Benchmarking the Tock operating system for embedded platforms

CSE 221 Project

Maximilian Apodaca, Grant Jiang, Gabriel Marcano

February 2, 2021

1 Introduction

The goal of this project is to measure CPU and memory operations of the Tock operating system. Tock allows embedded systems to also support third party drivers and kernel extensions [5]. Currently, the developers of Tock are rewriting Tock's syscall interface. Our benchmarks of the OS's operations before and after will show the performance differences between the old and new system syscall interfaces.

Since Tock is implemented in Rust and has libraries written in both Rust and C/C++, our benchmarks will be written in C/C++ and Rust. We will also try to measure any difference between having benchmarks in C/C++ vs Rust. Currently, our compilation settings for compiling are TBD.

We will first create our experiments on QEMU and then run them on an OpenTitan Earl Grey chip implemented on an Nexys Video Artix-7 FPGA [3]. Since we will run our final benchmarks on the FPGA board, we expect measurements to be mostly consistent, compared to running tests in QEMU.FIXME We have yet to divide up who performs what experiment, as there is only one board which Gabe has.

This project lasts for the duration of the quarter (10 weeks). We each plan to spend at most 10 hours per week (FIXME hopefully not that much a week, but we will see, setup has taken quite some time for Gabe).

2 Machine Description

2.1 Hardware

The FPGA bitstream loaded onto the Nexys Video development board implements an OpenTitan Earl Grey microcontroller (FIXME citation to docs?). The CPU core of the microcontroller is an Ibex RISC-V 32-bit CPU, configured to run at 100 MHz, with 4 kB of instruction cache (FIXME anything else relevant about its instantiation?). The microcontroller does not implement a

typical MMU, instead implementing Physical Memory Protection (PMP) per the RISC-V Privileged Specification, version 1.11 (FIXME link?). The CPU PMP supports up to 16 memory regions.

All of the memory used by the system is kept within the microcontroller. The microcontroller has 16 kB of ROM used to store the primary boot loader, 512kB of embedded flash to store the actual program data for the system, and 64 kB of SRAM as scratch space. There is no external memory support. It takes one cycle to access data from SRAM, as well as from flash (FIXME confirm this, I know flash is emulated, and that for a long time they had cache disabled, probably because of single cycle memory access timing). ROM, e-flash, and SRAM are mapped to the processors address space and can be accessed directly.

The Earl Grey microcontroller supports GPIO, SPI, UART, and JTAG interfaces to interface with it. Internally, it uses a customized data bus to connect all internal peripherals to the CPU (TLUL bus interconnect). (FIXME what is the bandwidth of the different components?)

We have done the testing on version X (FIXME need to settle on a version of the bitstream for testing).

2.2 Operating system

The operating system is Tock [2], an "embedded operating system designed for running multiple concurrent, mutually distrustful applications on Cortex-M and RISC-V based embedded platforms" [5]. The internals of the operating system are written in Rust, with most parts of the kernel, including all drivers, written in safe Rust, and only low level portions hardware specific components written in unsafe Rust. The kernel uses the Ibex PMP provided by the Earl Grey microcontroller to segregate running applications from each other.

The operating system loads applications from flash on boot, and there is no real way (FIXME really?) to load applications dynamically after booting. Applications run preemptively, while kernel level instructions (capsules/drivers and below) execute cooperatively (citation? FIXME).

This version supports the original version of the system call interface to the Tock operating system. (FIXME need to determine what version/release/commit of Tock to use).

3 Experiments

4 Operations

The goal of an embedded operating system is to keep overhead low. This is because embedded operating systems run on much slower hardware. For example there is an over 15000x performance difference between the latest AMD CPUs [4] and the Earl Grey CPU we used [1]. To determine the overhead we ran a series of micro-benchmarks to measure common operations. These can be roughly grouped into the categories Context Switching, Memory access, Inter process

communication, and disk access. In this section we discuss our methodologies and results for each of these categories.

4.1 Context Switching

A context switch is the most basic operation in any program. This includes the time it takes to make a procedure call, system call, start a new task, or switch to a running tasks. We first measured the overhead of a time measuring operation. Using the $RDCYCLE$ and $RDCYCLEH$ instructions we can measure how many CPU cycles an operation took. To measure the measurement overhead we repeatedly start and stop and start the measurement and measure the increase in time of each additional start and stop.

The simplest type of context switch if a procedure call within the same process. Since this occurs very often it is paramount that the overhead be as minimal as possible. We profile the procedure call overhead by measuring the time between a function invocation and the first instruction execution in the function. After this we subtract our measurement overhead as determined in the previous experiment.

Another important aspect of context switching is making a syscall. In a broader sense this is crossing the kernel-userspace boundary. Tock has two ways to cross the kernel-userspace boundary. The first is making a syscall to the kernel, the second is receiving a callback from kernel space. We intend to measure both of these directions. In this experiment we first measured the latency of making a standard syscall by using the memop syscall is it is the most lightweight one. Then we can use a simple capsule to redirect a syscall as a callback to measure the kernel to userspace system call.

Creating tasks is often an expensive operation. As a result tock does not support dynamic process creation. Instead all processes are created at boot time. We used the debug memory of the FPGA with a modified kernel to measure the process creation time. This was profiled in the same way as the procedure calls.

The last major component of Context switching is switching the currently executing process. In tock's case we accomplish this with the yield syscall. We can trigger a context switch and write the timing counters before and after in each of the processes. This gives us the context switch time.

4.2 Memory

We have no data-caches in our processor so a constant access time should be reported. If not we can look into this again. We can verify the single cycle cache access time.

This can be measured with an array copy over a large array, the lack of data cache makes cache line prefetching not an issue and the instruction cache should save time when loading instructions to give a better result.

We don't have virtual memory to speak of and all allocation is static. We can however change the brk and sbrk values which allows us to test page faults.

We might want to look at the overhead of the *allow* syscall instead as this is how we can give a capsule access to process memory.

4.3 Network

We might want to use IPC or we can attempt to profile USB communication.

4.4 File System

We can do this but there is no file system cache. Instead we might want to profile the instruction cache as reading instructions from disk is by far the most common use. It is possible to write to flash for a program however it is a byte addressable array and has no FS to speak of.

The instruction cache could affect this as flash bandwidth is shared. It would be interesting to measure the read time. We can do this and it should give clean results. We don't have to worry about any caches as our program reads directly from the device.

We could set up an IPC channel that writes to flash in another process, however there is no support for a TCP stack or file system.

We can do this for userspace applications however capsules use a cooperative multiprocessing environment and might give interesting results. We would expect throughput to be slightly less than 1/N for each process.

References

[1] *Earl Grey Top Level Specification*. `https://docs.opentitan.org/hw/top_earlgrey/doc/`. Accessed: 2021-02-01.

[2] Amit Levy et al. "Multiprogramming a 64kB Computer Safely and Efficiently". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP'17. Shanghai, China: ACM, Oct. 2017, pp. 234–251. ISBN: 978-1-4503-5085-3. DOI: `10.1145/3132747.3132786`. URL: `http://doi.acm.org/10.1145/3132747.3132786`.

[3] *lowRISC/opentitan*. `https://github.com/lowRISC/opentitan`. Accessed: 2021-02-01.

[4] *Open Benchmarking Core Mark*. `https://openbenchmarking.org/test/pts/coremark`. Accessed: 2021-02-01.

[5] *tock*. `https://github.com/tock/tock`. Accessed: 2021-02-01.