Draft: 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 [2]. The Tock operating system is undergoing a system call interface change, so it would be beneficial to benchmark the operating system before the change to form a baseline to compare the changes against. This project will provide that baseline.

Tock is implemented in Rust and supports userland applications written Rust and C/C++. The benchmarks for this project will be written in C/C++ and Rust (FIXME will we do all? Or just a select few). The generated binaries will be compared to ensure that the critical parts of the benchmarks are similar enough in the implementations using different languages (FIXME would time be better spent elsewhere?).

The following compilers and flags are used for building the benchmarks: TBD

Benchmark application development will take place on Linux x86_64 platforms, leveraging QEMU to simulate the OpenTitan platform, and the final results will be acquired by running the benchmarks on an OpenTitan Earl Grey microcontroller instantiation on a Nexys Video Artix-7 FPGA [3]. Running the tests on the FPGA instead of on the simulators should help reduce uncertainty introduced in timing due to QEMU running as a userspace application (and subject to the sharing of systems resources and time). (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.

2 Machine Description

2.1 Hardware

The FPGA bitstream loaded onto the Nexys Video development board implements an OpenTitan Earl Grey microcontroller (`https://docs.opentitan.org/hw/top_earlgrey/doc/`). The CPU of the microcontroller is an Ibex RISC-V 32-bit CPU, configured to run at 100 MHz, with 4 kB of instruction cache and no data cache. It has a two stage pipeline consisting of a an Instruction Fetch (IF) stage followed by an Instruction Decode and Execute (ID/EX) stage (some instructions may spend more than one cycle in the execute stage). As configured for the Earl Grey microcontroller, there is no branch prediction (there is no branch penalty if the branch is not taken). The CPU does not support virtual memory, instead implementing Physical Memory Protection (PMP) per the RISC-V Privileged Specification, version 1.11 [6]. The CPU PMP is configured to support up to 16 memory protection regions. (for more information on stalls, see `https://github.com/lowRISC/ibex/blob/master/doc/03_reference/pipeline_details.rst`). There is no floating point unit attached to this CPU.

All of the memory used by the system is kept within the microcontroller; it does not support interfacing with external memory. The microcontroller has 16 kB of ROM used to store the primary boot loader, 512kB of embedded flash (e-flash) to store the actual program data (such as the operating system and application programs and data), and 64 kB of SRAM as scratch space. It takes one cycle to access data from SRAM, and currently also one cycle to read from ROM and e-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 by the CPU.

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), which was built using Xilinx Vivado 2020.2.

2.2 Operating system

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

The operating system loads applications from e-flash on boot, and there is no way currently to load new applications dynamically the initial loading of applications. Applications run preemptively, while kernel level instructions (capsules/drivers and underlying kernel code) execute cooperatively.

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

The operating system was built using the following compiler and linker:

- TBD, some kind of LLVM 11 or 12 and a modern enough clang

3 Operations

The operations benchmarked for this project can be grouped the following categories:

- CPU, scheduling, and OS services
- Context switching
- Memory access
- Inter-process communication
- Disk access

There is no networking support in the Earl Grey microcontroller.

This section contains the methodologies and results for each of these categories. Benchmarking methods are inspired by the approaches taken by lmbench [4] and hbench [1].

3.1 CPU, scheduling, and OS services

libtock-c and libtock-rs provide the runtime interface to the OS for applications.

3.1.1 Time measurement overhead

Timing information is accessible via a system call for driver 0, command 2 to get the current tick count of the alarm driver. The frequency of the alarm driver is given by a system call for driver 0, command 1. See the System call overhead section for more information about system calls. libtock-c exposes this system call through the `alarm_read()` function, and libtock-rs exposes it through the `alarm_read()`. (FIXME– need to figure out if the alarm driver uses the system clock or something else for timing... if not, we might have to dig deeper to use instruction counter support).

Timing measurement overhead is is done by acquiring the tick frequency and then calling the timing system call many times in quick succession (without

looping, to avoid accounting for looping overhead). The average of the differences (FIXME or geometric mean???) between every two timing calls should account for the overhead incurred by measuring time (FIXME anything about caching we should worry about?).

FIXME We should disassemble a test binary to see if procedure calls are optimized out. If not, we should do this test in an assembly language portion.

3.1.2 Loop overhead

The CPU pipeline does not use a branch predictor, so there is some overhead for every branch taken (there is no stalls in the pipeline added for a branch that is not taken).

- A simple loop
- A forcefully alternating loop
- More complicated loops (to confirm the presence or absence of a branch predictor)
- Large loops to probe for the existence and behavior of the instruction cache

3.1.3 Procedure call overhead

Per RISC-V calling convention (https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md), the only registers that need to be saved by the callee are registers s0-s11 (the stack pointer is unmodified on a procedure call, and does not need to be saved). Arguments are passed in via the a0-a7 registers for integer values (this CPU core does not support floating point instructions). If there are more than eight integer/pointer arguments, the first eight are passed in via registers, and the remainder are passed in via the stack.

Need to make sure to avoid compiler inlining when benchmarking overhead of procedure call

The methodology for testing is to capture timestamps before and after the procedure calls. Many different empty procedure calls will be made, with different numbers of arguments, to test the impact of the number of arguments passed on procedure call speed.

3.1.4 System call overhead

Tock supports four kinds of system calls:

- Yield (0) - Yields execution to another process
- Subscribe (1) - Used to assign callbacks to respond to system events
- Command (2)- Instructs a driver to do something
- Allow (3) - Share an area of userspace with the kernel

- Memop (4)- Memory manipulation operations

System calls can be made through libtock-c and libtock-rs, or by manually calling `ecall` with the right parameters (`https://github.com/tock/tock/blob/master/doc/Syscalls.md#0-yield`).

(FIXME need to test to make sure libtock* functions are optimized properly, so tests don't include procedure call overhead. Worst case scenario, we can do ecall calls manually in C or C++, and/or figure out how to do that in Rust efficiently)

3.1.5 Threading overhead

As far as I can tell, Tock does not support threading for userspace applications

Tock does not support multiple threads per process at the moment. What the operating system supports in lieu of threads is the ability to subscribe callbacks to be called in the case of events. (FIXME is there any good way to check the overhead? We can measure how long it takes for a subscribe syscall, but that doesn't actually run the callback– the callback is run asynchronously. Maybe if there is a way to have the callback run instantly?).

3.1.6 Task creation time

Tock only creates new processes when it boots up, and there is no way, currently, to create a new process afterwards. Applications are loaded sequentially from e-flash, until all applications are loaded, an invalid application header is found, or the hardware runs out of RAM. After all applications are loaded, the scheduler begins and starts scheduling processes for execution. (FIXME is there any way to salvage this section? I'm not there is a good way to measure from userland how long it takes to do this. We could modify the kernel for this specific test, but I'd rather not do that, just to keep everything consistent with every other step.)

3.1.7 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.

3.2 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.

3.3 Network

While Tock does have support for UDP on IPv6, the Earl Grey microcontroller has no networking hardware.

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

3.4 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] Aaron B. Brown and Margo I. Seltzer. "Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel X86 Architecture". In: *SIGMETRICS Perform. Eval. Rev.* 25.1 (June 1997), pp. 214–224. ISSN: 0163-5999. DOI: 10.1145/258623.258690. URL: https://doi.org/10.1145/258623.258690.

[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] Larry McVoy and Carl Staelin. "Lmbench: Portable Tools for Performance Analysis". In: *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*. ATEC '96. San Diego, CA: USENIX Association, 1996, p. 23.

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

[6] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified*. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf. Accessed: 2021-02-02. RISC-V Foundation, June 2019.