BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

IMPACT OF FAULTS AND ERRORS IN CONTEXT OF EMBEDDED OPERATING SYSTEMS

VLIV PORUCH A CHYB V KONTEXTU VESTAVNÝCH OPERAČNÍCH SYSTÉMŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MAREK DOLEŽEL**
AUTOR PRÁCE

SUPERVISOR **Ing. JOSEF STRNADEL, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Computer Systems (DCSY) Academic year 2019/2020

Bachelor's Thesis Specification

22987

Student: **Doležel Marek**
Programme: Information Technology
Title: **Impact of Faults and Errors in Context of Embedded Operating Systems**
Category: Operating Systems
Assignment:

1. Familiarize yourself with key concepts in the areas of embedded systems and operating systems (OS). Summarize the concepts and create an overview of embedded OSes.
2. Choose suitable instruments for your work: an embedded OS, a platform, etc. Summarize key concepts of the chosen platform, OS kernel/services and principles of creating applications based on the OS and the platform. Pay a special attention to concepts that are crucial in terms of the OS operation, services it offers and its application layer.
3. Classify and characterize faults and errors, summarize the concepts of dependability control mechanisms; discuss OS related faults and errors, their possible causes and effects.
4. Propose and then realize a framework and a set of experiments for evaluating the impact of faults and errors on the operation of the chosen OS, its services and its application layer. First, focus your proposal and realization on isolated faults/errors; then, focus on their combination.
5. Gather and interpret experimental results achieved using your framework, identify the pros and cons of your framework.
6. Based on your experiments, discuss and propose mechanisms for mitigating fault/error effects on the operation of the OS and its application layer.

Recommended literature:

- According to the supervisor's recommendation.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Strnadel Josef, Ing., Ph.D.**
Head of Department: Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work: November 1, 2019
Submission deadline: May 28, 2020
Approval date: March 31, 2020

Abstract

The aim of the thesis is to evaluate the impact of injected faults and errors on the embedded operating system and its application layer. This problem is solved with the proposed fault injection framework. The framework injects faults at run-time, i.e, the faults are injected during the executing of the embedded system. The injection is time triggered, or in other words, the fault is injected when a timer expires. Two fault injection campaigns were executed against μC/OS-II kernel running under Nios II system. Both campaigns were targeting the code of μC/OS-II scheduler. The code of the scheduler was enhanced by control-flow checking mechanism for the second campaign. The results of conducted fault injection campaigns have shown that the variant with the CFCSS technique performed poorly compared to the unxenhanced version.

Abstrakt

Cílem práce je zhodnotit dopad injektovaných chyb a poruch na vestavěný operační systém a jeho aplikační vrstvu. Tento problém je vyřešen navrženým softwarem pro injektaci poruch a chyb. Tento software vkáládá poruchy a chyby za běhu, tj. během provádění vestavěného systému. Injektace je spuštěna časem, jinými slovy, chyba je injektována po uplynutí časovače. Proti μC/OS-II jádru spuštěnému pod Nios II systémem byly provedeny dvě kampaně s injekcí chyb. Obě kampaně cílily na kód plánovače. Kód plánovače byl vylepšen mechanismem kontroly toku v případě druhé kampaně. Výsledky provedených injektačních kampaní ukázaly, že varianta s technikou CFCSS se ve srovnání s nezabezpečenou verzí chovala hůře.

Keywords

operating systems, μC/OS-II , embedded systems, Nios II system, faults, errors, failures, fault injection, fault tolerance techniques, redundancy

Klíčová slova

operační systémy, μC/OS-II , vestavěné systémy, systém Nios II , poruchy a chyby, selhání systému, injektace poruch a chyb, techniky zvyšování spolehlivosti, redundance

Reference

DOLEŽEL, Marek. *Impact of Faults and Errors in Context of Embedded Operating Systems.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Josef Strnadel, Ph.D.

Rozšířený abstrakt

Tato práce se zabývá umělým vkládáním poruch a chyb do vestavěného operačního systému, a jejich vlivem na aplikaci běžící na tomto systému. Teoretická část práce uvádí základní koncepty a principy z oblastí úzce souvisejících s touto problematikou. Jedná se především o koncepty a principy z oblastí operačních a vestavěných systémů. Práce dále vymezuje základní pojmy a techniky z oblasti systémů odolných vůči poruchám. Tyto techniky si kladou za cíl zvýšit spolehlivost počítačových a jiných elektronických systému. Tohoto cíle je dosaženo použitím vhodné formy redundance (zejména hardwarové, softwarové, informační a časové).

Praktická část práce nejprve nastiňuje možná řešení výše uvedené problematiky a popisuje zvolený operační systém μC/OS-II a vývojovou desku DE0-Nano . Operační systém μC/OS-II byl zvolen z důvodu jeho velmi častého použití v odborných publikacích, které se zabývají podobnou problematikou.

Dále práce popisuje implementaci nástroje pro vkládání poruch a chyb do vestavěného operačního systému. Tento nástroj umožňuje uživateli vykonat sadu experimentů, během kterých jsou do uživatelem vybraných částí operačního systému vkládány poruchy a chyby. Existuje více způsobů jak toto vkládání implementačně realizovat. Nejčastěji se jedná o hardwarové a softwarové metody. Navržený injektor používá tzv. run-time injekci, tj. poruchy a chyby jsou do cílového zařízení vkládány za běhu, nikoliv v době kompilace programu. Jednou z dalších vlastností je oddělení samotného injektoru od cílového zařízení to znamená, že poruchy a chyby jsou vkládány řídícím počítačem, který je oddělen od cílového vestavěného systému; toto řešení je realizačně náročnější než například vkládání chyb a poruch v době kompilace nebo zabudování injektoru přímo do cílového zařízení, protože je nutné implementovat komunikační protokol mezi cílovým zařízením a řídícím počítačem. Injektor používá relativně rozšířený RSP (remote serial protocol) protokol, který se ve vestavěných systémech používá k ladění kódu nástrojem GNU GDB nebo jeho grafickou nadstavbou. RSP protokol umožňuje řídícímu počítači přistupovat k obsahu paměti a registrů cílového zařízení. Nad tímto protokolem je vybudován samotný mechanismus injekce, který injektuje poruchy a chyby na základě rozhodnutí tzv. ManažeraInjektacePoruch (angl. FaultManager).

V další části textu následuje popis prostředí ve kterém byly prováděny injektační experimenty. Celkem bylo provedeno 2000 experimentů v rámci jedné demonstrační aplikace. Každá demonstrační aplikace obsahuje neseřazené celočíselné pole o 100 prvcích; toto pole je generováno vždy před spuštěním úlohy řazení. Pro samotné řazení je použito 4 různých řadících algoritmů a to bubble sort, insertion sort, merge sort a selection sort. Každá demonstrační aplikace používá vždy jeden algoritmus.

Celkem byly provedeny dvě injektační sady experimentů do kódu plánovače μC/OS-II `OS_Sched`. V případě první sady nebylo použito žádných ochranných mechanismů. V druhé sadě byla implementována metoda kontroly toku softwarovými signaturami. Ve výsledku se ukázalo, že experimentální výsledky první sady jsou ve všech sledovaných aspektech lepší, než výsledky druhé sady. Například aplikace s řadícím algoritmem bubble sort skončila v cca. 75% instancí korektně seřazeným polem a to v případě sady bez kontroly toku softwarovými signaturami. V případě sady s kontrolou toku si tento stejný algoritmus vedl úspěšně, tj. pole seřazeno, pouze v cca. 57% instancí.

Impact of Faults and Errors in Context of Embedded Operating Systems

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Josef Strnadel.

I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Marek Doležel
June 4, 2020

Acknowledgements

I wish to express my sincere appreciation to my supervisor, Mr. Josef Strnadel, who provided me with valuable guidance and professional advice throughout my work on this project.

Contents

Chapter 1

Introduction

With the growing usage of computer systems in mission and safety critical environments, where a failure could lead to an injury or environmental damage, it is increasingly important to design systems that can operate without a fatal failure in a such harsh environments.

This demand has lead to the development of techniques that help the designers with the development of more reliable systems. The objective of these techniques is to remove or tolerate occurring faults that may lead to system's failure in certain situations. A fault may occur as a result of wear-out, radiation, etc. A fault tolerant system can remain operational despite of presence of faults.

To evaluate the fault tolerance of a system, a fault injection and other techniques have been developed. The fault injection is a deliberate introduction of faults into the system. The fault injection experiments are executed in sequence, i.e, when one experiment is done, the next is about to be executed. Experiments that has been executed or are pending to be executed will be referred to as experiment set or fault injection campaign throughout this text.

The goal of this thesis is to design a fault injection campaign, and then evaluate the impact of the injected faults on the operating system and the workload. This task can be accomplished in several ways; the main fault injection techniques are introduced in Chapter 3.

The easiest way is to modify the operating system and embedded the fault injector directly into the kernel. This approach, however, has its drawbacks; one of them is that the operating system no longer corresponds to the version that is to be deployed in the field, and thus the produced results may be affected by the modifications.

More demanding approach has been selected as it provides some advantages over the injector embedded inside the kernel. One of the advantages is a separation of the environment, which is controlling the entire fault injection campaign and injecting the faults and errors, from the target system. Another advantage is the possibility of adding support for more target systems.

A run-time fault injection framework has been created (implementation of this framework is described in Chapter 4) as a result of demanding requirements. Currently, the framework is capable of executing multiple fault injection experiments targeted against global C „objects“, such as variables, structures, and functions.

The above mentioned framework was used to execute several fault injection campaigns against the scheduler of the μC/OS-II kernel with four sorting algorithms as a workload. The workload consists only of one sorting algorithm. The framework iterates over all available workloads. The experimental setup and the results are presented in Chapter 5.

Chapter 2 reviews crucial concepts, principles, and terminology from three distinct areas – operating systems, embedded systems, dependability impairments and common fault tolerance techniques.

Chapter 2

Literature Review

This chapter covers key concepts of embedded and operating systems, and provides an introduction to dependability impairments (faults, errors, failures) and an overview of fault mitigations.

Section 2.1 focuses on the area of embedded systems. First, key concepts, such as microprocessor, and micro-controller, etc., are informally defined, then, core components of an embedded system are described in more detail.

In the next section (Section 2.2), key concepts revolving around operating systems are introduced with the emphasis on process and memory management.

Dependability impairments (faults, errors, and failures) are introduced in Section 2.3.

In the last section (Section 2.4), a selection of different fault tolerance techniques is described. These techniques utilize redundancy that would be unnecessary in fault-free environment to minimize impact of potential faults.

2.1 Embedded Systems

This section is derived from [11, 17].

An embedded system is a computer system that is configured to perform a specific task as a part of more complex system. In contrast to general purpose computers, a higher demands are placed on embedded systems in terms of reliability and safety. This is because embedded systems are used in mission-critical applications, where a failure could have fatal consequences.

These systems can be decomposed into individual components. The most important components include processor, memory, and input and output ports. Figure 2.1 shows a simplified structure of such a system. It can be seen that this system consists of five components, each of which is connected to address, data, and control bus. An address bus is used for addressing devices such as memory, a data bus transfers the actual data, and a control bus is used to send various control signals.

The processor fetches instructions from memory and executes them. This process of instruction execution can be roughly divided into four sequential stages. In this scheme, the processor must finish all of the stages before it can fetch the next instruction.

It is the job of the memory to store instructions and data, or any kind of information. More than one type of memory might be available to a programmer of an embedded system. For example, the firmware might be stored on read only memory (ROM), where as the

instructions and data might be sitting in static RAM. The most commonly used types of semiconductor memories in contemporary systems are briefly characterized in 2.1.2.

Communication with the outside world is handled through ports. A system might use an output port to communicate a change of internal state with the outside world. An input port might be used a user or other system to request certain action from the embedded system.

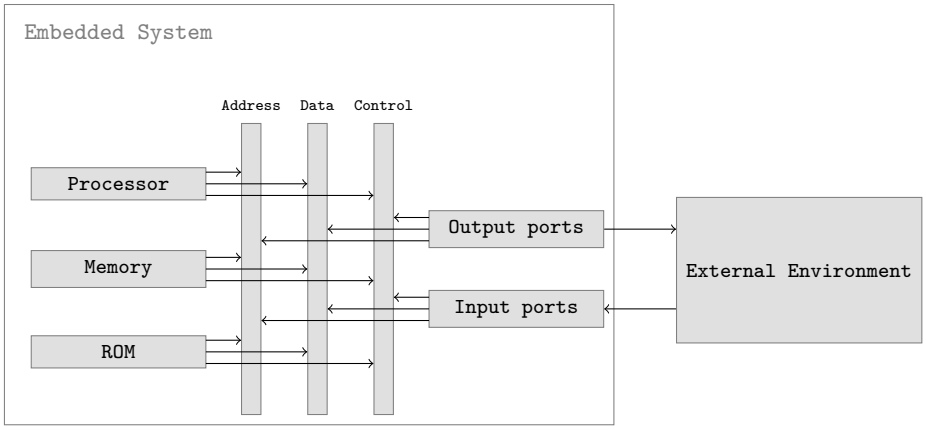


Figure (2.1) Simplified block diagram of an embedded system (Derived from [17])

2.1.1 Key Concepts

Number Representation

Computers store information in a binary form. Elementary unit of information that can be stored is a bit. The value stored in the bit is determined by the presence or absence of a voltage. This voltage is interpreted either as log. 0 (true) or log. 1 (false).

In the positive logic, log. 1 is represented by presence of voltage and log. 0 by absence of voltage. The opposite applies to negative logic, i.e, log. 1 is represented by absence of voltage and log. 0 is represented by presence of voltage.

This binary representation can only represent unsigned numbers (positive numbers). There are numerous ways of representing negative numbers. The most commonly used method for representation of signed integers is 2's complement.

This method can represent n-bit integer with values within $[-2^{n-1}, 2^{n-1} - 1]$. The positive numbers in this representation start with a leading zero and can be converted to decimal representation without a hassle. The negative numbers start with a one; they cannot be directly converted to decimal representation. First, a simple steps has to take place: 1) invert all bits including the leading zeros, and 2) add one to the number.

Micro-processor

A micro-processor or simply processor is an electronic device integrated into a single plastic package, which can perform basic arithmetic-logic operations such as addition, subtraction, division, etc. General purpose processors that are in use today are very complex devices utilizing advanced techniques, such as pipe-lining and branch prediction.

Micro-controller

A micro-controller or controller for short is a device, which integrates multiple components into a single package - processor, peripherals, memory, etc. Most micro-controllers are tailored to specific application. Small 8-bit micro-controllers are very limited in their capabilities, i.e., memory management unit and cache are not present, pipeline is simplified.

Instruction

An instruction is a sequence of bytes in memory executed by the processor; it consists of the operation code and operands. The exact binary values that represent a particular instruction might differ between different families of processors.

Instructions can be classified into several groups: arithmetic-logic, data transfer, program flow, and control instructions. Instructions available to the processor are referred to as instruction set of the processor.

Interrupt

An interrupt is a mechanism by which processor can respond to external or internal event that occurred and needs to be addressed. Different events are handled by their corresponding interrupt handlers. Interrupt handler is a routine with an entry point. Interrupt vector table stores addresses and other information of multiple handlers.

2.1.2 Components of an Embedded System

As noted in the introduction to this chapter, the embedded systems contain at least a processor, a memory for the data, a read-only memory for the instructions, and an I/O port. This section provides a few paragraph descriptions of these components.

Processor

The processor is an integral part of any embedded system; it consists of a data path, and a control unit.

The data path consists of an arithmetic logic unit (ALU), which performs all the arithmetic-logic operations (e.g., addition, subtraction, logical and, logical or, etc.), and a register file, which contains several registers and is used as a temporary storage for operands manipulated by instructions and for computed results.

The register file consists of several general purpose registers and special registers. Some of the most important special use registers are:

- Program counter (PC) - stores memory address of the next instruction to be executed.
- Instruction register (IR) - stores content of memory location pointed to by the program counter (i.e, an instruction to be executed next).
- Stack pointer (SP) - points to the top of the stack, or to the next free memory address.
- Status register - stores flags about the computed result. This flags can be used by control flow instructions to branch the execution.
  - Zero flag - the result is zero.

  - Negative flag - the result is a negative number.
  - Overflow flag - the result has overflown.
  - Carry flag - the operation produced a carry bit.

The control unit instruments other parts of the processor by setting appropriate signals. The instruction execution can be divided into the following four stages:

- *Fetch* - load the program instruction from the memory location pointed to by program counter to instruction register.
- *Decode* - an instruction is decoded by the control unit.
- *Execute* - set the data path by setting appropriate signals.
- *Write back* - the result is written back to the memory or register.

Memory

A memory is used to store data or program instructions for longer periods of time than the register file in the processor. From a programmers perspective, the memory can be classified into data and instruction memory.

Manufactures use a different classification based on the physical properties of the memory. These properties divide semiconductor memories into two basic types:

- *Volatile memory* - stores its content only when the power source is present. This type of memory can be further divided into two types:
  - *Static RAM* (SRAM) - is often being used as on-chip memory in micro-controllers for its low access time in ranges of nanoseconds.

    SRAM consists of array of cells, each cell contain 6 transistors and is capable of storing 1-bit of information. Due to large number of transistors needed per bit of information SRAM is very expensive and takes up a lot of on-chip space.
  - *Dynamic RAM* - the disadvantages (space requirements, price) of SRAM lead to the development of DRAM.

    This type of memory only needs one transistor and one capacitor to store 1-bit, and as a result of this DRAM takes less on-chip space. Due to the leakage current the capacitor requires to be recharged every 10 to 100 milliseconds. This problem is solved by accessing DRAM every few milliseconds. Certain models feature a built-in refresh cycle controller, which recharges the DRAM when it is not accessed by the processor.
- *Non-volatile memory* - retains the stored information even after a loss of a power source; thus it can be used as a permanent storage. There are several types of non-volatile memories:
  - *Read Only Memory* (ROM) - is one of the first type of non-volatile semiconductor memory. The data are imprinted into the memory chip by the manufacturer and cannot be changed later. This is only suitable high volume manufacturing (tens of thousands of units).

- *Programmable Read Only Memory* (PROM) - is composed of matrix of cells, where each cell contains a fuse. Initially, when a fuse is intact, a cell reads one. Zero can be written to a cell by applying a high current pulse. This process will blow the fuse.
- *Erasable Programmable Read Only Memory* (EPROM) - consists of field effect transistors (FETs) used to store the information. The information is stored by electrons in floating gates of FETs. These gates are insulated from the rest of the circuit and by applying an appropriate high voltage electrons are injected into the gate. Due to non-perfect insulators, the gate looses electrons over time. After a sufficient amount of electrons is lost, the cell becomes un-programmed. The EPROM has a window that allows UV light to hit the chip and erase its content.
- *Electrically Erasable and Programmable ROM* (EEPROM) - is similar in how it works to EPROM, but without the need for UV light source for erasing; it is done electrically by applying appropriately high voltage. EEPROM have a limited amount of rewrite cycles (in order of 100.000 cycles). They are used in micro-controller as a long term storage.
- *Flash* - is a special type of EEPROM memory, where only larger blocks can be erased, i.e, a byte cannot be erased without erasing the entire block the byte is part of. This limitation leads to simplified logic, and thus reduced price. Flash is suitable for storage of program and not suitable for data storage.
- *Non-Volatile RAM* (NVRAM) - combines advantages of both the volatile and non-volatile memories. This is can be implemented in several ways: 1) combination of SRAM and internal battery, or 2) combination of SRAM and EEPROM.

2.2 Operating Systems

This section is derived from [16], [15].

In this section, the term operating system is defined in the introduction, and key concepts and principles of the operating systems are briefly introduced.

Several definitions of the term operating system exist. An operating system can be defined as a computer software, which provides abstractions to application software and manages resources of the computer system [16].

Different real-world applications have distinct requirements, for example, a mobile phone will need to meet a different set of constraints than a high performance cluster or an embedded system. As a result of this, a variety of operating systems are being designed and developed. For example, embedded operating systems (see Section 2.2.2), desktop operating systems, which are dominated by Microsoft and Apple with Windows NT and MacOS respectively.

2.2.1 Key Concepts

Abstractions

An abstraction is a process of hiding non-essential details of a system, which makes the task at hand less demanding and less complex. The operating system should implement some of the essential abstractions, such as process, thread, and file, etc., to hide unnecessary implementation details and quicks of a specific hardware.

Resources

As described in previous section (2.1) an embedded system contains processors, memory, peripherals, and other components. All of these components are a physical resources of the computer system. It is a job of the operating system to share, i.e., multiplex, these resources between competing processes. Two multiplexing schemes can be utilized; time multiplexing, and space multiplexing. Time-multiplexed resource can be used by only one process at a time, e.g. processor. On the contrary, space-multiplexed resource may be used by multiple processes at the same time, however each participating process only gets a portion of the resource, e.g. memory.

System Call

A system call is a mechanism by which the kernel provides services to the application software.

System calls can be grouped into several categories: process control, file management, communications, etc. Process control category includes system calls responsible for process creation and process termination. Creation of a file, deletion of a file, opening and closing a file – are part of a file management group. Communications group might include system calls for sending and receiving messages, creation and deletion of a communication connection.

Kernel

A kernel is a core of an operating system, which is responsible for the management of hardware resources, and provision of services to the application software. This core is separated from the user processes by a system call interface. In addition to that, the kernel is running the kernel space, which is separate from the application software, i.e., processes.

Kernels can be classified by structure into several types: monolithic kernels, micro-kernels, and exokernels.

A brief description of these types follows in the next few paragraphs.

Monolithic kernel This type of kernel has a monolithic structure, meaning that subsystems, such as process scheduler, I/O subsystem, memory management, filesystem, etc., are implemented as part of the kernel.

From an implementation point of view, a monolithic kernel is one binary blob with a set of functions that can call each other without any restrictions. This approach can lead to reliability and security issues, but no decrease in performance has to be paid because all of the important subsystems are implemented in the kernel itself.

Micro-kernel A micro-kernel implements only necessary mechanisms in the kernel and the rest is implemented by well-defined modules in the user space. The reasoning behind this decision is that the bugs in the kernel space can cause a system failure. And by moving some of the functionality outside of the kernel, the reliability of the system will increase.

The user space modules, also known as servers, are running as a separate processes. For example, scheduler, filesystem, audio server, are a candidates that would be implemented as modules in micro-kernel.

This modularization of a kernel allows for manageable recovery from failure of individual server. When a server fails, it is simply restarted. This design increases number of context switches, thus leading to decrease in performance.

At present, the desktop operating systems do not use micro-kernel. Nonetheless, they are widely used in real-time, industrial, avionics, and military applications [16].

Kernel mode The kernel is running in the kernel mode separated from running processes, which are running in the user mode. This separation protects the kernel from potentially malicious application software. The application may request a service from the kernel only through well-defined interface, i.e., system calls.

When running in kernel mode, the CPU is allowed to execute privileged instructions (e.g., I/O operation instructions, memory protection instructions). In contrast, a user process is restricted to non-privileged instructions.

These modes are implemented in the CPU. The mode is determined by a bit or several bits in the CPU register depending on number of modes.

Processes

A process is a representation of a running program. Only one process is running on single core CPU at a time.

A time for which a process may run is called *quantum*. When this quantum expires, the operating system decides, which process to run next. *Context switch* is a mechanism by which an OS switches from process to another one. It is up to a scheduler to decide which process to run next. In a time interval of one second, many processes have been run and stopped. This allows for pseudo-concurrent execution.

Every process has its own address space for code, data, stack, and heap; virtual CPU with program counter and registers; global variables. All information required for the execution of a process is stored in an array, where each entry corresponds to single process.

Throughout the life of a process, it can be in one of several states (e.g. running, ready, blocked [16]). A running process is currently being executed by the CPU. A blocked process is waiting for an I/O operation to complete. A ready process is waiting to be scheduled. Different operating systems have a different sets of process states.

Threads

Threads extend the process abstraction with the ability of having multiple points of execution within a process. As shown in figure 2.2, all threads within a process share the same address space, open files, child processes, and more; each thread has its own internal state, program counter, registers, and a stack.

Items shared with all threads	Per-thread items
Address space	Program counter
Open files	Registers
Child processes	Stack
Pending alarms	State

Figure (2.2) Comparison of shared items between threads and per-thread items

Shared address space makes it possible to share data between threads within a process. This would require some form of interprocess communication with using just single threaded processes. With the former programming model synchronization issues arise when more than one threads try to read or write shared data and at least one of the threads is writing to shared data.

Process Synchronization

When more than one process or thread want to access same shared data, this may lead to inconsistency of data when no additional measures are taken.

A race condition is a situation, where at least two processes want to access (read/write) same shared resource (data), and the result depends on particular order of execution.

A critical section (CS) is a segment of code, where a process may access shared resources. The above-named section is preceded by entry section, which implements entry mechanism to the critical section. A critical section is followed by an exit section, and a reminder section. The former indicates, that process is exiting critical section, and therefore some other process may enter.

The implementation of a critical section must satisfy the following requrements (simplified):

- *Mutual exclusion* - only one process is executing critical section at a time.
- *Progress* - a progress is made in each of the processes.
- *Bounded waiting* - no process is waiting to enter forever.

2.2.2 An Overview of Embedded Operating Systems

This section is a brief overview of selected embedded operating systems that are available on the market today; both open-source and proprietary variants are covered.

Manufacturer	Name	License	Kernel type
Wind River Systems	VxWorks	Proprietary	Monolithic
Amazon Web Services	FreeRTOS	MIT	Microkernel
Micrium	μC/OS-II	Permisive	
Apache Foundation	mynewt	Apache License 2.0	?

Figure (2.3) Comparison of various embedded operating systems

VxWorks

Wind River offers a real-time operating system that is suited for hard real-time applications from exploration rovers to medical infusion pumps.

VxWorks[9] features deterministic preemptive priority-based scheduler, broad board support, IPv4 and IPv6 stacks with TSN[1] capability, support for modules, fault tolerant filesystem, and more.

Development and debugging tools based on eclipse are used during the design, development, testing, and debugging phases of the application software. Also, a system analysis tools are available to accommodate for application optimization.

FreeRTOS

FreeRTOS[6] is a real-time kernel with a preemptive scheduler, and a wide range of supported families of microcontrollers from various vendors, available free of charge under the MIT license (owned and managed by Amazon Web Services, Inc).

[1] time-sensitive networking

Optionally, commercial variants (openRTOS, safeRTOS) are available if needed. SafeRTOS meets several international safety standards, and should be used in environments with strict safety requirements. OpenRTOS is a variant of FreeRTOS with technical and development support.

μC/OS-II

μC/OS-II [13] is a preemptive, real-time kernel available for free evaluation from Micrium's website[2]. A commercial license is required for commercial applications, licensing fees are stated upon inquiry.

The kernel is capable of running on a wide range families of processors and microcontrollers. Support for task management, synchronization primitives (semaphores, mutexes, etc.), dynamic memory allocation, timer management, etc. is implemented in kernel source tree.

Additional features, such as IP stack and various other communications protocols, are available as a separate products. The same applies to permanent storage, display device, and ISO/OSI application layer protocols (DHCP[3], HTTP[4]).

Mynewt

Apache Mynewt[7] is a community driven, real-time kernel for interconnected IoT devices, namely, wristbands, wearables, lightbulbs, locks, which must operate with a limited hardware requirements (power, memory, and storage).

Some of the key features include remote firmware updates, signed firmware images, bluetooth 5 stack, WiFi support, and HAL[5] layer for several microcontrollers.

2.3 Dependability Impairments

This section is derived from [10].

In this section, dependability impairments, i.e., faults, errors, and failures are defined and characterized.

The dependability impairments are threats to dependability of the computer systems, i.e., they are the reasons for which the system may not deliver its intended service. There are various types of mitigation techniques, which can help improve the dependability of a computer system; fault prevention and fault tolerance are one of those. Hardware faults are typically caused by physical defects or environmental factors. On the other hand, software faults are usually induced during the design phase as a result of programmer's mistake or misunderstanding of the specification.

Dependability impairments can be informally defined as following:

- *A fault is a physical defect, imperfection, or flaw that occurs in some hardware or software component* [10].
- *An error is a deviation from correctness or accuracy in computation, which occurs as a result of a fault* [10].

[2] μC/OS-II website - www.micrium.com
[3] DHCP - Dynamic Host Configuration Protocol
[4] Hypertext Transfer Protocol
[5] HAL - Hardware Abstraction Layer

- *A failure is a non-performance of some action which is due or expected* [10].

There is a casual relation among dependability impairments, that is a fault can cause an error, and an error can cause a failure.

An active fault is a fault that causes an error, otherwise, the fault is dormant. A single fault may cause multiple errors.

2.3.1 Hardware Faults

Faults occurring in a hardware can be classified by duration into three types:

- *Permanent faults* - are caused by a physical defect (e.g., short circuit, broken interconnection, stuck bit in memory). This type of faults can be removed only by some corrective measure, e.g., replacement of a faulty module or component.
- *Transient faults*, also known as glitches - are caused by environmental factors, e.g., α-particles, cosmic rays, overheating, electric spike, mechanical shock, etc. These types of faults are present in the system for a short amount of time, thus are hard to detect.
- *Intermittent faults* - are transient faults that appear and disappear periodically.

The faults that occur in a real hardware cannot be easily enumerated, therefore various fault models, which try describe consequences of occurring faults, have been developed. Common fault models are:

- *A stuck-at fault* - is the most commonly used fault model. As a result of this type of fault a line in the circuit or a memory cell is permanently stuck at logic 0 or 1 state.
- *A transition fault* - is a fault, where a line in the circuit or a memory cell cannot transition from one particular state to other state.
- *A coupling fault* - is a fault that relates to multiple lines,

2.3.2 Software Faults

Software differs from hardware in that its operation cannot be disrupted by environmental factors, or a wear out. Most of the software faults are introduced into the system during the design phase as a result of programmer mistakes, or misunderstanding of the specification.

2.4 Fault Tolerance Techniques

This section is derived from [10].

Fault tolerance is one of a several means that enable the development of a dependable system, i.e, a system that can deliver its intended service even in a presence of faults.

Fault tolerance is achieved by redundancy that would be unnecessary in a fault-free environment. A redundancy can be classified into space and time redundancy. The former can be further classified into hardware, software, information, and time redundancy. In a hardware redundancy additional hardware components are added to the system. Software redundancy can be achieved by using diverse versions of a software component. Information redundancy can be accomplished by addition of extra bits to the data bits. In a time redundancy computation or transmission is repeated, then the results are compared.

2.4.1 Hardware Redundancy

Hardware redundancy is achieved by providing two or more identical copies of a hardware component. Redundancy can be applied at various levels of abstraction: transistor, gate, flip-flop, or a component level (memory, processor).

Three basic types of hardware redundancy exist: passive redundancy, active redundancy, and hybrid redundancy. Passive redundancy only masks the faults and no actions are performed by the system, or an operator.

Active redundancy does not mask the faults, therefore fault might still cause an erroneous results. The system must perform the following actions before the fault can be tolerated: fault detection, fault location, and fault recovery. After all these actions are taken the faulty component is removed from the system.

The last type of redundancy is a combination of the first two techniques (active and passive redundancy); it not only masks the fault, but also removes the faulty component from the system.

The rest of this section covers these hardware redundancy techniques:

- Passive techniques
  - Triple Modular Redundancy
  - N-modular Redundancy
- Active techniques
  - Duplication With Comparison
  - Standby Sparing
  - Pair-and-a-Spare
- Hybrid techniques
  - Self-purging Redundancy

Triple Modular Redundancy (Passive redundancy)

A system that uses triple modular redundancy consists of triplicated modules and a voter as depicted in Figure 2.4. All three modules (components) perform the same computation in parallel.

The results are then compared by the voter using majority voting method. When one module fails the voter detects it and produces the correct output. Therefore, the fault in one of the components is masked and does not propagate throughout the system. On the contrary, fault in more than one component cannot be masked by the voter and erroneous result is produced, thus fault might propagate, if active, to other parts of the system.

Due to a decrease in reliability below reliability of equivalent simplex system (system with no redundancy) after $0.7\lambda t$, this method is only suitable for applications, which have a short mission time.

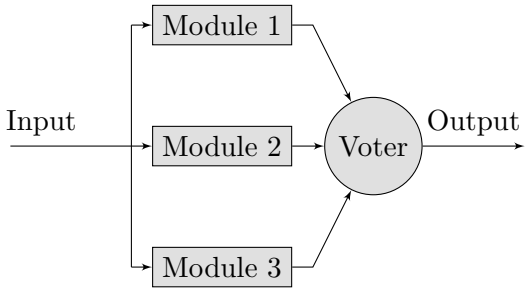


Figure (2.4) Triple Modular Redundancy (Redrawn from [10])

N-modular Redundancy (Passive redundancy)

This technique is works similarly as previously mentioned technique. The only difference is that N-modular redundancy utilizes n, where n is an odd number, modules instead of 3 (as can be seen in figure 2.5).

The voter can mask up to $\lfloor N/2 \rfloor$[6] module faults. When a fault occurs in more modules, the system will produce erroneous result.

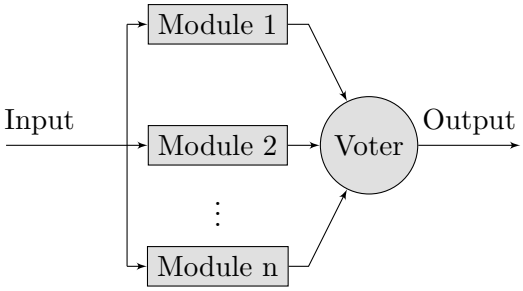


Figure (2.5) N-modular Redundancy (Redrawn from [10])

Duplication With Comparison (Active redundancy)

Duplication with comparison uses two identical modules, which perform the same computation in parallel, and a comparator (Figure 2.6).

Results from the modules are compared by the comparator. When the comparator detects, that the results are not identical, error signal indicating occurrence of fault in one of the components is set. The system will remain in a failure state.

Only fault in one of the modules is detectable by this system. When both modules produce the same erroneous result, the system will not detect that.

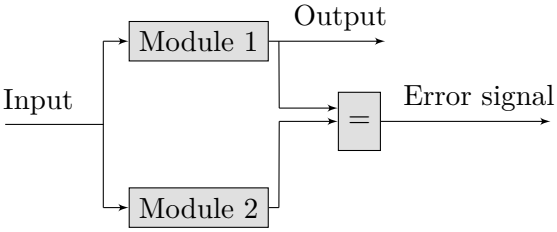


Figure (2.6) Duplication With Comparison (Redrawn from [10])

[6] floor function - $\lfloor x \rfloor$ returns the greatest integer that is less than or equal to x

Standby Sparing (Active redundancy)

A system that utilizes standby sparing consists of n modules, a fault detection unit, and a switch (figure 2.7). Only one of the modules is in operational state (connected to output of the system) and the remaining $n-1$ modules are in a stand-by mode.

Two types of sparing, hot stand-by and cold stand-by, can be differentiated based on whether the spares are powered on or off. In a case of hot stand-by sparing, the spare is switched into use immediately without a significant time delay. The opposite is true in a case of cold stand-by (i.e., spare needs a significant amount of time to finish its power-up cycle and recompute).

A switch is a device that selects one of n inputs and propagates the selected input to its output. When a fault is detected by fault detection unit in active module, the switch switches from active module to a spare.

Stand-by sparing can tolerate, i.e., the system stays operational, $n-1$ faulty modules and detect, i.e, the system is no longer operational, n faulty modules.

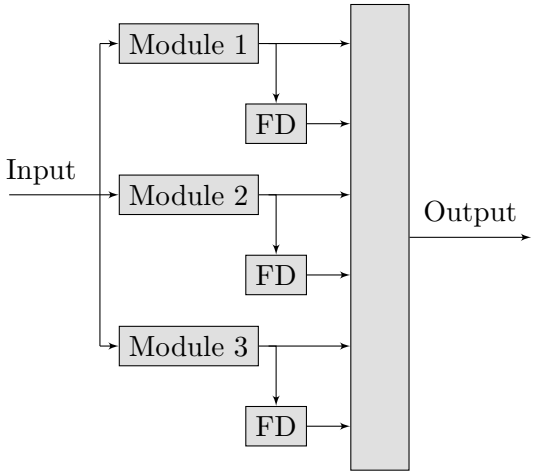


Figure (2.7) Standby Sparing (Redrawn from [10])

Pair-and-a-Spare (Active redundancy)

This technique combines stand-by sparing method and duplication with comparison method. A system consists of n modules out of which only two modules are in operational state, and the rest serves as a spares. Two operational modules are operating in parallel and the results are compared by a comparator. When a disagreement is detected, an error signal is sent to the switch. The switch analysis signals from fault detection units for operational modules, and decides which of the modules is faulty. The faulty module is removed from operation and a spare is switched into use as a replacement.

The above described mechanism can tolerate $n-1$ module faults. When $n-1$th fault occurs only 2 modules are in operational state out of which one is faulty, thus no recovery is possible.

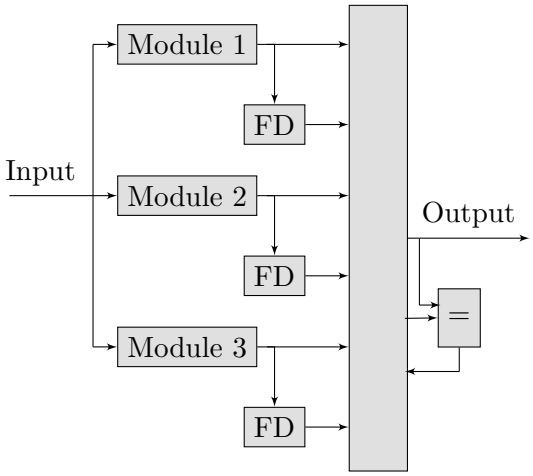


Figure (2.8) Pair-and-a-Spare (Redrawn from [10])

Self-Purging Redundancy (Hybrid redundancy)

This hybrid scheme (shown in Figure 2.9) consists of identical n modules, a switch for each of the n modules, and a voter.

Initially, all modules are active and their outputs contribute to the output of the voter. The output of each module is compared to the output of the voter. When a disagreement between the voter and a particular module's output is detected; it is purged (removed) from the system by a switch that corresponds to faulty module.

The voter can adapt to changing number of inputs as some of the modules gradually fail throughout the time.

This hybrid technique can mask $n-2$ module faults, where n is number of modules in the system. When $n-2$ modules out of n modules have failed, the system will be able to detect the next fault. This time, however, no consensus can be made as only one module out of the two active modules is the faulty one.

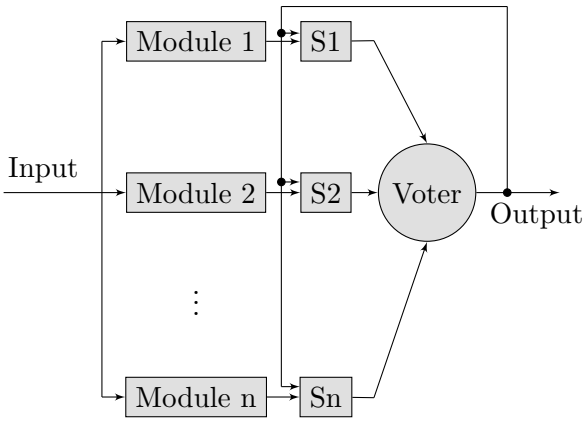


Figure (2.9) Self-Purging Redundancy (Redrawn from [10])

2.4.2 Software Redundancy

As noted in the previous subsection 2.4.1 software faults are not caused by environmental phenomenons, nor physical defects (as software does not wear-out). Most of the faults in software are introduced during the specification or design phase; when the software is

used the faults are already present. Whether the faults lead to a failure depends on the environment that generates the input to the software.

Hardware fault tolerance techniques were designed to mitigate primarily permanent faults, and transient faults secondarily; these techniques provide no protection against design and specification faults that are prevalent in software.

Due to the complexity of large software systems and large amount of states, testing and debugging cannot possibly successes in eliminating all of the faults. Similarly, formal verification cannot be used for the entire software system as it is computationally complex and only applicable to specific applications.

Software fault tolerance methods can be divided into two categories:

- Single version. These techniques do not require to implement multiple versions of a single module. Fault detection, containment, and recovery mechanisms are used instead.
- Multi-version. Redundant software modules need to be developed.

The rest of this section is an overview of a selection of single and multi version techniques.

Single Version Techniques

- *Fault detection techniques* - are able to detect whether a fault has occurred in a software system; this is done using acceptance tests. The test relies on evaluation of the result of a program. When the test passes, the system continues its execution. On the other hand, when the test fails, a fault is present within system.
  - Timing checks - are suitable for systems with known timing constraints. The timing checks are able to detect if the predefined timing constraints are met.
  - Coding checks - are applicable to systems whose data can be encoded
  - Reasonableness checks - use semantic properties of program's data. To perform the reasonableness check, a set of reasonable values must be defined for particular data.
  - Structural checks - rely on structural redundancy in data structures.
- *Fault containment techniques* - can be implemented by changing the structure of the system and by restricting actions that can be performed within the system.
  - Modularization - is achieved by dividing the system into separate modules. The communication between modules is limited and use of shared resources is eliminated.
  - partitioning
  - System closure - uses restrictive approach in which no action is not permitted unless explicitly authorized.
  - Atomic actions - are used by an alternative fault containment technique. This technique relies on an atomic action, which is an interaction between a group of modules. Within an atomic action, there is absolutely no interaction between a module from the group and rest of the system. If a fault is detected the atomic action is aborted; otherwise it terminates normally and results are correct. When a fault is detected, it must be located within the interacting group of modules. Thus, a fault recovery is only applied to participating modules.

- *Fault recovery techniques* - provide the system with the ability to recover from a faulty state. Once the system has recovered it is operational. Four main recovery techniques are described below:

  - Exception handling. The normal execution is interrupted by an exception as a reaction to abnormal event. The possible events causing the exception:
    * Interface exceptions
    * Local exceptions
    * Failure exceptions
  - Checkpoint and restart. This techniques relies on a fact that most software faults are design faults, which are activated by some input sequence. The system can recover from these faults by a restart of the module containing the fault. Figure 2.10 shows components of such a system. The operation is as follows: the program is executing and the output result is checked by acceptance test block; if the result is incorrect the system state is restored from the last checkpoint in memory, otherwise the execution continues.

    Checkpoints can be divided into two groups: static and dynamic. The static checkpoint is only created before the system starts execution of the program, while dynamic checkpoint is created at various times throughout the execution.

    The problem with this technique is that some actions are not recoverable. In case of non-recoverable action, consequences must be compensated, or output must delayed to allow the acceptance test block to determine whether the result is correct or not.

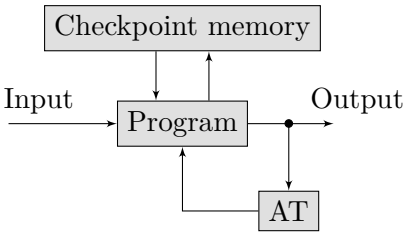


Figure (2.10) Checkpoint and Restart (Redrawn from [10])

  - Process pairs. This technique utilizes two separate processors. Initially, the first processor is active. The active processor executes the program and sends checkpoint information to the other processor. When the active processor detects a fault, it stops executing the program and starts executing offline diagnostics. While the one processor executes diagnostic checks, the second processor loads the last checkpoint and starts executing the program.

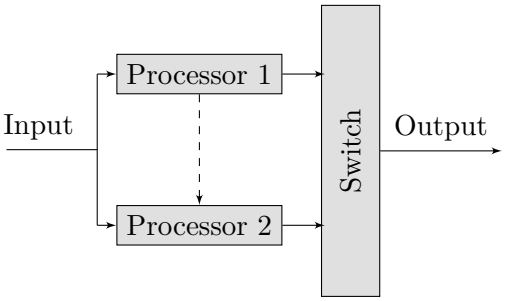


Figure (2.11) Process Pairs (Redrawn from [10])

Multi-version Techniques

- *N-version programming* - is technique that is similar to N-modular redundancy method that utilizes n identical modules. This software technique, however, must use modules that are identical only in terms of produced results; each module must be implemented in different way. The modules are executing concurrently and the results are then fed to selection algorithm module that decides, which result is the correct one and returns the selected result as correct one. The block diagram of such a system is shown in Figure 2.12.

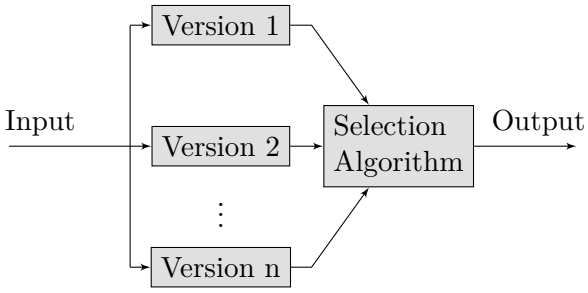


Figure (2.12) N-version programming (Redrawn from [10])

2.4.3 Information Redundancy

Information redundancy is a method of protecting data against errors that can occur, for example, during a transmission of the data over an unreliable communication link, or even when the data are stored on some type of storage device.

In general, information redundancy is most commonly achieved by different coding techniques, which add additional check bits to the data. These extra bits allow for detection of an error, or even correction in some cases. The rest of this section briefly describes a selection of coding techniques.

- *Parity codes.* A parity code of length n consists of $n-1$ data bits and one parity bit at the last position. There are two types of parity codes: even and odd. Even (odd) parity code has even (odd) number of 1s including the parity bit. Any single bit error can be detected, because the parity of the entire code will not match the parity bit.

  Parity codes are frequently used in computer memories, where about 98 % of occurring errors are single bit errors.

- *Unordered codes.* Unordered codes have the ability to detect unidirectional errors. A unidirectional error is an error that only flips zeroes to ones or only ones to zeros. An example of such an error is a change from [10110000] to [00010000].

  M-of-n code is a type of cyclic code that consists of n bits out of which exactly m bits are ones. Any k-bit unidirectional error can only change k ones to zeros or k zeros to one, thus changing the amount of ones in the codeword to $m - k$ or $m + k$, respectively, thus allowing for detection.

- *Arithmetic codes.* These types of codes are used to detect errors in arithmetic operations (e.g. addition, multiplication). Before the operation is performed, the operands of the operation are encoded. Operation is performed with the encoded operands. The result of operation is then decoded (this relies on the property of invariance, i.e., $A(b * c) = A(b) * A(c)$, where b and c are operands, and $A()$ represents encodeding).

2.4.4 Time Redundancy

Time redundancy technique performs the same computation or data transmission multiple times. The computed or transmitted result is compared with the stored result. These techniques can detect or correct a fault. When the repetition is done twice, and the results differ, a transient fault is detected. When the repetition is done more than twice, a fault can be corrected.

Alternating Logic

This scheme is used to detect permanent faults in digital data transmissions and in digital circuits. Assume that the data is transmitted over a parallel bus. First, the original data is transmitted at time t_0, then complement of the original data is transmitted at time $t_{0+\Delta}$. After that the receiver checks if the received data are complementary to each other. For example when the receiver first receives a log. 0 and then receives a log. 1 on a particular bus line, the results are complements of each other, and therefore fault is not detected. This scheme can detect permanent stuck-at faults at the bus lines.

Recomputing With Shifted Operands

This scheme was developed to detect faults in arithmetic-logic units with bit-slice architecture.

The REDO scheme operates in two steps. As a first step, the data are fed to the ALU, the computed result is stored in a register. As a second step, data are left shifted before being sent to the ALU, then the computed result is right shifted and compared with the result that is stored in a register. There is no error if the results are the same. Otherwise, a fault is present in a particular slice of the ALU.

Chapter 3

Problem Analysis

A fault injection is a process of artificially injecting faults into hardware or software. The technique is used to experimentally evaluate the dependability of computer systems. Section 3.1 summarizes main software and hardware injection techniques.

The embedded system based on Nios II processor that was selected to be used throughout the fault injection campaign is only briefly described in Section 3.2 as the main interest of this work is injection of fault into parts of the μC/OS-II kernel. (Section 3.3.

3.1 An Overview of Fault Injection Techniques

The information in this section are based on [12].

When designing and implementing a fault injection tool, one has to choose from different fault injection techniques that are available. Two main injection techniques with their advantages and disadvantages are described below:

1. *Hardware-Implemented Fault Injection.* This technique requires external hardware instrument that is used to inject faults into the target system. Two types of hardware-implemented fault injection methods exist:

   - *Hardware-fault injection with contact.* The fault injector is physically connected to the target system. Through this physical connection the fault injector can externally alter voltage or current flowing to the chip. The most common type of injection with contact is pin-level injection. Commonly used techniques by pin-level injection are active probes and socket insertion. The former technique utilizes probes that are attached to the pins of the target device connecting it with the fault injector. Because each probe is connected to only one pin, it is limited to stuck-at faults. The latter technique inserts a socket between the chip and the printed circuit board. The socket technique can inject stuck-at, open, or complex logic faults into the target system.
   - *Hardware-fault injection without contact.* The fault injector is not physically connected to the target system. Instead, an external device produces physical phenomenon, such as heavy-ion radiation, electromagnetic interference, etc. When the external device is placed near the target system it injects faults when charged ion particles generate current inside the system. Faults are also injected when the system is placed near electro-magnetic field. This method is highly

non-deterministic as it is impossible to control when the ion particle will be emitted.

The disadvantage of both of the above mentioned approaches is the requirement of special hardware instrument. Additionally, the target system may be damaged as a result of fault injection.

2. *Software Implemented Fault Injection.* The software implemented techniques require no additional hardware and can target the operating system or application running on the target system. The disadvantage of this method is the inability to inject faults to locations that are not accessible to the software. Software implemented fault injectors can usually only inject faults to memory and registers. Software based methods can into two classes: run-time injections and compile-time injections.

   - *Run-time fault injection.* This technique requires a mechanism that will trigger the injection of fault into the target system. Common triggering mechanisms are:
     - *Time-out.* When a timer expires it triggers injection of the fault. The timer can be hardware or software timer. This type of fault injection does not depend on state of the system, and thus effects of faults are unpredictable. This technique is satisfactory for emulating of transient or intermittent hardware faults. Software trap involves injecting trap instruction to particular location in the target software, when this instruction is executed, the execution is transferred to interrupt handler.
     - *Exception/trap.* Hardware exception or software trap transfer the execution to the fault injector.
     - *Code insertion.* This technique adds additional instructions to the target's software that will inject fault at run-time when executed.
   - *Compile-time fault injection.* The software of the target system is modified before it is loaded and executed. This modification can be done at source code (e.g., changing $<$ to $<=$, etc.) or at assembly code. The advantage of this method is a very simple implementation due to the fact that no fault injection software needs to be implemented.

3.2 The Target System

This section briefly introduces the target system (DE0-Nano development board with Nios II soft-core) that was selected for the experiments.

3.2.1 DE0-Nano Development Board

The DE0-Nano is an FPGA development board from Terasic with the following on-board compoments[8]:

- Cyclone IV FPGA with 153 I/O pins
- two 40-pin headers
- 32MB of SDRAM

- 13-bit A/D converter
- 50 MHz clock oscillator

The manufacturer provides several ready-made examples, that can be used to speed up development. I have selected one of the example projects, which includes all the soft-cores and interconnections required for Nios II processor and μC/OS-II . This allows me to run the experiment further explained in chapter 5. All example projects are available from the manufacturer's website[1].

3.2.2 Nios II Processor

The Nios II processor is a general purpose 32-bit soft-core from Intel featuring[5]:

- reduced instruction set computer architecture (RISC architecture)
- 32-bit instruction set, data path, and address space
- Optional shadow register[2] sets (up to 63 sets)
- 32-bit general purpose registers
- 32 interrupt sources
- Optional memory management unit (MMU)
- Optional memory protection unit (MPU)
- Instruction set architecture compatible with all Nios II processors
- Error correction code support for all Nios II internal RAM blocks

3.3 μC/OS-II Kernel

The μC/OS-II kernel has been selected for the fault injection campaign because of: 1) the operating is used by relevant papers in a similar problem area, and 2) it is natively supported by toolchain provided with the Nios II processor.

Source code of the kernel is divided into processor independent, processor specific, and application specific code. An example of application for μC/OS-II kernel is in Appendix B.1. Every μC/OS-II application consists of a main function, which can request a service from the kernel, and thus create a new task, a timer, or allocate memory etc. When the `main()` calls to `OSStart()` a scheduler starts running, and as a consequence created tasks start being scheduled.

The kernel provides the following services to the application layer:

- Task management
- Time management
- Semaphore management

[1] Example projects are available from https://www.terasic.com.tw/en/ Products > DE boards > DE0-Nano

[2] shadow register sets are used to accelerate context switching

- Memory management
- Message queue management

The rest of this section is concerned with the `OS_Sched()` procedure that implements the scheduler, and thus is one of the most critical parts of the system. An entire code of the scheduler is listed in Listing 3.1. The beginnings of all basic blocks are highlighted in the code as /*BBx/*, where x is a number of a basic block.

```
void OS_Sched (void)
{
#if OS_CRITICAL_METHOD == 3 /* Allocate storage for CPU status register */
    OS_CPU_SR cpu_sr = 0;
#endif

    /* BB1 */
    OS_ENTER_CRITICAL();
    if (OSIntNesting == 0) { /* Schedule only if all ISRs done and ... */
        /* BB2 */
        if (OSLockNesting == 0) { /* ... scheduler is not locked */
            /* BB3 */
            OS_SchedNew();
             /* No Ctx Sw if current task is highest rdy */
            if (OSPrioHighRdy != OSPrioCur) {
                /* BB4 */
                OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
#if OS_TASK_PROFILE_EN > 0
                /* Inc. # of context switches to this task */
                OSTCBHighRdy->OSTCBCtxSwCtr++;
#endif
                OSCtxSwCtr++; /* Increment context switch counter */
                OS_TASK_SW(); /* Perform a context switch */
            }
        }
    }
    /* BB5 */
    OS_EXIT_CRITICAL();
}
```

Listing (3.1) Barebones μC/OS-II application

A basic block can be defined as follows[14]: *A maximal set of ordered instructions in which its execution begins from the first instruction and terminates at the last instruction.* This means that all instructions in a basic block should be executed in sequence, then the flow of program should continue in one of the succeeding basic blocks of the current block; this is usually branching instruction.

The control flow graph of the scheduler is depicted in Figure 3.1. The graph consists of nodes (basic blocks) and directed edges. The paths that connect the nodes are the only valid branches. An example of an invalid branch is B1 to B4.

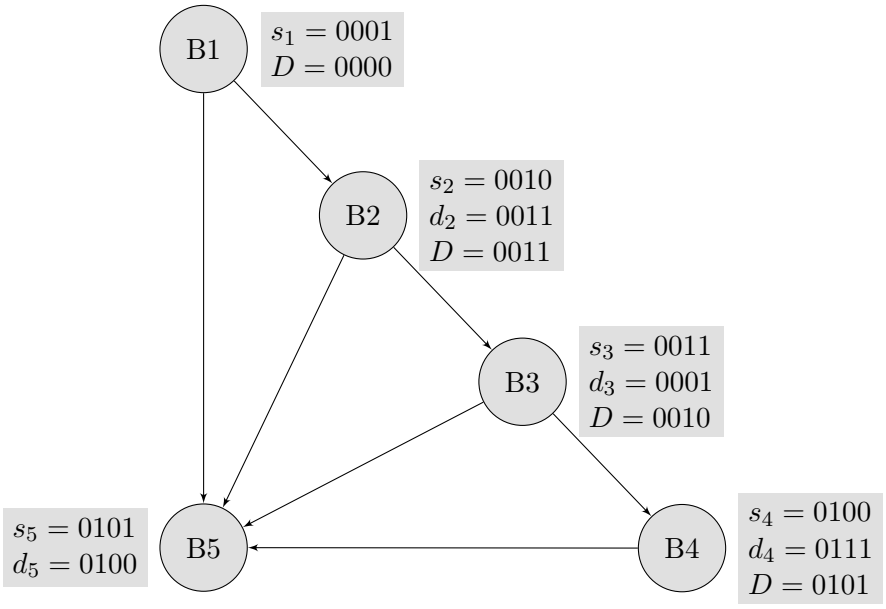


Figure (3.1) Control flow graph of the μC/OS-II scheduler

The scheduler has been enhanced with control flow checking by a software signatures[14]. This method works as follows: 1) a unique signature s_i is assigned to every basic block, 2) a difference is computed as $s_i \oplus s_j$, where s_i is predecessor of s_j. If a block has more than one predecessors a D is introduced.

The implementation uses 2 global variables *OS_SCHED_CFCSS_G* and *OS_SCHED_CFCSS_D*. The first variable represents a current signature of a block. For example, when the execution goes from B2 to B3, this signature is set to 0010. At the beginning of B3 the following operation occurs: $G \oplus d_3$ and because G is 0010 and d_3 is 0001, the result of $0010 \oplus 0001$ is 0011, then s_3 is compared with the result of $G \oplus d_3$. And if s_3 is equal to G, then no control flow error has occurred (this case). When a branch to B5 is taken additional xor operation ($G \oplus D$) is performed after the $G \oplus s$.

Chapter 4

Implementation Details

In this chapter, I introduce proposed fault injection framework and describe some of the more interesting aspects of implementation in detail.

The fault injection framework allows a user to conduct a fault injection campaign, gather experimental results, and evaluate the impact of injected faults on application that is running on embedded system(further referred to as the target system). The framework itself is running on a separate computer (further referred to as the control computer) with GNU/Linux.

The communication between control computer and the target system is handled by existing debugging protocol (remote serial protocol) implemented by the GNU Debugger. This protocol is widely adopted in area of embedded systems as a result of frequent use of the GNU toolchain.

Above mentioned protocol enables the control computer with the ability to manipulate memory and registers of target system. This aspect is used to inject transient faults at run-time into memory or registers. The fault is triggered by time-out.

4.1 An Overview of Execution Stages

The execution of the framework is divided into several stages, each of which must accomplish a specific task before the next stage can run. If an error occurs during the execution of a stage, the execution is stopped and an error code is returned. This could happen if, for example, the target system was accidentally disconnected.

The first three stages are implemented in bash programming language and are related to preparation of the target system, target application, and the environment.

The fault injection stage and memory analysis stages are implemented in C by the core of the framework.

The stages execute exactly in this order:

1. *Hardware programming stage.* Compiled design is loaded into the (FPGA[1]); optionally, this can be done manually if tools provided by vendor are prone to malfunction.

2. *Compilation stage.* The operating system and application for the embedded system are compiled and are ready to run. This stage is executed only once for each application.

[1]FPGA - Field Programmable Gate Array

3. *Environment preparation stage.* The framework launches a background tasks that are delegated with operations related to the target system, such as loading the software, pausing the target processor, gathering output, translating fault injector commands to JTAG[2] commands of the target.

4. *Fault injection stage.* At this stage the target is ready to run, i.e, the application is loaded and the processor is in paused state. This stage further divided into 3 phases:

   - *Phase 1: before a fault injection.* During phase 1 no faults are injected and the target is kept running for a time interval specified by $t_{injectionTime}$ microseconds. Once the time is up, the target is stopped and a transient fault is injected.
   - *Phase 2: fault presence.* This phase executes for interval defined by $t_{fault_duration}$. The framework makes sure that the fault is not overwritten during this phase. Once the time is up, the target is stopped and the next phase is ready to be executed.
   - *Phase 3: after a fault injection.* The target system is kept running for predefined time interval, and the fault is no longer present, i.e, fault in memory location or register can be overwritten.

5. *Memory analysis stage.* Optionally, memory of the target system can be fetched to the control computer and analyzed. For example, the current version of the framework can fetch an array of integers and compute inversion number of the array.

6. *Output analysis stage.* Output from the application is being analyzed. The output from the target is stored in user defined directory.

4.2 Components of the Framework

The framework is composed of several components:

- RunTimeFaultInjector Core - is a single ELF[3] executable that implements the fault injection mechanism and policy. The former is implemented by faultIinjectionMechanism and the latter by FaultManager. The core executes exactly one experiment and prints logs to stdout. The instrumentation is handled by bash script that is introduced below.

- A wrapper bash script (run_fi_campaign.sh) - provides a convenient way to execute a fault injection campaign[4], and gather experimental results. Also, some output (stdout) processing from the embedded system is handled by this script.

- Additional helper scripts - for target device dependent tasks such as programming the FPGA, loading operating system and application binary, etc.

[2] JTAG - Joint Test Action Group
[3] ELF - Executable and Linkable Format
[4] fault injection campaign - is a set of experiments

4.3 RunTimeFaultInjection Core

This section describes decomposition of the RunTimeFaultInjection Core into modules.

As noted above in the introduction, the core handles all the fault injection to the target system and fault injection policy.

A user must specify the experiment parameters such as a memory address and memory length, and parameters a,b of uniform distribution for both the fault duration time and fault injection time.

The core is composed of the following three modules: *FaultManager*, *FaultInjectionMechanism*, and FaultImpactClassifier. A *FaultManager* module implements fault injection policy, i.e., when to inject the fault (fault injection time), for how long (fault duration), type of fault, etc. Low-level fault injection mechanisms are implemented by the FaultInjectionMechanism module; they can be abstracted into three simple phases: fetch memory or register, inject fault, sent back to target.

whereas, *FaultInjectionMechanism* module implements, as a name suggests, fault injection mechanisms for a target system. FaultImpactClassifier module probes the memory of the embedded system to detect impact of the injected fault/faults; it is highly dependent on the operating system and application.

4.3.1 FaultInjectionMechanism Module

Before the framework

This module implements fault injection and execution control of the target system. The functionality is then used by higher-level modules as set of functions.

The fault injection is handled by `fim_inject()` function.

, which takes only one argument with the information required for the injection, such as type of the fault, whether to inject into memory or register, which location to inject the fault (randomly selected bit of certain memory address).

Pseudo-code in Algorithm 1 depicts implementation of `fim_inject()` function without unnecessary C clutter.

First, a memory or a register content is fetched and stored in a temporary data structure (see line 8, and 15). When the injector is fetching memory contents, then only one byte is fetched from the target device to save the unnecessary data transfers. Next, the selected bit is modified directly in the temporary data structure (see lines 9, and 16. Supported operations are a bit flip, a toggle to one, and a toggle to zero. Last, the contents of the temporary data structure are sent back to the target device.

Execution is stopped and resumed by `fim_platform_stop_execution()` and `fim_platform_execute()` respectively. These functions abstract details of backend protocols that are used to communicate with the target device.

Algorithm 1: Abstract overview of FaultInjectionMechanism

precondition : the embedded system is stopped/paused
postcondition: a fault is injected

```
1  Procedure fim_inject(struct fim_injection *injection)
2      local variables;
3      check that injection is not null, otherwise return error;
4      check that injection->device is memory or register, otherwise return error;
5      if injection->device is a memory then
6          check that injection->memory_address is valid, otherwise return error;
7          storage ← storage_create();
8          storage ← get memory content from device;
9          inject fault to storage;
10         send new storage with injected fault to device;
11     end
12     if injection->device is a register then
13         check that injection->register_id is valid, otherwise return error;
14         storage ← storage_create();
15         storage ← get memory content from device;
16         inject fault to storage;
17         send new storage with injected fault to device;
18     end
19     return success
```

Communication Protocol

The control computer and the target system communicate through existing debugging interface present in GDB. No modification to the operating on the target system is necessary. However, a modification is required on the framework side if one wants to add a support for a new target with different communication protocol.

4.3.2 FaultManager Module

The FaultManager is the high-level component that is responsible for preparation of structure representing a fault, and initiating the fault injection at a specified time.

The structure contains the following information about the fault:

- fault type
- fault injection time in [*us*]
- fault duration in [*us*]
- fault location
  - memory - memory address, length, bit index
  - register - register index, bit index

Most of the above mentioned fault parameters are generated from a uniform distribution, except for fault location, which is supplied by the framework as two arguments (memory address and length), or just one argument (register index). The bit index is a single bit within memory, or register, where the fault will occur; it is generated from a uniform distribution.

Injection of Transient Fault

The execution is conceptually divided into three phases. The next paragraph describes the first one.

When the fault parameters are computed, the FaultManager calls EXECUTE() function. This function is implemented by FaultInjectionModule. When the execution is returned back to FaultManager, a packet with execute command has been sent, and as a result of that the target's processor is now executing instructions. Now, the manager calls to `usleep`() with $t_{injectionTime}$ in microseconds as an argument; this function suspends execution of the manager for at least $t_{injectionTime}$ microseconds. When usleep call returns, the execution of the target is stopped and the fault is injected by calling to `fim_inject()`.

In the second phase, the execution is again resumed by call to `EXECUTE()`. After that, both the target and manager are executing concurrently. Next, the manager sets a timer by calling to `setitimer()`. The timer is set to $t_{faultDuration}$. At this point, the manager is running in a loop, waiting for a packet from the target indicating that the injected area was written to. When this packet arrives, the `fim_inject()` function is called to re-inject the fault. This happens until the timer set by `setitimer()` expires. The expired timer triggers a signal that changes condition of the loop to false, and as a result the manager proceeds to stop the target. This ends the second phase.

The last phase lets the target system to run for a specified amount of seconds. This ensures that every run of the target is given a chance to produce correct results and output.

Phase	FaultManager	Target System
Phase 1	EXECUTE() usleep() . . (wake up)	Running
	STOP() fim_inject()	Stopped
Phase 2	EXECUTE() setitimer() (loop: reintroducing transient fault) . (timer expired)	Running
	STOP()	Stopped
Phase 3	EXECUTE() usleep() . . (wake up)	Running
	STOP()	Stopped

Table (4.1) RTFI execution for a transient fault. FaultManager execution and the corresponding target system states are shown.

4.3.3 FaultImpactClassifier Module

The fault impact classifier depends on the target system, operating system, and the application. The idea here is that, when all of the above mentioned modules are done, then the control is passed to this module.

At the present time, the classifier can only scan an array in memory of the target system and compute an inversion number, i.e., check if an array is sorted (inversion number is equal to zero) or if an array is not sorted (inversion number is greater than zero).

Chapter 5

Experiment Setup and Results

In this chapter, the experiment setup used to inject faults/errors into the selected parts of the μC/OS-II kernel is described, and the results of the fault injection campaigns are presented.

5.1 Experiment Setup

The experimental setup consists of the target system, i.e, the embedded system in which the faults are injected and control computer that controls the entire fault injection campaign. The block diagram in Figure 5.1 shows a connection between the above mentioned systems.

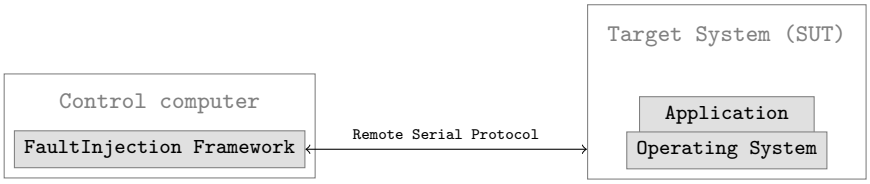


Figure (5.1) The block diagram shows a connection between the target system and control computer. The faults are injected using remote serial protocol.

5.1.1 Fault Parameters

The injected faults have the following parameters:

- **Fault model** - single bit-flip
- **Fault location** - global data structure, procedure, or register
- **Fault injection time** - is a time of fault injection to the target system and is given by $U(a, b)$, where a, b are real numbers in μseconds.
- **Fault duration** - similarly as with injection time, fault duration is given by $U(a, b)$

5.1.2 Operating System and Workload

The software on the target system consists of 1) the μC/OS-II operating system, and 2) the workload.

Each workload implements one sorting algorithm out of a set of selected sorting algorithms. The selected algorithms are: bubble sort[1], insertion sort[3], merge sort[2], and selection sort[4]. The workload consists of two μC/OS-II tasks:

- Generator task - fills an array of 100 integers with random numbers before the sorting task begins; the random numbers are generated with `rand()` function from standard C library. Result of `clock()` function is used as a seed.
- Sorting task - starts sorting the array immediately after its creation. The results are printed to console in format suitable for automatic processing. The sorted array is stored in global variable. One variable is used as a flag to indicate that the sorting task is done with sorting.

5.1.3 Fault Injection Environment

The experiments are controlled, executed, and evaluated from a control computer that is separate from the target system. The communication between the control computer and the target system is handled by existing GDB debugging interface. This arrangement makes fault propagation from the target system to the control computer impossible.

5.1.4 The Control Computer and Software Tools

The control computer used throughout the experiments has the following hardware configuration:

- Intel Core 2 Duo P8600 processor, 4GB of RAM
- Development board DE0-Nano is connected to the USB port of the control computer

The following software is running on the control computer and is involved in experiment preparation, execution, and evaluation:

- Ubuntu 18.04.4 LTS - an operating system from Canonical; provides execution environment for the tools below.
- Quartus Prime Lite 18.1 - an FPGA[1] design software from Intel; used for hardware synthesis and programming of DE0-Nano development board.
- Nios II compiler toolchain and other tools from Intel
  - nios2-elf-gcc - a C compiler for Nios II processor
  - nios2-gdb-server - a bridge between proprietary JTAG[2] interface for debugging purposes. The debugging interface is used by the RunTimeFaultInjector framework to inject faults into the Nios II system.
  - nios2-terminal - a tool for viewing the output from the Nios II system; used by RunTimeFaultInjector framework to store target output for further automatic or manual analysis.
- RunTimeFaultInjector framework - a fault injection framework (described in Chapter 4).

[1] FPGA - Field Programmable Gate Array
[2] JTAG - Joint Test Action Group

5.2 Experimental Results

This section introduces the results of each of the conducted fault injection campaigns; in the first campaign the scheduler was not secured by control flow checking mechanisms, the second campaign was run with enabled CFCSS[14] (Control-Flow Checking by Software Signatures) technique.

At total 8000 of experiments were conducted per campaign. This means that 2000 experiments were run per one workload. Due to the limitations of the FaultInjector framework only one single-bit fault was injected per independent run (experiment).

Two aspects of the workload instance are watched: 1) an inversion number of the array in memory of the target system, and 2) the results presented to the output of terminal. The former aspect is used to measure a sortedness of the array, i.e, a sorted array is given a score of zero and an unsorted array is given a positive score.

The latter aspect classifies the terminal output of an instance into three classes: the output is correct, the output is not correct, the output is not preset. Only output inside cut lines is considered to be relevant output from the workload, everything output cut lines is considered debugging output not relevant to the experiment.

5.2.1 Fault Injection Campaign 1

Transient single-bit fault was injected into code of the μC/OS-II scheduler (`OS_Sched()`) at a time $t_{injectionTime}$ and for a duration $t_{faultDuration}$ given by $U(0, 1000000)$ in microseconds.

The results are presented in Figure 5.2 and 5.3. It can be seen from Figure 5.2 that at least 74.8 % of instances per workload ended with correctly sorted array, and the least amount of incorrectly sorted array presented merge sort workload with 21.4 % of instances with incorrectly sorted array.

Workload	Array is sorted		Array is not sorted	
	#	%	#	%
Bubble sort	1509	75.45	491	24.55
Insertion sort	1496	74.8	504	25.2
Merge sort	1572	78.6	428	21.4
Selection sort	1541	77.05	459	22.95

Figure (5.2) The results of fault injection campaign 1. Each column with a # represents a number of instances that ended with sorted or not sorted array.

Also, the two tables (5.2, 5.3) show that when the array was sorted in memory of the target system, the printed output was either incorrect or there was no output. For instance, in the first workload 1509 instances resulted in sorted array in memory and only 1477 instances of the same workload produced correct output.

Workload	Correct output		Incorrect output		No output	
	#	%	#	%	#	%
Bubble sort	1477	73.85	1	0.05	523	26.15
Insertion sort	1425	71.25	1	0.05	575	28.75
Merge sort	1501	75.05	1	0.05	499	24.95
Selection sort	1465	73.25	1	0.05	535	26.75

Figure (5.3) The results of fault injection campaign 1. Each column with a # represents a number of instances that ended with correct, incorrect, or no output.

5.2.2 Fault Injection Campaign 2

This campaign differers from the previous campaign in Section 5.2.1 only in enabled CFCSS technique for `OS_Sched()` function. The fault parameters are the same.

The results presented in 5.4 are much worse than the results without control flow checking mechanisms in 5.2. Only circa 57% of instances ended up with a sorted array in a case of bubble sort.

Workload	Array is sorted		Array is not sorted	
	#	%	#	%
Bubble sort	1151	57.55	849	42.45
Insertion sort	1231	61.55	769	38.45
Merge sort	1267	63.35	733	36.65
Selection sort	1297	64.85	703	35.15

Figure (5.4) The results of fault injection campaign 2. Each column with a # represents a number of instances that ended with sorted or not sorted array.

Workload	Correct output		Incorrect output		No output	
	#	%	#	%	#	%
Bubble sort	1136	56.8	1	0.05	864	43.2
Insertion sort	1184	59.2	0	0	816	40.8
Merge sort	1159	57.95	1	0.05	841	42.05
Selection sort	1176	58.8	0	0	824	41.2

Figure (5.5) The results of fault injection campaign 2. Each column with a # represents a number of instances that ended with correct, incorrect, or no output.

Chapter 6

Conclusion

The aim of this work was to analyze impact of faults and errors on the operating system and its applications layer. To solve this goal, a new tool has been developed. This tool is capable of injecting faults and errors into the memory of an embedded system. It also supports injection to register, however, this is known to be broken at this time. Unfortunately, the tool proved to be very difficult to implement, then anticipated at the beginning of this project. The main flaw of this software is the inability to classify effects of injected faults on the operating system as it can only classify effects of injected faults on a very specific workload – a sorting workload. The main advantage of the framework is a support of remote serial protocol, which is very common among various embdedded systems. This makes the framework portable to other platforms.

Several fault injection campaigns were conducted, but proved to be invalid due to incorrect fault model. The fault model has been fixed and two sets of experiments were conducted. The first experiment injected the faults into the scheduler, which was not secured by control flow checking methods. The second set was run against the same scheduler only with control flow checking by software signatures enabled. The experimental results have shown that the enhanced scheduler performed much worse than the stock version. Due to the time intensity of one experiment set, no additional campaigns were conducted.

Bibliography

[1] *Bubble Sort* [online]. Virginia Polytechnic Institute and State University [cit. 2020-06-02]. Available at: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/BubbleSort.html.

[2] *Implementing Mergesort* [online]. Virginia Polytechnic Institute and State University [cit. 2020-06-02]. Available at: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/MergesortImpl.html.

[3] *Insertion Sort* [online]. Virginia Polytechnic Institute and State University [cit. 2020-06-02]. Available at: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/InsertionSort.html.

[4] *Selection Sort* [online]. Virginia Polytechnic Institute and State University [cit. 2020-06-02].

[5] *Nios II Processor Reference Guide* [online]. Intel Corporation, 2019 [cit. 2020-24-02]. Available at: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf.

[6] *FreeRTOS website* [online]. Amazon Web Services, Inc., 2020 [cit. 2020-11-04]. Available at: www.freertos.org.

[7] *Mynewt website* [online]. The Apache Software Foundation, 2020 [cit. 2020-11-04]. Available at: www.mynewt.apache.org.

[8] *Terasic DE0-Nano User Manual* [online]. Terasic Technologies Inc., 2020 [cit. 2020-24-02]. Available at: https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=593&FID=75023fa36c9bf8639384f942e65a46f3.

[9] *VxWorks product overview* [online]. Wind River Systems, Inc., 2020 [cit. 2020-11-04]. Available at: https://resources.windriver.com/vxworks/vxworks-product-overview.

[10] Dubrova, E. *Fault-Tolerant Design*. Springer Publishing Company, Incorporated, 2013. ISBN 1461421128, 9781461421122.

[11] Gridling, G. and Weiss, B. *Introduction to microcontrollers*. Vienna University of Technology, 2007.

[12] Hsueh, M. C., Tsai, T. K. and Iyer, R. K. *Fault injection techniques and tools*. 1997. DOI: 10.1109/2.585157. ISSN 00189162.

[13] Labrosse, J. J. *MicroC/OS-II : the real-time kernel*. 2nd ed. CMP Books, 2002. 605 p. ISBN 1578201039.

[14] Oh, N., Shirvani, P. P. and McCluskey, E. J. Control-flow checking by software signatures. *IEEE Transactions on Reliability*. 2002, vol. 51, no. 1, p. 111–122. DOI: 10.1109/24.994926. ISSN 00189529.

[15] Silberschatz, A., Galvin, P. B. and Gagne, G. *Operating system concepts*. Ninth editionth ed. Wiley. ISBN 978-1-118-06333-0.

[16] Tanenbaum, A. S. *Modern operating systems*. 4th ed. Pearson. ISBN 978-0-13-359162-0.

[17] Valvano, J. W. *Embedded Microcomputer Systems: real time interfacing*. 3rd ed. Cengage Learning. ISBN 978-1-111-42625-5. OCLC: ocn654314460.

Appendix A

Contents of the Included Medium

- **Thesis/** - latex source code and PDF version of this text
- **RunTimeFaultInjector/** - the RunTimeFaultInjector framework
- **Experimental_results/** - the results of fault injection campaigns

Appendix B

μC/OS-II snippets

```
#include <stdio.h>
#include "includes.h"

#define TASK_STACKSIZE 2048 /* Size of a stack of particular task */
OS_STK task_stk[TASK_STACKSIZE]; /* Declaration of global variable representing a stack

/* Definition of Task Priorities */
#define TASK_PRIORITY 1

/* Generates array with ARR_SIZE random elements */
void task_fn(void* pdata) {
       while (1) {
              printf("Hello world\n");
              OSTimeDlyHMSM(0, 0, 2, 0);
       }
}

/* The main function creates two task and starts multi-tasking */
int main(void) {
    INT8U perr;

    /* Creates the first task */
    OSTaskCreateExt(task_fn,
        NULL,
        (void *)&task_stk[TASK_STACKSIZE-1],
        TASK_PRIORITY,
        TASK_PRIORITY,
        task_stk,
        TASK_STACKSIZE,
        NULL,
        0);
    OSStart(); /* Starts the scheduler */
}
```

Listing (B.1) Barebones μC/OS-II application

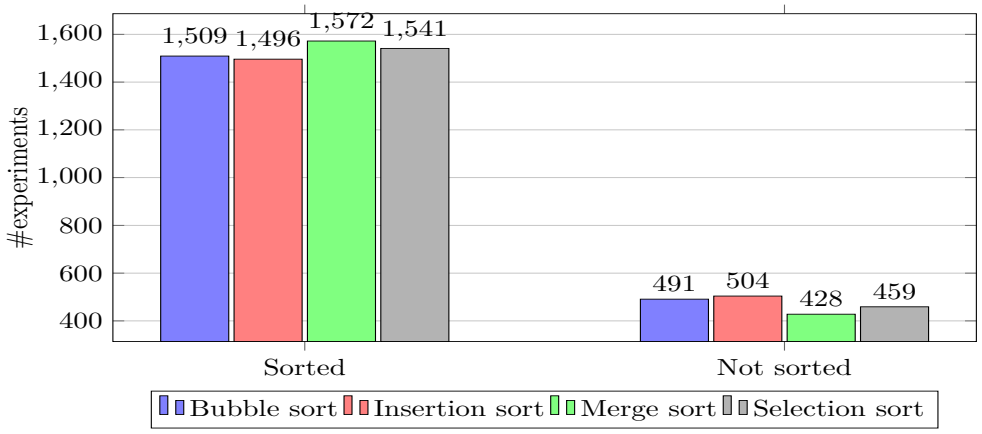
```
/* Global variables definitions */
// ...

/* bubble sort, insertion sort,
merge sort, or selection sort */
void sort_fn_bubble_sort(int A[], int size)
{
   // ...
}

/* Sorting task */
void sorting_task(void* pdata)
{
    struct task_data *tsk_data = (struct task_data * ) pdata;
    int *A = tsk_data->A;
    int size = tsk_data->size;

    PRINT_ARRAY(A, size);

    /* Sorting */
    sort_fn_bubble_sort(A, size);
    demo_state = 1;

    printf("<--\n");
    PRINT_ARRAY(A, size);
    printf("<--\n");
    printf("SORTING_TASK: DONE <ticks 0x%lx>\n", GET_TICKS());

    while(1) {
        printf("SORTING_TASK: DONE\n");
               OSTimeDlyHMSM(0, 0, 2, 0);
    }

}

/* Generates array with ARR_SIZE random elements */
void generator_task(void* pdata)
{
    // 1) generate array of 100 integers, array[i] = rand() with seed clock()
        // 2) create sorting_task

        OSTaskCreateExt(sorting_task, ...);

        while (1)
        {
               printf("GENERATOR_TASK: INFINITE LOOP\n");
               OSTimeDlyHMSM(0, 0, 2, 0);
        }
```

```
}

void timer_callback(void *pdata)
{
    timer_fired_cnt++;
}

/* The main function creates two task and starts multi-tasking */
int main(void)
{
    timer = OSTmrCreate(...);
    OSTaskCreateExt(generator_task, ...);
    OSStart();
    printf("OSStart() has returned <ticks 0x%lx >\n", GET_TICKS());
    while(1) {}

    terminate:
    demo_state = -1;
    printf("An error has occurred, waiting for termination <ticks 0x%lx>\n", GET_TICKS(
    while(1) {}
}
```

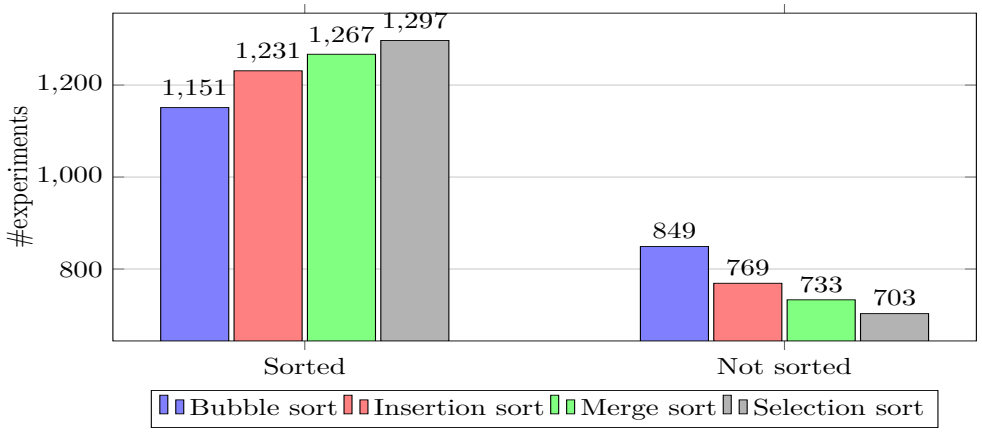
Listing (B.2) Barebones μC/OS-II application

Appendix C

Results of Fault Injection Campaign



(a) This chart shows how many experiments ended with sorted array, i.e., a fault was masked, and how many experiments resulted with unsorted array, i.e., a fault was activated. No CFCSS



(b) This chart shows how many experiments ended with sorted array, i.e., a fault was masked, and how many experiments resulted with unsorted array, i.e., a fault was activated. With CFCSS

Figure (C.1) An application is sorting an array with 100 items. A fault f is injected into `OS_Sched()` procedure at a time $f_{injection\_time}$ and with duration $f_{duration}$. Both the injection time and fault duration are given by $U(0, 1000000)$ in μs. The transient fault model was used. After each experiment, the array is read from target's memory and evaluated.