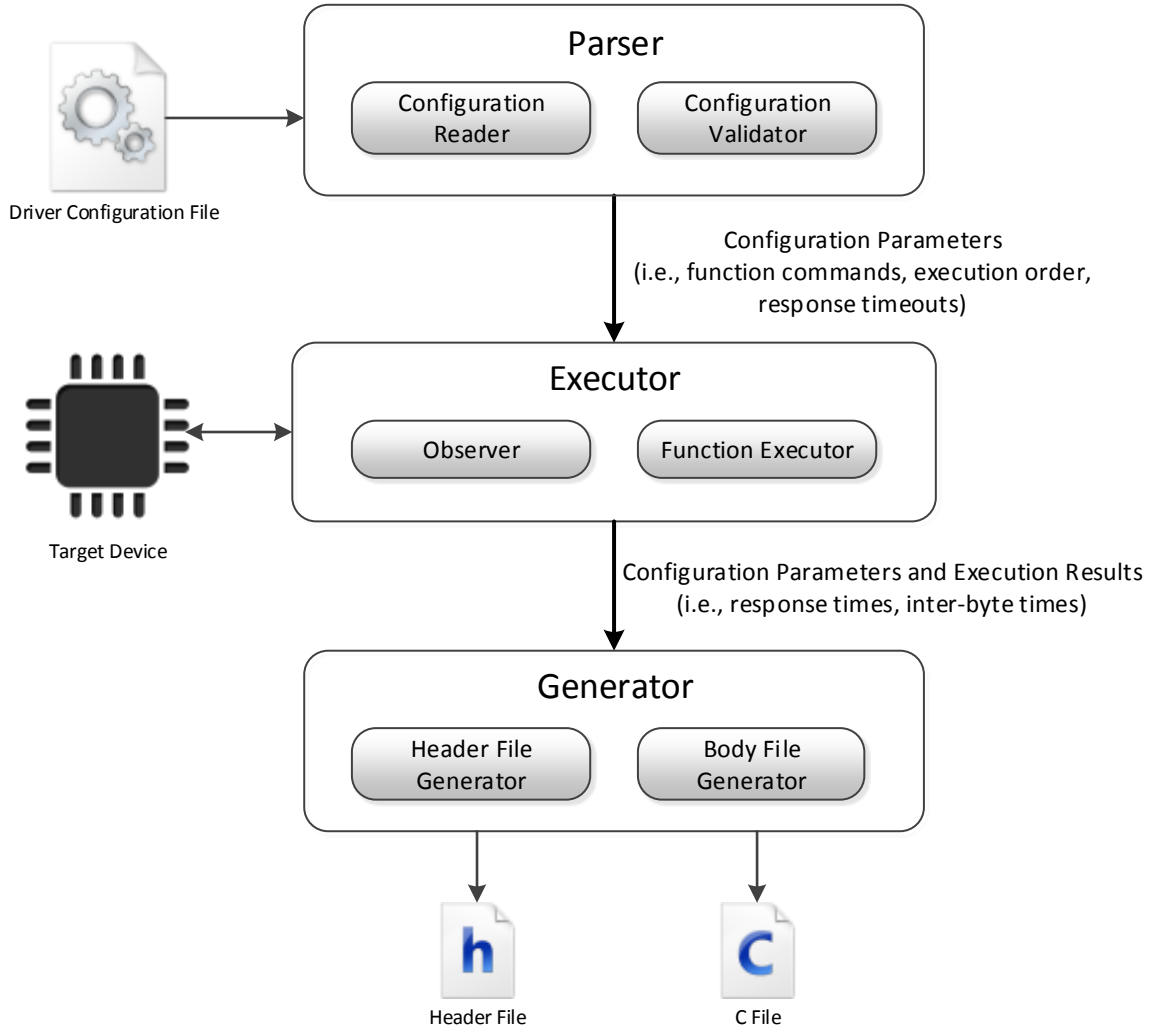


Figure 5.5: System Architecture

5.1.6.1 Parser

The parser module is used to read, parse, and validate the driver configuration. It consists of the Configuration Reader and the Configuration Validator.

Configuration Reader. The Configuration Reader is used to read the specified driver configuration file in the current working directory, and to parse the file to obtain the driver parameter values. These values will be used by the Executor and the Generator.

Configuration Validator. The Configuration Validator validates the driver parameter values. It also validates the response patterns using function `regex_compile`; the response strings will be used by the Executor. It then generates the function execution sequence to be used by the Executor based on the dependency relationships specified by the `Dependency` tags. The Configuration Validator passes the function parameters to the Executor if the parameters are valid; otherwise, it stops the process and reports the error.

5.1.6.2 Executor

The Executor module is used to execute the functions specified in the configuration file, and to handle any failures that might occur. The functions are executed in the specified order. The Executor consists of the Observer and the Function Executor, explained below. In our implementation, two FT232R chips, `F1` and `F2`, are used by the Function Executor to communicate with the Observer and the target device, respectively.

Observer. The Observer, written in C, executes on a MoteStack and is used to measure response times and inter-byte times based on the approach discussed in Section 5.1.3. The Observer accepts commands from the Function Executor and performs two operations. (i) `Prepare`: Once the Observer receives a `prepare` command, it calculates T_b, the time to transmit one bit, using the UART configuration information passed along with the `prepare` command. Next, it resets the system clock and enables external interrupts. The recording process then begins. The Observer records the times when it detects UART data frame transmissions. (ii) `Report`: Once the Observer receives a `report` command, it uses the recorded detection times to calculate response times and inter-byte times, which are then sent to the Function Executor.

Function Executor. The Function Executor, written in Java, executing on a Desktop and is used to execute the functions defined in the configuration file. Executing a function that does not expect a response simply involves sending the command to the target device. To execute a function which is expected to produce a response, the Function Executor first sends a `prepare` command to the Observer to start the time keeping process. Next, it sends the specified command to the target device and starts receiving response data. Each received byte is passed to the regular expression module. If the regular expression module returns *true*, the function has been executed successfully. The Function Executor then sends a `report` command to the Observer and saves the response and inter-byte times received from the Observer. Finally, the Function Executor starts executing the next function.

If a response/inter-byte timeout occurs, the Function Executor sends a `report` command to the Observer. Next, it changes the global response/inter-byte timeout values according to the corresponding measured times received from the Observer, and then retries the function. For example, if the reported response time is 3500 milliseconds, the global response timeout value will be changed to 3850 milliseconds (3500×1.1). If the reported response time is 0, which indicates that the target device did not respond to the command, the global response time will be doubled. The Function Executor stops and reports failure if the number of function retries exceeds the retry value specified in the configuration.

Finally, the configuration parameters and execution results, including response and inter-byte times, are passed to the Generator module.

5.1.6.3 Generator

The Generator module is used to generate the driver source code based on the configuration parameters and execution results passed from the Executor module. It consists of the Header File Generator and the Body File Generator.

Header File Generator. The Header File Generator is used to generate the header file of the driver. Listing 5.4 shows the generated preprocessing directives and function declarations for functions `gm862_wake` and `gm862_time_query` in the GM862 header file. The remaining declarations are omitted. Lines 1-3 define the library includes. Line 1 includes the standard delay library, and lines 2-3 include the UART and regular expression libraries, respectively, provided as part of the

```
1  #include <util/delay.h>
2  #include "uart.h"
3  #include "regex.h"
4
5  #define     GM862_COMMAND_WAKE                      ((char*) "AT\x0d")
6  #define     GM862_COMMAND_LENGTH_WAKE               ((uint8_t) 3)
7  #define     GM862_RESPONSE_WAKE                     ((char*) ".*OK")
8  #define     GM862_REGEX_ID_WAKE                     ((uint8_t) 0)
9  #define     GM862_RESPONSE_TIMEOUT_WAKE             ((uint16_t) 8)
10 #define     GM862_INTERBYTE_TIMEOUT_WAKE            ((uint16_t) 1)
11 #define     GM862_DELAY_WAKE                        ((uint16_t) 1000)
12
13 #define     GM862_COMMAND_TIME_QUERY                ((char*) "AT+CCLK\x0d")
14 #define     GM862_COMMAND_LENGTH_TIME_QUERY         ((uint8_t) 8)
15 #define     GM862_RESPONSE_TIME_QUERY               ((char*)
16                                  "+CCLK:(\d2/\d2/\d2),(\d2:\d2:\d2).*\s*OK")
17 #define     GM862_REGEX_ID_TIME_QUERY               ((uint8_t) 1)
18 #define     GM862_RESPONSE_TIMEOUT_TIME_QUERY       ((uint16_t) 10)
19 #define     GM862_INTERBYTE_TIMEOUT_TIME_QUERY      ((uint16_t) 1)
20
21 bool gm862_wake();
22 bool gm862_time_query();
```

Listing 5.4: Header File of GM862 Driver: gm862.h (Partial)

generated driver(s). Lines 5-11 show the constant definitions used by function `gm862_wake`. Line 5 defines the `wake` command, "AT", followed by a carriage return. Line 6 defines the length of the command, in bytes. Line 7 defines the expected response, indicating that the `wake` command was successfully executed, if "OK" is received, ignoring all the characters before "OK". Line 8 defines the identifier corresponding to the regular expression used by function `wake`. Lines 9-10 define the response and inter-byte timeout values (in milliseconds), calculated based on the execution results passed from the Executor. In this example, the response and inter-byte times are 7.3 ms and 0.0512 ms, rounded to 8 and 1, respectively. Line 11 defines the delay time, in milliseconds, introduced after `wake` is executed. Lines 13-18 correspond to function `gm862_time_query`. The response string defined on line 15 indicates that the retrieved time and date will be saved separately, accessible using function `regex_get_string` from the regular expression library. No delay directive is defined for this function because it is not specified in the configuration. Lines 20-21 show the function declarations for `gm862_wake` and `gm862_time_query`. Return values are specified to be `bool`; *true* will be returned if the command is successful, and *false* otherwise.

Body File Generator. The Body File Generator is used to generate the body file of the driver. It generates the required global variable, and then generates the `gm862_drivergen_init` function, which handles driver initialization. Finally, it generates the functions specified in the configuration file.

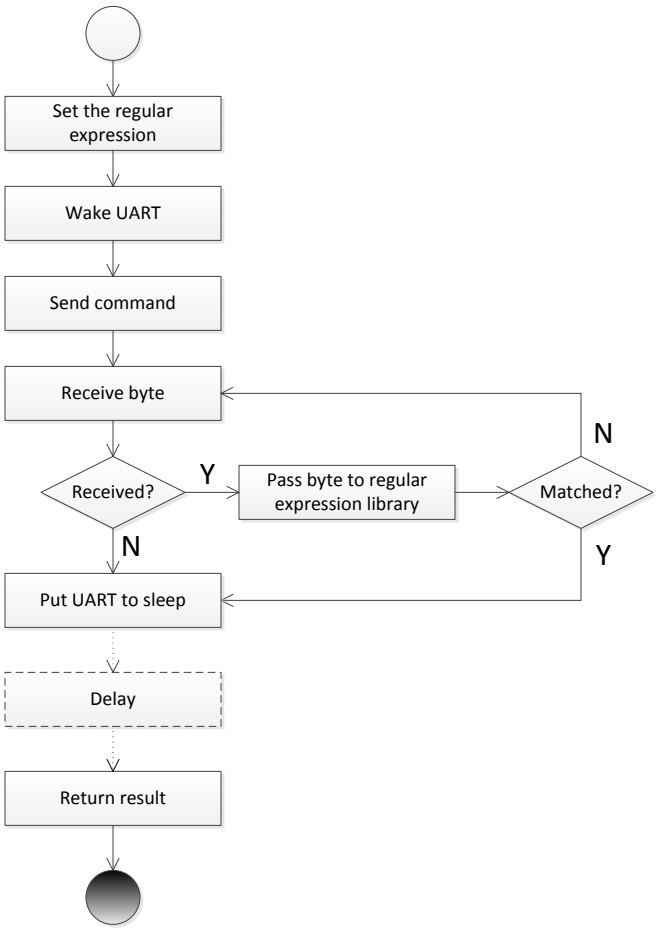


Figure 5.6: Generated Function Structure

Each generated function follows the same structure, illustrated in Figure 5.6. First, the function sets the regular expression to be used by calling `regex_use` and wakes the UART from power reduction mode. After the specified command is sent to the target device, it starts receiving the response until a response/inter-byte timeout occurs, or the received bytes match the specified regular expression. Finally, it puts the UART back into power reduction mode, introduces a delay (if specified), and returns a status result. Note that the notation of Delay is different from the others in the Figure because the delay is introduced only when it is specified in the configuration.

Listing 5.5 shows the definitions of functions `gm862_wake` and `gm862_time_query`. Lines 3-5 show global variable definitions; `receive_byte` is used as a receive buffer, `received` is used to record whether a byte was received before a timeout occurred, and `matched` is used to indicate whether the response matches the selected regular expression after a new byte is received. Lines 7-12 define the driver initialization function. Lines 8-9 initialize the UART, and lines 10-11 compile the regular expressions. The IDs of the regular expressions used by functions `gm862_wake` and `gm862_time_query` are 0 and 1, respectively. These are represented by `GM862_REGEX_ID_WAKE` and `GM862_REGEX_ID_TIME_QUERY`.

Lines 14-29 are the body of `gm862_wake`. Line 15 sets the regular expression to be used. Line 16 wakes the UART from power reduction mode. Line 17 resets `received` and `matched` to false. Line 18 sends the wake command to the GM862. Lines 19-25 receive response bytes and pass the received bytes to the regular expression library. The `while` loop terminates when a timeout occurs, or the response matches the regular expression. Line 26 puts the UART back into power reduction mode. Line 27 introduces a delay, required by the specified command. Line 28 returns the result of the command execution. Lines 31-45 are the body of `gm862_time_query`, which follows the same structure as `gm862_wake`.

5.2 Evaluation

We now present our evaluation of the driver generation approach. We introduce three serial devices, and corresponding applications developed previously to operate with functionally equivalent, handwritten device drivers. We validate the correctness of each generated driver via substitution within the corresponding application. Finally, we consider the relative performance of the generated drivers, both in terms of space and execution speed. In our experiments, the drivers and applications are implemented based on the AVR platform.

5.2.1 Test Devices and Applications

Three serial devices are used to evaluate our approach. The WH2004A is an LCD device, with 4 rows and 20 columns per row. It executes commands to display characters and perform control operations, such as moving the cursor and deleting characters. The RN131 is a standalone embedded WiFi device with built-in TCP/IP support. It executes commands to configure its WiFi

```c
#include "gm862.h"

uint8_t received_byte;
bool received = false;
bool matched = false;

void gm862_drivergen_init(){
  uart_config_t uart_config = (uart_config_t){9600, NONE, 8, 1};
  uart_init(&uart_config);
  regex_compile(GM862_COMMAND_WAKE);
  regex_compile(GM862_COMMAND_TIME_QUERY);
}

bool gm862_wake(){
  regex_use(GM862_REGEX_ID_WAKE);
  uart_wake();
  received = matched = false;
  uart_send_bytes(GM862_COMMAND_WAKE, GM862_COMMAND_LENGTH_WAKE);
  received = uart_receive_byte(&received_byte, GM862_RESPONSE_TIMEOUT_WAKE);
  while(received && !matched){
    matched = regex_feed(received_byte);
    if(!matched){
      received = uart_receive_byte(&received_byte,
          GM862_INTERBYTE_TIMEOUT_WAKE);
    }
  }
  uart_sleep();
  _delay_ms(GM862_DELAY_WAKE);
  return received;
}

bool gm862_time_query(){
  regex_use(GM862_REGEX_ID_TIME_QUERY);
  uart_wake();
  received = matched = false;
  uart_send_bytes(GM862_COMMAND_TIME_QUERY,
      GM862_COMMAND_LENGTH_TIME_QUERY);
  received = uart_receive_byte(&received_byte,
      GM862_RESPONSE_TIMEOUT_TIME_QUERY);
  while(received && !matched){
    matched = regex_feed(received_byte);
    if(!matched){
      received = uart_receive_byte(&received_byte,
          GM862_INTERBYTE_TIMEOUT_TIME_QUERY);
    }
  }
  uart_sleep();
  return received;
}
```

Listing 5.5: Body File of the GM862 Driver: gm862.c

settings, access WiFi networks, and transmit/receive data via TCP/IP. The GM862 is a quad-band GSM/GPRS cellular modem with built-in TCP/IP, FTP, and SMTP support. It executes commands to connect to the GSM network, update its time, and transmit/receive data using various protocols.

To evaluate the generated driver for the WH2004A, an application that detects door trespassing events and displays the cumulative event count on an LCD was used. The WH2004A does not respond to incoming UART commands. Therefore, the evaluation of the generated driver is focused on observable correctness only. The application displayed the event counts without any errors for 100 door trespassing events using the driver.

To evaluate the generated drivers for the RN131 and GM862, two test applications were used. The applications sense data from a group of sensors every 10 and 120 seconds, respectively, and record the execution time of each function. (This function diagnostic information was collected to estimate device power consumption.) Sensor readings and execution times are then sent to a campus server.

During each round of transmission, each application first initializes the corresponding device. Next, it connects to the server using a WiFi or GPRS network, respectively. After it finishes transmitting, the connection is closed. Finally, the server stores the sensed data along with execution times for future analysis. Each application is configured to perform 1000 transmission rounds in each test, and the average is used. Both the WiFi and cellular applications continuously send messages to the server. Each process is supported by the corresponding driver functions.

Based on the stored messages in the database, both drivers worked as expected. Table 5.1 lists the key driver functions used to support data transmission. The first column of the table lists the names of the functions. The second column briefly explains the operation each function performs. The third and fourth columns list the cumulative errors encountered for each key driver function during the tests. For example, the number of errors that occurred in functions `gm862_gsm_registered` and `gm862_gprs_registered` total the number of failed registration checks made to the GSM and GPRS networks, respectively, before a successful registration was detected. On average, the GM862 takes approximately 40 checks before successful GSM registration, and 3 checks before successful GPRS registration, with a delay of 500 milliseconds between each check, generating about 40000 and 3000 errors, respectively, in the 1000 test rounds. The registration errors in Table 5.1 are unavoidable. Note that the generated driver for the GM862 reduced the number of gm862_sleep function errors since a longer response timeout was identified. No errors were

detected during the test of other functions in the GM862 driver and the RN131 driver. Reducing the number of errors can reduce the number of retries before an operation is performed, improving the performance and energy efficiency of the driver.

Driver Function	Summary	Number of Errors for Generated Driver	Number of Errors for Handwritten Driver
gm862_wake	Powers up the GM862	0	0
gm862_gsm_registered	Acquires GSM registration status	43315	43369
gm862_gprs_registered	Acquires GPRS registration status	2675	2716
gm862_acquire_ip	Activates GPRS and obtains IP	0	0
gm862_tcp_connect	Opens a TCP connection	15	17
gm862_sleep	Puts the GM862 into low-power mode	0	17
wifly_enter_command_mode	Enters command mode	0	0
wifly_set_ssid	Sets the wireless SSID	0	0
wifly_set_remote_host	Sets the remote host address	0	0
wifly_obtain_ip	Obtains the local IP address	0	0
wifly_connect	Opens a TCP connection	0	0
wifly_exit_cmd_mode	Exits command mode	0	0

Table 5.1: Key Driver Functions for the GM862 and the RN131

5.2.2 Performance Evaluation

We next evaluate the performance of the generated drivers relative to the handwritten drivers in terms of space and execution speed. Since the LCD executes commands without sending responses back to the microprocessor, we focus on the WiFi and cellular devices.

5.2.2.1 Execution Speed

We first evaluate the execution speed of the generated drivers by sending 1000 messages containing 850 bytes to the server and tracking the execution time of each function. Figures 5.7 and 5.8 summarize the speed of the generated driver functions for the RN131 and the GM862, compared to the handwritten drivers. The x-axis represents the driver functions, and the y-axis represents the average cumulative execution time, in seconds, in a single transmission round. The functions are ordered by execution time, in decreasing order from left to right. As the figures illustrate, the generated drivers run faster than the handwritten drivers across all functions. The speed-up is achieved by reducing the time spent waiting for each response. The cumulative speed-up is proportional to the number of executions of each function in a transmission round. For

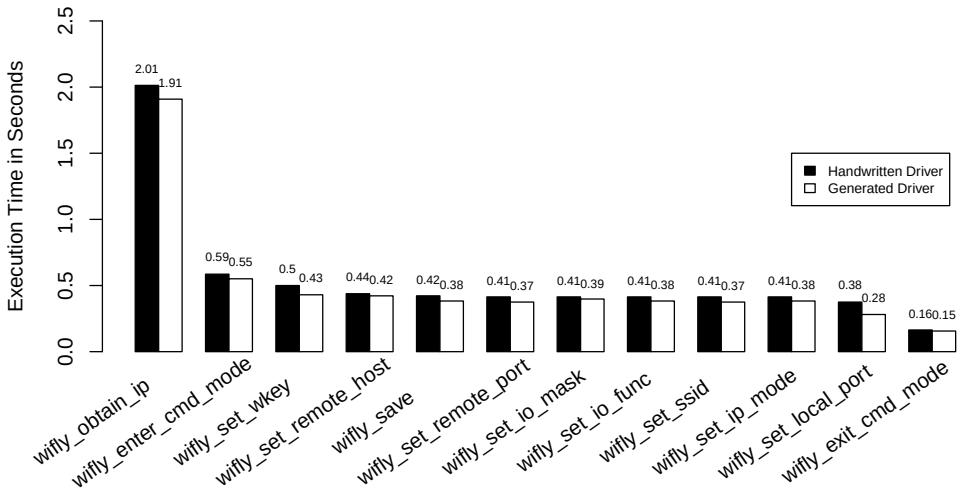


Figure 5.7: Driver Function Execution Time (RN131)

example, in each round, the `gm862_gsm_registered` function executes approximately 40 times before detecting a valid registration status. Therefore, it shows a high speed-up in Figure 5.8. For

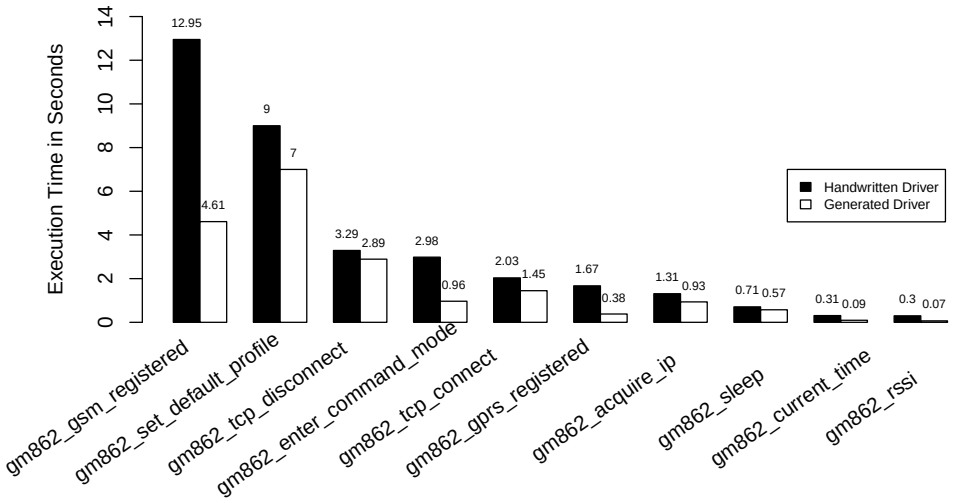


Figure 5.8: Driver Function Execution Time (GM862)

the GM862, the overall execution time in each round is 48.50 seconds for the generated driver, compared to 59.60s for the handwritten driver. For the RN131, the overall execution time in each round is 11.99s for the generated driver, and 14.68s for the handwritten driver. By reducing the execution time of each command, the drivers reduce the time the executing device spends in an active state, thereby saving power, extending device lifetime.

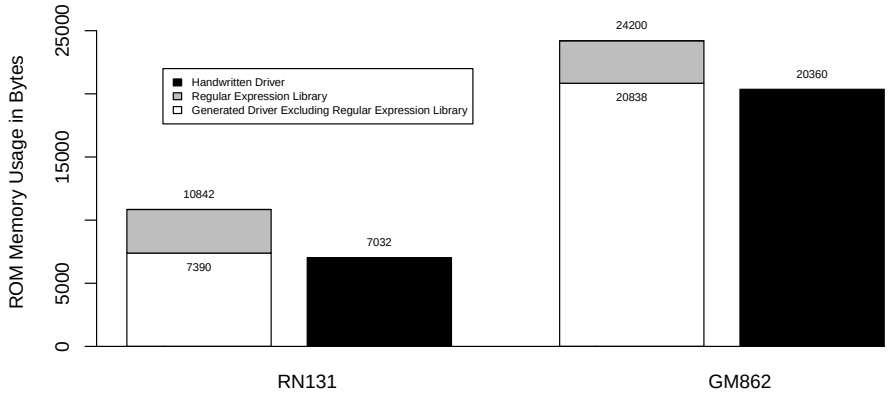


Figure 5.9: Driver ROM Usage (GM862 and RN131)

5.2.2.2 Memory Usage

We next evaluate the memory overhead introduced by the generated drivers. *Avr-size* [14] is used to collect the memory data. Figure 5.9 summarizes the drivers' program memory (ROM) usage. The x-axis represents the drivers, and the y-axis represents size, in bytes. The hashed area represents ROM overhead introduced by the regular expression library. The ROM overhead is approximately 3400 bytes for both drivers. Excluding the regular expression library overhead, the size of the generated driver is comparable with the handwritten driver.

Figure 5.10 summarizes the drivers' data memory (RAM) usage. Again, the x-axis represents the drivers, and the y-axis represents size, in bytes. The hashed area represents RAM overhead introduced by the regular expressions used in the generated driver. The RAM overhead is introduced mainly by the regular expression library and is closely related to the number of regular expressions used. The GM862 driver uses 8 regular expressions, requiring 503 bytes of overhead. The RN131 driver uses 5 regular expressions, requiring 335 bytes of overhead. On average, each regular expression requires approximately 65 bytes. Since the GM862 requires more regular expressions, the overhead for the GM862 driver is slightly larger than for the RN131 driver.

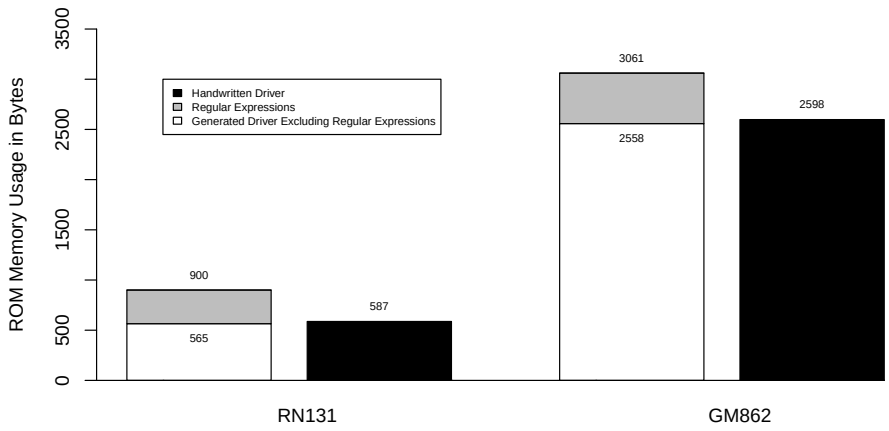


Figure 5.10: Driver RAM Usage (GM862 and RN131)

5.3 User Study

To assess the usability of DriverGen in comparison with driver development in C, we conduct a small user study. In this section, we discuss the experimental setup and the associated results.

5.3.1 User Study Setup

5.3.1.1 The Participants

Ten graduate students from different research areas with various levels of programming skill were randomly selected. Nine of these students were majoring in Computer Science, and one was majoring in Forestry, with no prior programming experience. The students were divided into two groups. The first group, *group C*, wrote drivers in C, and the second group, *group D*, used DriverGen to generate their drivers. The Forestry student was assigned to group D. Five of the remaining nine students were assigned to group C, and four were assigned to group D via random selection.

5.3.1.2 The Driver

The RN131 WiFi chip was used in this study. Three functions, `wifly_enter_command_mode`, `wifly_get_wlan_info`, and `wifly_exit_command_mode`, were selected for implementation.

Group C was provided with the header file `wifly.h`, the body file `wifly.c`, and the UART library. The command, response, and timeout values for each function were provided in the

```
S1. Rate your level of familiarity with 8-bit embedded systems.
S2. Rate your level of familiarity with serial communication in
    8-bit embedded systems.
S3. Rate your level of familiarity with driver development in 8-bit
    embedded systems.
S4. Rate your level of familiarity with the C programming language.
```

Listing 5.6: User Study Pre-Survey

header file as macros using "#define". The three functions used in the study were defined, but not implemented. The supporting functions, such as chip initialization were provided.

Group D was provided with an incomplete XML configuration file, `wifly.xml`. Each participant was required to create a configuration block for each function. The supporting elements in the configuration file, such as basic driver information, global definitions, and UART parameters were provided in the configuration file, analogous to the information provided to group C.

5.3.1.3 The Surveys

Two surveys were used in the study. The pre-survey, shown in Listing 5.6, was used to assess participants' knowledge of the C programming language and embedded systems. It consists of four Likert-style questions, which ask participants to rate their level of familiarity with 8-bit embedded systems, serial communication in 8-bit embedded systems, driver development in 8-bit embedded systems, and C programming in general. Each question is weighted from 1 to 5, with 1 being *not at all familiar*, and 5 being *expert*.

The post-survey, shown in Listing 5.7, was used to assess the difficulty encountered by participants in writing the driver in C or generating the driver with DriverGen. It consists of two Likert-style questions that ask participants to rate the level of difficulty experienced writing the driver in C or writing the driver with DriverGen for groups C and D, respectively. Each question is weighted from 1 to 5, with 1 being *easy*, and 5 being *difficult*.

5.3.2 User Study Process

The two groups were scheduled at different times to avoid interference.

```
S5. Rate your level of difficulty you experienced writing the driver
    in C (group C).
S6. Rate your level of difficulty you experienced writing the driver
    with DriverGen (group D).
```

Listing 5.7: User Study Post-Survey

5.3.2.1 The Process: Group C

After participants completed the pre-survey, a short tutorial session was provided to cover the basic concepts involved in serial device driver development, including i) embedded systems, ii) serial devices and serial communication, iii) serial device driver structure, iv) the UART library, v) the WiFi chip, and vi) the functions to be implemented. The (incomplete) driver source code was then sent to each participant, who was then asked to complete the three functions described in Section 5.3.1.2. No additional help was provided. The time each participant spent writing the driver was recorded. Each participant was then asked to complete the post-survey. Finally, the drivers were compiled and tested on our hardware.

5.3.2.2 The Process: Group D

The process followed by group D was analogous. After participants completed the pre-survey, a short tutorial session was provided to cover the basic concepts involved in serial device driver development and using DriverGen, including i) embedded systems, ii) serial devices and serial communication, iii) DriverGen, iv) the WiFi chip, and v) the functions to be implemented. The (incomplete) XML driver configuration file was then sent to each participant, who was then asked to complete the three function configurations described in Section 5.3.1.2. No additional help was provided. The time each participant spent generating the driver was recorded. Each participant was then asked to complete the post-survey. Finally, the drivers were compiled and tested on our hardware.

5.3.3 User Study Results

We first analyze the survey results, summarized in Table 5.2. Based on questions S1-S4 on the pre-survey, we assign each student a total score, reflecting their knowledge of embedded systems and their level of programming skill; the score appears in the seventh column. The average scores

of group C and D are close, at 8 and 6.8, respectively. We next study the time each participant spent writing the driver; this figure appears in the last column. The average times to complete the driver are 31 minutes and 13.4 minutes for groups C and D, respectively. The results shows that DriverGen significantly improved the development speed (ignoring correctness, which we consider next). We next consider questions S5-S6 on the post-survey. As shown in the eighth and ninth columns, average participant rating for the level of difficulty encountered while writing the driver with DriverGen is 1.2, whereas the average rating for group C is 3.6. Note that participant 7 is the Forestry student with no programming experience. The time this participant spent writing the driver is close to the average of group D, and the participant rated the difficulty of using DriverGen as 1 (easy). The results suggest that writing serial device drivers with DrivenGen is easier than using C, and DriverGen is usable even for people with little or no programming experience if only anecdotally.

Group	**Participant**	**S1**	**S2**	**S3**	**S4**	**Score**	**S5**	**S6**	**Time (minutes)**
C	1	3	3	3	3	12	2	-	14
	2	1	1	1	2	5	3	-	34
	3	1	1	1	2	5	5	-	50
	4	3	3	2	3	11	3	-	29
	5	2	2	1	2	7	5	-	28
	Average					8	3.6		31
D	6	2	1	1	3	7	-	1	12
	7	1	1	1	1	4	-	1	14
	8	2	2	3	3	10	-	1	16
	9	1	1	1	3	6	-	1	15
	10	1	1	1	4	7	-	2	10
	Average					6.8		1.2	13.4

Table 5.2: Survey Summary

The drivers generated with DriverGen all compiled, followed the same structure, and worked as expected. Unfortunately, in group C, one driver was not completed, and the remaining four did not compile, mostly due to syntax errors. We fixed the drivers with simple syntax errors, e.g., missing semicolon or parentheses. However, some drivers still did not compile. One driver compiled but did not work because it did not call `uart_wake` to wake the UART before used it.

5.4 Summary

In this chapter, we presented our approach to automatically generating serial device drivers. We first described the UART and regular expression libraries used in the generated drivers. We then described our approach to accurately measuring response and inter-byte times. We next discussed the driver configuration syntax and the driver generation process. Finally, we presented an evaluation of our approach, including a small user study to assess usability. Results show that our approach automates the serial device driver generation process and improves the performance and energy efficiency of generated drivers relative to their handwritten counterparts. The user study suggests that DriverGen is easy to use and simplifies the driver development process, yielding higher-quality drivers.

Chapter 6

Related Work

In this chapter, we summarize the most relevant related work. Section 6.1 summarizes work related to our distributed network simulation approach. Section 6.2 summarizes work related to our SEU protection approach. Section 6.3 summarizes work related to our automatic serial driver generation approach.

6.1 Network Simulation

Levis *et al.* [28] introduce TOSSIM, a simulator for TinyOS applications. TOSSIM, which is integrated with the TinyOS distribution, simulates native code by replacing hardware-dependent components with corresponding (desktop-based) simulation components. TOSSIM uses a simple discrete event simulation model to simulate the event-driven execution of TinyOS. TinyOS's events are mapped to discrete simulation events, which are then enqueued for later execution. In TOSSIM, event execution time is not considered, making the simulation time-inaccurate. Moreover, TOSSIM is neither parallel, nor cycle-accurate; no synchronization techniques are used. Landsiedel *et al.* present TimeTOSSIM, an extension of TOSSIM, which tracks event execution time to achieve better time fidelity than TOSSIM [25]. By mapping each line of source code to a set of instructions, and then retrieving the number of clock cycles consumed by each instruction for the corresponding microprocessor, TimeTOSSIM computes the execution time of each line of source code. To achieve this, TimeTOSSIM injects code into the target program to track execution time. Since the code is injected at the source code level, the execution time estimation is not perfect. However, results

show that TimeTOSSIM achieves timing accuracy of over 99%. Shnayder *et al.* present PowerTOSSIM, another extension to TOSSIM, which supports accurate, per-node power consumption estimation for TinyOS-based sensor networks. In PowerTOSSIM, hardware peripherals (e.g., radio, EEPROM) are represented as TinyOS components, which are instrumented to collect each peripheral's activities during simulation. The number of clock cycles executed by each node is computed by a code-transformation technique. Peripheral activities and cycle counts are used within an energy consumption model to estimate the power consumption of each node. None of these systems — TOSSIM, TimeTOSSIM, and PowerTOSSIM — are parallel, so no synchronization techniques are used.

ATEMU [44] is a cycle-accurate sensor network simulator, which emulates each sensor node at the instruction level and simulates network communication using a wireless medium model. Each hardware component, including the microprocessor, the radio, and internal timers, is implemented as a module in ATEMU. Node interactions are simulated based on a radio propagation model, which implements each radio as a receiver/transmitter pair, assigned a frequency and transmission power level. ATEMU's goal is to achieve accurate simulation by modeling the operations provided by the underlying hardware platform. Although ATEMU achieves high fidelity simulation, it is much slower than TOSSIM [53].

Other authors have focused on parallel sensor network simulation techniques. Avrora [53] is a sensor network simulator for AVR-based applications. Avrora achieves cycle-accuracy, emulating the AVR hardware platform. Implemented in Java, Avrora uses Java multithreading to achieve parallel simulation of wireless sensor networks. Each simulated node is run as a separate Java thread and is treated as an event-generating object. Given that each simulated node is able to communicate with all other nodes, all threads periodically synchronize at an interval defined by the time required for the radio chip to complete the transmission of one byte of data. Each node runs freely until the end of a synchronization interval, and then waits until all other nodes in the network report that they have passed that point. Interval synchronization is costly, however, especially for distributed simulators, since the physical networking delay introduced per message significantly increases the overall synchronization delay.

Jiang *et al.* [24] present SnapSim, an extension of Avrora, based on an optimistic simulation algorithm with support for backtracking and re-execution. The approach adopts the idea of probing executions to find the nearest transmission time for each node in the network, and to backtrack

nodes that have passed the global nearest transmission time. Nodes that proceed beyond the global nearest transmission time and thereby miss a read channel event must backtrack and re-execute to correct for the missed transmission. Snapshots containing the simulation state of each node are used to restore nodes to correct former states. Although SnapSim significantly improves simulation performance, the computational limits of the host system become a performance bottleneck.

Legedza *et al.* [26] present *local barriers* and *predictive barrier scheduling*, two approaches to reducing synchronization overhead in parallel simulation systems. Local barriers are used when synchronizing nearest neighbors of a simulated node (i.e., adjacent nodes in a simulated network). These nodes are simulated on the same processor to reduce inter-processor communication. Predictive barrier scheduling takes into account both compile-time and runtime analysis to predict the amount of time that a process will not communicate with other processes. The prediction is used to allow extended execution periods before synchronizing.

Bajaj *et al.* [3] introduce GloMoSim, a scalable, parallel simulation environment for large networks. GloMoSim adopts a layered architecture like most network systems, simplifying the integration of different components at different layers. For example, a TCP layer is integrated into GloMoSim by extracting source code from FreeBSD [15]. Similar to local barriers, each simulation processor simulates multiple geographically-proximate nodes to support fast parallel simulation. Both conservative and optimistic synchronization protocols are used in GloMoSim. Conservative synchronization protocols only allow events to execute in time-stamp order. Three conservative synchronization protocols are used, including null-message-based synchronization, conditional event synchronization, and accelerated null-message synchronization [29] [13] [2]. Optimistic protocols allow events to execute out of time-stamp order, but detect and correct dependencies using techniques such as checkpointing and rollback.

Naoumov *et al.* [37] present an extension to the NS-2 [33] simulator to simulate large ad hoc networks. Two node organization strategies are introduced to improve simulation performance. The *grid-based* approach divides the simulation area into cells, forming a grid. During transmission, the cells covered by the radio are calculated by the position and power level of the transmitting node. Nodes participating in each data transmission are readily identified. The *list-based* approach forms a doubly-linked list, ordering nodes by their x-coordinates. Due to the lower cost of identifying affected nodes, results show that the list-based approach achieves better simulation performance than the grid-based approach.

The work presented above focuses on sensor network simulation, based on either parallel or non-parallel strategies. However, these approaches suffer the performance bottleneck caused by the limited computational power of a single execution system. Our work improve simulation performance from this perspective. We describe a distributed sensor network simulator that simulates sensor network applications within cluster environments and improves simulation performance through distributed execution.

6.2 Memory Protection

Single event upset mitigation has been explored from both hardware and software perspectives. Common hardware solutions focus on hardware design modifications to prevent radiation from causing SEUs, and extensions to correct SEUs. Common software solutions adopt one or more error-correcting codes to detect and correct SEUs. In practice, hardware and software are usually used together to achieve better SEU protection, and to balance among cost, execution efficiency, and power consumption.

One of the primary hardware-level radiation hardening approaches is Silicon-on-Insulator (SoI) technology, used in microprocessor fabrication [4]. In this approach, transistors are placed on a thin layer of silicon, which is then placed on top of an insulator. The design improves the circuit's tolerance to highly-charged particles, reducing the chance of SEU occurrence. Moreover, SoI technology reduces the capacitance of the switches and the size of the processors, and thus reduces the area over which highly-charged particles can strike, statistically reducing the likelihood of impacts, and therefore errors.

Irom et al. [23] compare SEU error rates in SoI microprocessors to conventional microprocessors. They subject both types of microprocessors to proton impacts within a cyclotron, and to heavy-ion impacts within an accelerator, both of which are known to cause SEUs in processors. From these tests, Irom et al. conclude that due to the significant reduction of cross-sectional area in SoI microprocessors, SEU rates are lower than those in commercial microprocessors. Though SoI technology practically protects systems from SEUs, it prevents developers from using commercial off-the-shelf (COTS) devices, increasing system cost due to the high price of SoI circuits.

Redundancy is a widely used fault-tolerance technique that can be implemented at both the hardware and software levels. The Triple Modular Redundancy (TMR) approach [32] synchronously

executes instructions on three unique systems. A voting module is used to compare the results and choose the common result. Due to the low probability that more than one SEU will occur simultaneously at the same geographic location in more than one device [55], TMR is a popular SEU protection technique and allows the use of COTS components. However, hardware-based TMR introduces significant hardware overhead and power consumption, which can present concerns for weight-limited and power-critical systems. It also increases hardware and software complexity.

Time Redundancy [8] is a software-only redundancy technique, which runs each instruction three times on a single processor. The results are stored, and a voting module is invoked to yield the (most) common result. Error Detection by Duplicated Instructions (EDDI) [40], a variation on Time Redundancy, duplicates each instruction during the compilation phase and assigns each different registers and memory space. As a result, EDDI is able to protect systems from not only data SEUs, but also instruction SEUs. Time Triple Modular Redundancy [8] is a combination of time redundancy and hardware-based TMR. Each instruction is executed by three unique systems, as in standard TMR, but the systems execute the instructions in different clock cycles in a time-redundant fashion. This allows more instructions to be executed in parallel.

Others have worked on replacing memory cells by SEU-hardened cells and circuits. Weaver *et al.* [58] present LRAM, which adds decoupling resistors to slow down long pulses, enabling cells to differentiate pulses generated by an SEU versus a write signal. She et al. [48] improve the design of conventional latches by implementing an error detection circuit and integrated multiplexer. While conventional latches are susceptible to voltage changes caused by SEUs, the proposed latch uses an error detection circuit that checks for faults using the precharge and discharge operations. The latch uses a multiplexer to output a corrected signal based on the faults detected by the error detection circuit. The authors found that the proposed latch introduces little overhead and offers good performance, as well as better SEU protection than conventional latches.

A watchdog timer (WDT) [22] is a timer used to detect and recover from system crashes by repeatedly querying the protected system and resetting the system if no response is received. A software-based WDT is straightforward to implement and introduces little code and performance overhead. Since the timer runs along with the protected system on the same device, it suffers the risk that an SEU may cause the WDT itself to malfunction, compromising the protection it is designed to afford. Despite increased cost, hardware-based WDT introduces a reliable solution to protecting systems from serious faults. Note, however, that WDT is typically used with other techniques since

it only detects severe system faults.

Shirvani *et al.* [49] present analysis results for common error detection and correction (EDAC) approaches used to detect and correct memory bit errors, including SEUs. The hardware-based approaches provide better reliability over time, but add more cost. Although more cost-effective, the reliability of software approaches tends to decrease faster than hardware-based approaches. However, the reliability loss rate is low, making software approaches widely used as memory error protection solutions. The authors demonstrate that low-cost, software-based EDAC approaches can enhance memory protection and system stability.

Mhatre *et al.* [36] present a hybrid radiation fault method, combining both fault tolerant hardware and EDAC techniques. The combination is applied in the design of a pico-satellite's on-board computer, with a goal of providing detection and correction from SEUs without adding expensive radiation-hardened devices. A Hamming Code component is used to detect and correct memory bit errors. A watchdog hardware component is introduced to prevent the system from possible crashes. TMR is also used, enabling the use of inexpensive commercial microprocessors.

Similarly, Dutton and Stroud present a design implemented in configurable logic blocks for SEU detection and correction in the configuration memory of field programmable gate arrays (FPGAs) [10]. The architecture of the Xilinx Virtex FPGA is modified to implement an SEU controller that uses Hamming codes and parity values to detect and correct single-bit errors in memory. This combination of Hamming and parity can also detect multiple bit upsets, but correction is still not possible. The benefits include the protection of the controller from SEUs and the high speed of error detection and correction, as compared to other methods.

Many authors have considered memory space protection at the operating system level. For example, Erlingsson *et al.* [12] present XFI, a software-based protection system based on memory access control and integrity guarantees. Protection is achieved by performing static source code analysis, followed by runtime checks on system states. The static analysis step inserts code into the target program and verifies that there are enough guards across all execution paths. At runtime, memory accesses and control flow are checked by the injected guards that protect the system from execution violations, particularly memory access violations. Witchel *et al.* [62] present Mondrix, a modified Linux Kernel integrated with Mondriaan Memory Protection (MMP) [61]. It offers memory protection based on a protection interface to MMP, with complementary and hardware support. Mondrix protects system stacks through each thread's permission table, maintained by the

MMP hardware. Each table contains a stack base pointer, a stack frame pointer, and the read/write permissions for to the current stack frame. Stack frame protection is achieved by allowing only the thread granted with write permission to access the frame. While these approaches protect memory, the protection is mainly via access control, without regard for bit errors in memory, making the system still vulnerable to SEU errors.

The work presented above focuses on memory protection against SEUs, based on either software or hardware techniques. However, additional hardware introduces additional expense and weight to a target system. The software approaches increase system complexity and development time. Our software-only approach protects system memory from SEUs through duplication, validation, and recovery without introducing additional hardware. Operating at the assembly level, our approach introduces small overhead, and is nonintrusive and language neutral.

6.3 Serial Device Driver Generation

Merillon *et al.* [34] present Devil, an interface definition language used to specify the functional interface of a device. Their approach allows developers to write driver specifications in Devil, which are then automatically translated to low-level driver code. During the translation process, critical safety properties are checked to detect errors. However, compared to our XML-based approach, Devil is complex and requires highly experienced developers to write hardware specifications.

Zhang *et al.* [63] present DEVEL+, a domain-specific language designed to enable developers to write and verify device drivers. Instead of focusing on the automation process, DEVIL+ concentrates on using a specification language to simply the driver development process. However, this method forces developers to learn a new language to specify the operations of the device. Further, developers must still develop a new driver for each new device. These problems are eliminated in automated driver generation.

Ratter *et al.* [45] discuss automated driver synthesis, which relies on modeling the structure of a driver, supporting a correct driver generation process. The approach creates a finite state machine based on specifications of both the target device and the host operating system. The state machine is used to generate the driver, guiding the driver generation process and reporting errors. When applied to embedded driver generation, this approach experiences additional challenges, such as limited memory availability, limited computational power, and constrained energy reserves.

Chou *et al.* [6] describe a method for synthesizing multiple device drivers for a given microprocessor, with a goal of optimizing pin allocations. Based on the attributes of the microprocessor and the device, their approach uses a recursive algorithm to assign microprocessor pins to each controlled device based on need. A pin can either be assigned to only one device or be shared by multiple devices, increasing the upper-bound on controlled devices. A time-based multiplexing method is introduced to handle communication between the devices and shared pins. Functions used to control communication signals are grouped together based on their tasks. However, this approach performs poorly because of its time-multiplexed nature, while our approach focuses on improving the performance of serial device drivers.

Termite [46] is another synthesis tool for automating device driver generation. A formal specification is used to specify the interface between a target device and a host operating system. Termite links the device and the operating system based on these specifications, which define the intended device behavior and the programming model and services provided by the operating system. Since the interface specifications are independent of the operating system, any changes to the device side do not affect the OS side, and vice-versa. Termite device interface specifications can be reused across operating systems, reducing the time required to port a driver from one operating system to another. This also increases the reusability of existing OS-independent Termite device interface specifications.

O'nils *et al.* [41] present a method for generating device drivers by modeling the desired behaviors in ProGram, a grammar-based protocol specification language based on sequences of permissible events. The technique takes an architecture-independent protocol described in ProGram, processor and bus interface descriptions, and a target operating system description as inputs to generate a device driver. Results show that by separating the architecture-dependent and architecture-independent components of a driver, reusability of operating system and processor interfaces can be significantly increased.

Pendharkar and Kolathur present DDGEN [47], a driver generation tool designed for embedded systems. A domain-specific language, Device Programming Sequence (DPS), is designed to specify the functions and attributes of the target device and is used by hardware developers to formalize the hardware interface. Another specification language, Runtime Specification (RTS), is used by software developers to capture software characteristics, such as the OS and driver models, the communication model, interrupt services, etc. Based on DPS and RTS, DDGEN generates device

drivers for the specified operating system.

Ambiguities and inconsistencies commonly found in natural language-based specifications are caused mainly by a lack of formal semantics. In [30], the authors present a semantic model used to formalize hardware/software interface specifications to identify these problems. To enable synthesis, the approach requires formal specifications of the target device and the host operating system. Our approach requires a configuration file that specifies the device driver parameters, such as commands, expected responses, and expected response times. Written in XML, our configuration file is easier for developers to create, understand, and modify compared to formal specification languages.

The work presented above focuses on device driver generation. Some approaches are not designed for embedded systems and are not able to capture embedded system behavior. Other approaches target drivers for embedded systems. However, they don't focus on improving the performance of the driver for a specific device (e.g., a serial device). They typically require the use of complex specification languages, reducing case-of-use and increasing the chance of programmer error. Our approach, in contrast, focuses on automated driver generation for serial devices, based on a straightforward XML configuration syntax. It improves driver performance by accurately measuring the timeouts associated with each command and response.

Chapter 7

Conclusion

Embedded network systems have become critical components of many aspects of our daily lives. At the same time, the associated applications have become increasingly complex. As a result, new challenges are presented to system developers. We have identified three challenges encountered in embedded network systems. First, parallel simulation is commonly used during embedded network application development, where simulation speed is the key performance concern. Fast sensor network simulation saves time (and money) during application development. Second, beyond the applications used in our daily lives, embedded systems are widely used in spacecrafts and satellites. The single event upset (SEU) is one of the most common types of damage caused by radiation in such environments. Protecting embedded system memory from SEUs is a critical and challenging, but necessary task in such environments. Finally, serial-based peripherals are among the most popular peripherals used in embedded systems. Due to the significant repetitive work involved in serial device driver development, an automated serial driver generation approach can significantly improve the efficiency and reliability of serial device driver development. In the sections below, we summarize the challenges and our contributions to address them.

7.1 Challenge Summary

7.1.1 Network Simulation

In this component of the dissertation, the main challenge addressed by our work centers on enabling sensor network simulation across multiple physical systems to improve embedded network simulation performance. We present D-SnapSim, a distributed version of SnapSim, which runs on a cluster. Threads executing on the same system communicate via shared variables. Communication between processes executing on different systems is handled via Java Remote Method Invocation (RMI). To achieve scalability, each simulation is tailored based on configuration options, including cluster size, number of SnapSim instances, and number of simulated nodes per SnapSim instance. As a result, D-SnapSim is effective on clusters of varying size and computational power, maximizing overall simulation performance. The experimental results show that D-SnapSim significantly improves the performance of SnapSim as network size and bitrate increase.

7.1.2 Memory Protection

In this component of the dissertation, the main challenge addressed by our work centers on protecting system memory from SEUs. We design and implement a software-only approach to protecting the system stack from SEUs through duplication, validation, and recovery. We extend AVR-GCC to inject assembly code into a target application to achieve memory protection without adding extra hardware. The injected code modifies the function invocation and return processes. Before a callee enters its function body, it computes a checksum over its caller's stack frame and saves both the checksum and the stack frame. Before the callee returns, it verifies the caller's stack frame using the saved checksum. If a bit error is detected, the caller's stack frame is overwritten using the saved duplicate. Our approach operates at the assembly level, so it is language neutral. It achieves a stack protection success rate of over 99% for typical embedded programs.

7.1.3 Serial Device Driver Generation

Finally, in the third focus area, the main challenge addressed by our work centers on designing and implementing a system that provides automated serial device driver generation and accurate response time measurement. We present a configuration-based tool that simulates the exe-

cution process of a driver by issuing specified commands to the target device and receives and parses the corresponding responses. Based on the simulated execution results, a target device driver is generated. Device's response times affect the amount of time a microprocessor spends in the reduced power state, which affect the energy efficiency of the driver. A hardware component is introduced to accurately measure the device's response times, which are used to improve the performance and energy efficiency of the driver. The experimental results show that the generated drivers work as expect and improve performance and energy efficiency compared to handwritten equivalents.

7.2 Contribution Summary

7.2.1 Contribution 1

While parallel simulation techniques can increase simulation performance and scalability, the benefits are constrained by the limited computational power available on a single physical system. We introduce a *distributed* sensor network simulation technique, which supports cluster-based execution to improve simulation performance.

D-SnapSim extends SnapSim to support cluster-based execution. D-SnapSim is realized as a network of SnapSim instances, each of which runs on a single physical machine. To balance synchronization overhead among threads and physical systems, multiple cluster configurations are possible. The experimental results reflect a total of 453 hours of physical execution time across networks of size 2, 4, 8,..., 1024. The results show that D-SnapSim is faster than SnapSim, even for applications with high bitrates. For XnpCount, which has a high bitrate (9500 bits/s in a network with 8 nodes), D-SnapSim is 4 times faster than SnapSim when the simulated network size is 128, and 10 times faster when the simulated network size is 1024.

7.2.2 Contribution 2

The SEU is among the most common types of faults introduced by radiation, posing significant risk to spacecraft embedded systems. Modern approaches to guarding against such faults typically introduce additional hardware to detect and correct SEU errors in target systems. We present a software-only approach to protecting embedded system memory from SEUs.

Our approach focuses on the system stack, which is the most important and dynamic region

in memory. The stack is protected by injecting auxiliary assembly code within the target program. The prototype implementation is based on the AVR architecture, but is easily adapted to other architectures. We provide a study on protection efficacy to analyze the probability of successful SEU protection as SEU frequency increases. Results show that for typical programs, our approach achieves a stack protection success rate of over 99%. Experimental results for both ROM and execution (speed) overhead have also been provided, using three applications with different degrees of stack dynamism. Since the size of each injected code segment is fixed, code space overhead depends only on the number of functions in the target program and the size of each function. Speed overhead depends largely on function frame size and stack dynamism, as well as the occurrence rate of SEUs.

7.2.3 Contribution 3

Writing serial device drivers involves a large amount of repetitive work because of the standard command/response pattern that most serial-based devices follow. Moreover, the timing parameters associated with each command are important factors that affect the amount of time a microprocessor can spend in a reduced power state, impacting energy efficiency. We present a system that simulates communication between the microprocessor and a serial device, measures the timing parameters of each command, and generates a corresponding device driver based on the results. The system, written in Java, runs on a desktop and uses a USB-to-serial interface chip, the FT232R, to communicate with the target device. After the configuration file is read, parsed, and validated, the system starts executing the command sequence. C header and body file are generated based on the execution results.

Three serial devices and previously developed drivers and applications were used to evaluate our approach via driver substitution. Results show that the drivers perform correctly. They do introduce memory overhead because of an integrated regular expression library. The overhead is proportional to the number of regular expressions used in each driver. However, the execution time of each command is reduced compared to the handwritten driver. As a result, driver performance is increased, and improved energy efficiency is achieved. (User study conclusion coming soon)

7.3 Expected Impact

Our contributions offer significant benefit to embedded application developers. The distributed network simulator enables developers to more efficiently simulate large-scale embedded network applications within cluster environments. The software-only memory protection approach enables developers to introduce SEU protection to embedded system memory without adding additional hardware. DriverGen enables developers to easily develop high-performance, energy efficient serial device drivers. We addressed three important challenges in embedded systems research. We expect that our work will help developers to develop embedded applications that are faster, safer, and more energy efficient.

Appendices

Appendix A Memory Protection Test Applications

```
#include <avr/io.h>

#define F_CPU 10000000L

int a(){
  int i = 1;
  int j = 2;
  int k = i+j;
  int l = i+j+k;
  int m = i+j+k+l;
  int n = i+j+k+l+m;
  uint8_t u = 100;
  while(u>0){
    u--;
  }
  return n;
}

int main(void){
  while(1){
    a();
    uint8_t u = 100;
    while(u>0){
      u--;
    }
  }
}
```

Listing 1: Delay

```c
#include <avr/io.h>

#define F_CPU 10000000L

int a(){
  int i = 9;
  i = b();
  return i;
}

int b(){
  uint16_t i = 10;
  int j = 12;
  int k = c(1, i, j);
  k++;
  return k;
}

int c(int a, uint16_t b, int d){
  int x,y,z;
  x=a+d;
  y=d-a;
  z=x+y;
  return z;
}

int main(void){
  uint32_t l;
  l = 243;
  uint8_t k;
  k = 32;
  uint16_t i;
  i = 100;
  int j;
  j=300;
  while(1){
    int result = a();
    result++;
  }
}
```

Listing 2: Double Function Call

```c
#include <avr/io.h>

#define F_CPU 10000000L

uint32_t counter = 0;
uint32_t i =0;
uint16_t size;
uint16_t addr;
uint8_t* p;

uint16_t fibonacci_recursive(uint16_t n){
  if (n == 0){
    return 0;
  }

  if (n == 1){
    return 1;
  }

  return fibonacci_recursive(n-1) + fibonacci_recursive(n-2);
}

int main(void){
  while(1){
    fibonacci_recursive(10);
  }
}
```

Listing 3: Fibonacci

Bibliography

[1] Pramod V Argade and Michael R Betker. Apparatus and method for computer processing using an enhanced Harvard architecture utilizing dual memory buses and the arbitration for data/instruction fetch, November 19 1996. US Patent 5,577,230.

[2] Rajive L. Bagrodia, Mineo Takai, and Vikas Jha. Performance evaluation of conservative algorithms in parallel simulation languages, 1998.

[3] L. Bajaj, M. Takai, R. Ahuja, and R. Bagrodia. Simulation of large-scale heterogeneous communication systems. In *Military Communications Conference Proceedings, 1999. MILCOM 1999. IEEE*, volume 2, pages 1396–1400 vol.2, 1999.

[4] G. K. Celler and Sorin Cristoloveanu. Frontiers of silicon-on-insulator. *Journal of Applied Physics*, 93(9):4955–4978, 2003.

[5] chamberlain. Using ld, January 1994. ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_mono/ld.html.

[6] Pai Chou, Ross Ortega, and Gaetano Borriello. Synthesis fo the hardware/software interface in microcontroller-based systems. In *Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, ICCAD '92, pages 488–495, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[7] Capers Jones Christof Ebert. Embedded software: Facts, figures and future. Technical report.

[8] David Czajkowski and Merrick McCartha. Ultra low-power space computer leveraging embedded seu mitigation. In *IEEE Aerospace Conf*, volume 5, pages 2315–2328, 2003.

[9] P. Darling. Intel to Invest More than 5 Billion to Build New Factory in Arizona, October 2013. newsroom.intel.com/community/intel_newsroom/blog/2011/02/18/intel-to-invest-more-than-5-billion-to-build-new-factory-in-arizona.

[10] Bradley F. Dutton and Charles E. Stroud. Single Event Upset Detection and Correction in Virtex-4 and Virtex-5 FPGAs. In Wei Li 0025, editor, *CATA*, pages 57–62. ISCA, 09.

[11] G. W. Eidson, S. T. Esswein, J. B. Gemmill, Jason O. Hallstrom, T. R. Howard, J. K. Lawrence, Christopher J. Post, C. B. Sawyer, Kuang-C. Wang, and D. L. White. The south carolina digital watershed: End-to-end support for real-time management of water resources. *IJDSN*, pages – 1–1, 2010.

[12] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88. USENIX Association, 2006.

[13] Faith Fich, Danny Hendler, and Nir Shavit. On the inherent weakness of conditional synchronization primitives. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 80–87, New York, NY, USA, 2004. ACM.

[14] Free Software Foundation. Wh2004a/b, April 2014. ccrma.stanford.edu/planetccrma/man/man1/avr-size.1.html.

[15] FreeBSD, 2014. www.freebsd.org.

[16] FTDI. ft232r, 2013. www.ftdichip.com/Products/ICs/FT232R.htm.

[17] FutureGrid, June 2013. https://portal.futuregrid.org/.

[18] Joe Geluso. CRC16-CCITT, January 2014. srecord.sourceforge.net/crc16-ccitt.html.

[19] Sven Goldt, Sven van der Meer, Scott Burkett, and Matt Welsh. The Linux Programmers Guide. *Linux Documentation Project (ftp://sunsite. unc. edu/pub/Linux/docs/linux-doc-project/programmers-guide/llpg-0.4. tar. gz)*, 1995.

[20] Jan Goyvaerts. Regular expression, 2013. www.regular-expressions.info.

[21] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174. ACM, 2008.

[22] Leon Huang and John J Selman. Watchdog timer, December 2 1986. US Patent 4,627,060.

[23] F. Irom, F.F. Farmanesh, A.H. Johnston, G.M. Swift, and D.G. Millward. Single-event upset in commercial silicon-on-insulator PowerPC microprocessors. *Nuclear Science, IEEE Transactions on*, 49(6):3148–3155, 2002.

[24] Hao Jiang, Jiannan Zhai, S.K. Wahba, B. Mazumder, and J.O. Hallstrom. Fast distributed simulation of sensor networks using optimistic synchronization. In *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on*, pages 1–8, June 2011.

[25] Olaf Landsiedel, Hamad Alizai, and Klaus Wehrle. When timing matters: Enabling time accurate and scalable simulation of sensor network applications. In *The* 7th *International Conference on Information Processing in Sensor Networks*, pages 344–355, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Ulana Legedza and William E. Weihl. Reducing synchronization overhead in parallel simulation. In *The* 10th *Workshop on Parallel and Distributed Simulation*, pages 86–95, Washington, DC, USA, 1996. IEEE Computer Society.

[27] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005.

[28] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: accurate and scalable simulation of entire tinyos applications. In *The* 1st *International Conference on Embedded Networked Sensor Systems*, pages 126–137, New York, NY, USA, 2003. ACM.

[29] Cheng-Hong Li, Alfred J. Park, and Eugen Schenfeld. Analytical performance modeling for null message-based parallel discrete event simulation. In *MASCOTS*, pages 349–358. IEEE, 2011.

[30] Juncao Li, Fei Xie, Thomas Ball, Vladimir Levin, and Con McGravey. Formalizing hardware/-software interface specifications. In *Procceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 143–152, Washington, DC, USA, 2011. IEEE Computer Society.

[31] J. L. Lions. Ariane 5 Flight 501 Failure Report, October 2013. www.ima.umn.edu/~arnold/disasters/ariane5rep.html.

[32] R.E. Lyons and W. Vanderkulk. The Use of Triple-Modular Redundancy to Improve Computer Reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[33] Steve McCanne, 2014. www.isi.edu/nsnam/ns/.

[34] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An idl for hardware programming. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 2–2, Berkeley, CA, USA, 2000. USENIX Association.

[35] METASTUFF. dom4j, April 2014. dom4j.sourceforge.net/.

[36] Nishchay Mhatre and Shravan Aras. A Hybrid Approach to Radiation Fault Tolerance in Small Satellite Applications. In *Proc. 62nd International Astronautical Congress*, 2011.

[37] Valeri Naoumov and Thomas Gross. Simulation of large ad hoc networks. In *Proceedings of the 6th ACM International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems*, MSWIM '03, pages 50–57, New York, NY, USA, 2003. ACM.

[38] NASA. JPL Institutional Coding Standard for the C Programming Language, April 2014. lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf.

[39] Roving Networks. Wifly gx 802.11g super module, April 2014. ww1.microchip.com/downloads/en/DeviceDoc/rn-131-ds-v3.2r.pdf.

[40] Nahmsuk Oh, , et al. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

[41] M. O'Nils and A. Jantsch. Operating system sensitive device driver synthesis from implementation independent protocol specification. In *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pages 562–567, 1999.

[42] MATTIAS O'Nils and AXEL Jantsch. Device driver and dma controller synthesis from hw /sw communication protocol specifications. *Design Automation for Embedded Systems*, 6(2):177–205, 2001.

[43] OshonSoft. AVR SIMULATOR IDE, January 2014. www.oshonsoft.com/avr.html.

[44] Jonathan Polley, Dionysus Blazakis, Jonathan McGee, Daniel Rusk, and John S Baras. Atemu: a fine-grained sensor network simulator. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 145–152. IEEE, 2004.

[45] Adrian Ratter. Automatic device driver synthesis from device specifications.

[46] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 73–86, New York, NY, USA, 2009. ACM.

[47] Venugopal Kolathur Sandeep Pendharkar, 2014. www.design-reuse.com/articles/23316/automated-device-driver-generation-tool.html.

[48] Xiaoxuan She, N. Li, and J. Tong. SEU Tolerant Latch Based on Error Detection. *Nuclear Science, IEEE Transactions on*, 59(1):211–214, 2012.

[49] P.P. Shirvani, N.R. Saxena, and E.J. McCluskey. Software-implemented EDAC protection against SEUs. *Reliability, IEEE Transactions on*, 49(3):273–284, 2001.

[50] sourceware. Linker Scripts, October 2013. sourceware.org/binutils/docs/ld/Scripts.html.

[51] Sparkfun. GM862 Cellular Quad Band Module, March 2014. www.sparkfun.com/products/retired/757.

[52] Oracle Java Technology. Java native interface, April 2014. docs.oracle.com/javase/6/docs/technotes/guides/jni/.

[53] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[54] C.I. Underwood, J.W. Ward, C.S. Dyer, and A.J. Sims. Observations of single-event upsets in non-hardened high-density SRAMs in Sun-synchronous orbit. *Nuclear Science, IEEE Transactions on*, 39(6):1817–1827, 1992.

[55] CI Underwood, JW Ward, CS Dyer, and AJ Sims. Observations of single-event upsets in non-hardened high-density srams in sun-synchronous orbit. *Nuclear Science, IEEE Transactions on*, 39(6):1817–1827, 1992.

[56] vdcresearch. Embedded processors top 10 billion units in 2008, Septemer 2013. www.vdcresearch.com/_documents/pressrelease/press-attachment-1503.pdf.

[57] Harry K. Utterback Vincent L. Pisacane. Embedded Software Systems. In *Fundamentals of Space Systems Second Edition*. Oxford University Press, New York, 2005.

[58] HT Weaver, CL Axness, JD McBrayer, JS Browning, JS Fu, A Ochoa, and R Koga. An seu tolerant memory cell derived from fundamental studies of seu mechanisms in sram. *Nuclear Science, IEEE Transactions on*, 34(6):1281–1286, 1987.

[59] Ye Wen. Disens: Scalable distributed sensor network simulation. Technical report, In Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 07, 2005.

[60] Winstar. Wh2004a/b, April 2014. www.winstar.com.tw/products_detail_ov.php?ProID=36.

[61] Emmett Witchel, Josh Cates, and Krste Asanović. *Mondrian memory protection*, volume 36. ACM, 2002.

[62] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for linux using mondriaan memory protection. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 31–44. ACM, 2005.

[63] Qing-Li Zhang, Ming-Yuan Zhu, and Shuo-Ying Chen. Automatic generation of device drivers. *SIGPLAN Not.*, 38(6):60–69, June 2003.