First draft: Benchmarking the Tock operating system for embedded platforms

CSE 221 Project

Maximilian Apodaca, Grant Jiang, Gabriel Marcano

February 18, 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 in the near future, 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 development board [3]. Running the tests on the FPGA instead of on QEMU should 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.)

Currently: Max: * Measurement Overhead * Context switch time Grant: * Loop Overhead Gabe: * Procedure call overhead * System call overhead

This project lasts for the duration of the quarter (10 weeks). Each team member plans 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 (implementing the RISC-V RV32IMC specification), 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 two cycles to access data from SRAM, and currently also two 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?)

Testing on the Tock operating system git commit X, and on OpenTitan git commit 99cb19827 (built using Xilinx Vivado 2020.1), as this is the latest version currently supported by the Tock operating system.

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 and Methodology

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

RISC-V offers performance counters, including a cycle counter. The cycle counter is a 64 bit register containing the number of cycles elapsed from an arbitrary starting point. As the Earl Grey CPU core does not have 64 bit general purpose registers, it offers two pseudoinstructions to read the upper and lower halves of the 64 bit counter, $RDCYCLEH$ and $RDCYCLE$ respectively.

The bottom 32 bits of the cycle counter should be sufficient precision to measure all benchmarks in this paper, as no benchmark window should be more than 4 billion cycles. The following is an example of taking a measurement, in assembly in Rust:

```
let timestamp: u32;
unsafe {
    asm!("RDCYCLE {}", out(reg) timestamp);
}

// Code to benchmark

let timestamp_end: u32;
unsafe {
    asm!("RDCYCLE {}", out(reg) timestamp_end);
}

let cycles = timestamp_end - timestamp;
```

And the same code in C/C++:

```
uint32_t timestamp;
asm ("mov %1, %0" : "=r" (timestamp));

// Code to benchmark

uint32_t timestamp_end;
asm ("mov %1, %0" : "=r" (timestamp_end));

uint32_t cycles = timestamp_end - timestamp;
```

Timing measurement overhead is measured by taking a timestamp before and after a set of timestamp reads. The following C code shows an example:

```
uint32_t timestamp;
asm ("mov %1, %0" : "=r" (timestamp));

uint32_t timestamp_tmp;
asm ("mov %1, %0" : "=r" (timestamp_tmp));
asm ("mov %1, %0" : "=r" (timestamp_tmp));

asm ("mov %1, %0" : "=r" (timestamp_tmp));
asm ("mov %1, %0" : "=r" (timestamp_tmp));

asm ("mov %1, %0" : "=r" (timestamp_tmp));
asm ("mov %1, %0" : "=r" (timestamp_tmp));

asm ("mov %1, %0" : "=r" (timestamp_tmp));
asm ("mov %1, %0" : "=r" (timestamp_tmp));

uint32_t timestamp_end;
asm ("mov %1, %0" : "=r" (timestamp_end));
```

```
uint32_t cycles = (timestamp_end - timestamp)/4;
```

The example takes 4 pairs samples back to back, and divides the enclosing timing by the number of pairs sampled to estimate the number of cycles it took for a single sample to execute.

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.

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

The methodology for testing is to measure the time between a process call and the first instruction execution in the function, by timestamping before the procedure call, and once again at the start of the procedure in question. The two timestamps are then subtracted, and then the measurement overhead from the Time measurement overhead section is subtracted from the result, providing the time it took for the procedure call to occur. 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)

One of the simplest system calls is a Command system call for driver 0, command 2, used 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()`.

System call overhead will be tested by acquiring timestamps before and after a series of system calls to get the current alarm tick count. Averaging the measured time over the number of system calls made should produce an estimate of the system call overhead.

FIXME The kernel calling userspace functions in the form of a callback are not really system calls. We can measure the overhead of the kernel calling a userspace function if we really want: "Then we can use a simple capsule to redirect a syscall as a callback to measure the kernel to userspace system call."

3.1.5 Threading overhead

Tock does not support multiple threads per process at the present. 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 on the OpenTitan Earl Grey MCU only creates new processes when booting, and there is no way, currently, to create new processes 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. Something like this could work, maybe?

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.)

3.1.7 Context Switching

Tock user-level applications are preempted by the scheduler. Userspace applications can communicate with each other with some IPC facilities provided by the kernel, and this can be used to test context switching by explicitly handing control from one application to another through the Yield system call (FIXME or by having one application Yield from the beginning, and have the other wake it up via IPC, and measure how long it takes for it to wake up). Triggering a context switch by Yield (and or? IPC) and writing the timing counters before and after in each of the processes provides the context switch time.

3.2 Memory

The OpenTitan Earl Grey microcontroller has three distinct regions of memory: SRAM, Flash, and ROM, all mapped to the CPU address space.

3.2.1 RAM access time

RAM access time is measured by employing the strategy outlined in the lmbench paper for measuring back-to-back-load latency [4]. The lmbench paper measures back-to-back-load "because it is the only measurement that may be easily measured from software and because we feel that it is what most software developers consider to be memory latency." [4]

The lmbench paper provides this code fragment as an example of how to measure back-to-back-load latency:

```
p = head;
while (p->p_next)
    p = p->p_next;
```

Specifically, the code causes back-to-back-loads if each successive access forces a cache miss. Per the paper, some CPUs implement a "critical word first" optimization where the "subblock of the cache line that contains the word being loaded is delivered to the processor before the entire cache line has been brought into the cache." [4] If another load is done, with a resulting cache miss, before the previous cache line is completely loaded the cache the CPU must stall the load until that operation completes. The total time between these loads is the back-to-back-load latency. In order to test back-to-back-load latency, then, each access must be approximately the length of the cache line apart.

The specific approach being taken for this benchmark makes no assumption about the cache size of the Earl Grey CPU, with the goal to demonstrate that there is, in fact, there is no data cache. As such, multiple tests are carried out with data 16, 32, and 64 bytes apart, to show that the behavior does not change regardless of the spacing between data in memory, as expected due to the lack of a data cache.

There is no data cache in the processor so a constant access time should be reported. As there is no data cache, each load and store instruction should complete in a constant number of cycles, ideally one cycle for SRAM, and possibly one cycle for flash as well (based on the implementation of "flash" in the FPGA bitstream).

3.2.2 RAM bandwidth

Memory bandwidth is measured in three approaches, as described by the lmbench paper [4]. The first approach measures copy bandwidth (a combination of read and write bandwidth) by using an existing memory copying routine, in this case memcpy, to copy large buffers from one portion of memory to another. The second approach measures read bandwidth by using a manually unrolled loop to read large amounts of data, printing the sum of all values read (to prevent the compiler from optimizing out the memory accesses). The third approach measures write bandwidth, by writing a constant value over a large memory buffer with the use of an unrolled loop.

The OpenTitan Earl Grey microcontroller has three distinct regions of memory, but only two of these are accessible from userspace in Tock, namely, flash and SRAM. Measurements will be taken from both regions separately.

If SRAM and flash read and writes take two cycle per access, optimally, both read and write bandwidths should be close to 200 MB/s, as the CPU clock is 100MHz.

3.2.3 Kernel memory operations

The Tock operating system does not support virtual memory and thus has no concept of page faults [1]. However, it does support some memory-related system calls. The brk and sbrk memop system calls alter the memory layout, specifically the application boundary, of the current application. Benchmarking brk and sbrk should yield timing about adjustments to memory protection regions.

FIXME? 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.

It is estimated that each system call for brk and sbrk will take roughly double the amount of time to get the timer count (3.1.1), due to overhead in updating kernel internal structures and the memory protection unit registers.

[1] There is a pending pull request with similar functionality at `https://github.com/tock/tock/pull/2424`

3.3 Network

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

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

3.4 File System

Tock does not provide any native filesystem support, but it does provide applications access to raw flash memory.

3.4.1 Size of file cache

Tock does not have a 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.

FIXME can we access a filesystem on USB?

3.4.2 File read time

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.

3.4.3 Remote file read time

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.

3.4.4 Contention

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 * memory bandwidth for each process.

4 Measurements and discussion

5 Conclusion

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 A
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.