GEBZE TEKNİK ÜNİVERSİTESİ

ELEKTRONİK MÜHENDİSLİĞİ BÖLÜMÜ

ELEC 334

2020 – 2021 BAHAR DÖNEMİ

HW3

Öğrencinin

Numarası : 1801022071

Adı Soyadı : Alperen Arslan

Embedded C/C++ Unit Testing Basics

Using the CppUTest 3.8 unit test framework, we present a complete real-world example of a unit test. This is the second installment of our series on "Building Better Firmware." It's typical to construct different implementations of modules that make sense for a given unit test when first starting to write unit tests. If you wish to declare each return value of a function, mocks come in handy.

The simplest technique to test every path of the module under test is to use many mocks in a single unit test. A key/value store is used in littlefs to store various values in the kv directory. The value of each key will be written as the file data in the /kv directory. We put this to the test.

CppUTest is one of many C/C++ unit test frameworks, and it was picked because it is one that I am familiar with. We're going to try developing a fake mutex implementation because mutexes are vital in this module and we don't want to forget to unlock one. When I was implementing a difficult raw flash storage to filesystem migration for our firmware four years ago, a coworker proposed unit testing.

High Performance Embedded Computers: Development Process and Management Perspectives

The development process for high-performance embedded computing (HPEC) systems is described in this chapter. The requirements, plans, and implementation decisions of the systems in which they are incorporated influence system development. Front-end, data recorder, and back-end are the three subsystems that make up a typical HPEC system. The development management must be adjusted to the specific technological choices.

Each technology has its own development cycle, price, technical constraints, and hazards. HPEC systems must meet throughput and latency requirements while adhering to stringent size, weight, and power limits. COTS processor software is also a significant source of risk. The HPEC development process is discussed in this chapter, followed by a full case study. For a more detailed technical overview of the process, the reader is directed to Chapter 3.

To expose the control needs, system-level simulations, concept-of-operations investigations, and up-front processor architecture simulations will be used. Early prototyping of critical software modules can be used to control large software codes for HPEC systems. COTS hardware is purchased off the shelf, reducing the expense of specialized design and development. Extrapolation to the broader system is sometimes necessary since it is not feasible or cost-effective to examine all alternative solutions at full scale. COTS programmable systems use more energy and take up more space than their ASIC equivalents.

It is preferable to benchmark representative hardware as soon as possible. The RAPTOR* space-time adaptive-processing (STAP) radar signal processor project is a great example of how HPEC works. This system represented the state of the art in HPEC systems at the time it was implemented (approximately 1989). The processor's general needs were established during the first spiral.

On the Spectre and Meltdown Processor Security Vulnerabilities

Due to ever-more complicated software, complex hardware, running untrusted downloaded code, and cloud cotenancy, the attack surface is continually rising. The Spectre variations raise concerns about the industry's half-century-old concept of hardware correctness. Strategic industry goals, such as accelerators and vector instructions, which are required for growing performance, are anticipated to open up new opportunities. Spectre variants are a type of "side-channel" attack5 in which the state of the microarchitecture is visible at the architectural level. This state may contain secrets that have been loaded into the shared architectural state.

Fixing Spectre by simply removing such speculation, we believe, will make it difficult to develop viable, high-performance solutions. A processor delivers instructions before resolving control flow dependencies, such as branching, while speculating. A hostile attacker intentionally misspeculates instructions to exploit Spectre.

One of the Meltdown hacks [SysTrans8] has made a variety of CPU architectures susceptible. Exfiltration can occur here because the border between microarchitectural and architectural states is breached. In certain circumstances, Meltdown can be prevented by keeping secrets out of the L1 data cache. It may even be feasible to modify the BTB state from a separate simultaneous multithreading (SMT or hyperthreading) thread due to functional unit sharing. The mitigation is either limiting the amount of speculation or substituting a safe alternative for indirect branches.

The Operating System can limit indirect branch speculation using a new microcoded processor speculation control interface (SPEC CTRL) added to Intel and AMD microarchitectures. For certain applications, the interface is implemented as an MSR (Model Specific Register) in microcode, which has a performance impact. Single Threaded Indirect Branch Predictors have been added to Intel x86 CPUs, which may be exploited by an application.