Benchmarking the Tock operating system for embedded platforms

CSE 221 Project

Maximilian Apodaca, Grant Jiang, Gabriel Marcano

March 11, 2021

1 Introduction

The goal of this project is to measure CPU and memory operations of the Tock operating system [1]. The Tock operating system is undergoing a system call interface change, so it would be beneficial to benchmark the original system call interface to form a baseline to compare the new version against. This project provides that baseline, benchmarking CPU and system operations, memory, and inter-process communication facilities on Tock.

Tock is implemented in Rust and supports userspace applications written Rust and C/C++. The benchmarks for this project are written in C. It was considered whether to also implementing the benchmarks in Rust, but this was decided against due to the lack of IPC support from the Rust Tock userspace library.

1.1 Compilation flags

The additional flags are used for building the benchmarks on GCC:

- `-O3 -U_FORTIFY_SOURCE`

By default, the Tock C userspace library, libtock-c, is built using `-Os`, which tends to prevent inlining in order to save space. For many tests, this lack of inlining added undesired extra overhead, thus the build flags in the libtock-c build system were modified to use `-O3` instead. One of the computers the team used required the use of the `-U_FORTIFY_SOURCE` flag to build and link with libtock-c, due to that symbol being defined by default on the cross compiler used, which conflicted with other definitions in the userspace library and the newlib (version 4.1.0) standard library the compiler was using. All other flags remained unmodified as declared in the libtock-c build system.

Tock itself was built without modifying any of its default compiler flags. By default, Tock is built in Release mode with debugging symbols. For more information on the build environment and compilers used, refer to section 2.3.

1.2 Development

Benchmark application development took place on Linux x86_64 platforms, leveraging QEMU to simulate the OpenTitan platform, and the final results were acquired by running the benchmarks on an OpenTitan Earl Grey microcontroller instantiation on a Nexys Video Artix-7 FPGA development board [4]. Running the tests on the FPGA instead of on QEMU should reduce uncertainty introduced in timing due to QEMU running as a userspace application thus having to share systems resources and time.

Benchmark development was carried out by all team members, and Gabriel Marcano ran the actual experiments on the physical FPGA development board. Table 1 lists which benchmarks were done by which team member.

Maximilian Apodaca	Threading overhead, Task creation timing overhead, Context switch, Kernel memory operations, IPC connection overhead, Size of file cache, File read time
Grant Jiang	Loop overhead, RAM access time, IPC round trip time
Gabriel Marcano	RAM Bandwidth, Time measurement overhead, Procedure call overhead, System call overhead, IPC peak bandwidth

Table 1: Distribution of benchmarks across team

This project lasted for the duration of the 2020-2021 UCSD Winter quarter (10 weeks). Each team member spent, on average, around 10 hours a week on the project, with longer periods of time near the beginning (to set up the physical hardware) and near the end (to complete the report and tests).

1.3 Modifications to Tock and user libraries

Some modifications were necessary for both Tock and user libraries in order to be able to benchmark Tock on the Earl Grey OpenTitan microcontroller. The CPU of the microcontroller does not expose performance counters to userspace, so a new capsule, Perf, was developed to expose two new system calls to get at the lower 32-bits of the cycle counter and instruction count registers. For the purposes of this work, 32-bits of cycle and instruction precision is enough as no timestamp would be more than four billion cycles or instructions apart from another. Also, patches were applied to the Earl Grey platform code in Tock to enable IPC support and to increase the number of memory protection regions Tock is able to manage, which is required for proper IPC usage.

libtock-c, the userspace C library, was modified to add support for the Perf capsule system calls, to facilitate getting the performance data from the kernel. In addition, system call related functions in libtock-c were moved to header files

and declared inline in order to allow the compiler, when using optimization level `-O3`, to inline the system calls and eliminate procedure call overhead.

2 Machine Description

2.1 Hardware

Table 2 summarizes the basic specifications of the OpenTitan Earl Grey microcontroller.

CPU	Ibex RISC-V RV32IMC
External Clock	10 MHz
Instruction cache	4 kiB
Data cache	None
Pipeline	3-stage (experimental writeback), variable execution length
Branch prediction	None (no penalty for not taken)
Memory protection	RISC-V Physical Memory Protection[11]
Memory regions supported	16 (1 locked by bootrom)
Memory	SRAM, Flash, ROM
ROM size	16 kiB
SRAM size	64 kiB
SRAM latency	1 cycle [7]
Flash size	512 kiB
Interfaces	GPIO, SPI, UART, and JTAG
Networking	None

Table 2: OpenTitan Earl Grey Microcontroller specifications

The FPGA bitstream loaded onto the Nexys Video development board implements an OpenTitan Earl Grey microcontroller [4]. The CPU of the microcontroller is an Ibex RISC-V 32-bit CPU (implementing the RISC-V RV32IMC specification), configured to run at 10 MHz, with 4 kiB of instruction cache and no data cache. It has a three 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), and has an experimental writeback stage at the end. 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 [11]. The CPU PMP is configured to support up to 16 memory protection regions. 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

kiB of ROM used to store the primary boot loader, 512 kiB of embedded flash (e-flash) to store the actual program data (such as the operating system and application programs and data), and 64 kiB of SRAM as scratch space. SRAM is documented to have a 1 cycle latency to access. The documentation does not state how long it takes to read from ROM or flash ROM, e-flash, and SRAM are mapped to the processors address space and can be accessed directly by the CPU in kernel/machine mode if the PMP is configured to allow access to the right memory regions.

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). The only peripheral interface used for benchmarking is UART as a way to get program output from the board.

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" [8]. 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 userspace applications from each other. Figure 1 show the general architecture of Tock.

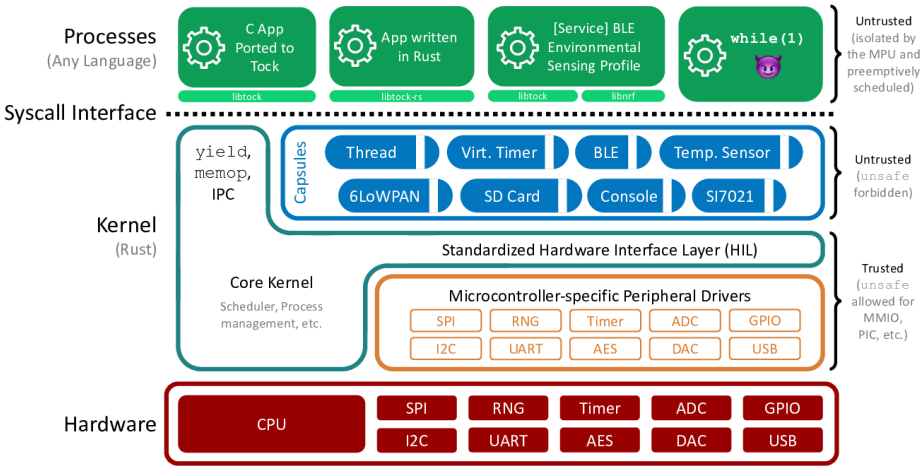


Figure 1: Tock architecture[10]

The operating system loads applications from flash on boot, and there is no way currently to load new applications dynamically the initial loading of ap-

plications. Applications run preemptively, while kernel level instructions (capsules/drivers and underlying kernel code) execute cooperatively.

2.3 Versions and builds

The Nexys Video FPGA devkit was programmed using the bitstream generated from OpenTitan git commit `99cb19827` (built using Xilinx Vivado 2020.1), as this is the latest version currently supported by the Tock operating system.

The version of Tock used for benchmarking supports the original, version 1 of the system call interface to the Tock operating system. Specifically, Tock was built using commit `b2141c0a7` from a fork maintained by the team [9]. The fork adds IPC support to the Earl Grey board configuration, and adds a performance capsule to allow userspace applications to query the number of retired instruction and the current cycle count.

The compilers and libraries used to build Tock and the benchmarks are shown in table 3. The GCC RISC-V cross-compiler was built using Gentoo Linux's crossdev tool with stack protection disabled (useflag `-ssp`). Listing 1 shows the full set of configuration parameters used to build GCC.

Rust	rustc 1.51.0-nightly (c2de47a9a 2021-01-06)
GCC	riscv32-unknown-elf-gcc (Gentoo 10.2.0-r5 p6) 10.2.0
LLVM	LLVM version 11.0.1
libtock-c	904bef7[2]
newlib	4.1.0

Table 3: Compilers used for building applications and Tock

Listing 1: RISC-V RV32 cross-compiler configurations

```
Using built-in specs.
COLLECT_GCC=riscv32-unknown-elf-gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/riscv32-
   ↪ unknown-elf/10.2.0/lto-wrapper
Target: riscv32-unknown-elf
Configured with: /tmp/portage/cross-riscv32-
   ↪ unknown-elf/gcc-10.2.0-r5/work/gcc-10.2.0/
   ↪ configure --host=x86_64-pc-linux-gnu --
   ↪ target=riscv32-unknown-elf --build=x86_64-pc
   ↪ -linux-gnu --prefix=/usr --bindir=/usr/
   ↪ x86_64-pc-linux-gnu/riscv32-unknown-elf/gcc-
   ↪ bin/10.2.0 --includedir=/usr/lib/gcc/riscv32
   ↪ -unknown-elf/10.2.0/include --datadir=/usr/
   ↪ share/gcc-data/riscv32-unknown-elf/10.2.0 --
   ↪ mandir=/usr/share/gcc-data/riscv32-unknown-
   ↪ elf/10.2.0/man --infodir=/usr/share/gcc-data
   ↪ /riscv32-unknown-elf/10.2.0/info --with-gxx-
```

```
    ↪ include-dir=/usr/lib/gcc/riscv32-unknown-elf
    ↪ /10.2.0/include/g++-v10 --with-python-dir=/
    ↪ share/gcc-data/riscv32-unknown-elf/10.2.0/
    ↪ python --enable-languages=c,c++ --enable-
    ↪ obsolete --enable-secureplt --disable-werror
    ↪  --with-system-zlib --enable-nls --without-
    ↪ included-gettext --enable-checking=release
    ↪ --with-bugurl=https://bugs.gentoo.org/ --
    ↪ with-pkgversion='Gentoo 10.2.0-r5 p6' --
    ↪ disable-esp --enable-libstdcxx-time --with-
    ↪ build-config=bootstrap-lto --enable-poison-
    ↪ system-directories --disable-libstdcxx-time
    ↪ --with-sysroot=/usr/riscv32-unknown-elf --
    ↪ disable-bootstrap --with-newlib --enable-
    ↪ multilib --disable-fixed-point --with-abi=
    ↪ ilp32d --disable-libgomp --disable-libssp --
    ↪ disable-libada --disable-systemtap --disable
    ↪ -vtable-verify --disable-libvtv --without-
    ↪ zstd --enable-lto --without-isl --disable-
    ↪ libsanitizer --disable-default-pie --disable
    ↪ -default-ssp
Thread model: single
Supported LTO compression algorithms: zlib
gcc version 10.2.0 (Gentoo 10.2.0-r5 p6)
```

3 Operations and Methodology

The operations benchmarked for this project are grouped in 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 [3].

3.1 CPU, scheduling, and OS services

3.1.1 Time measurement overhead

RISC-V offers performance counters, including a 64-bit cycle counter. The availability of the performance counters from userspace depends on the CPU implementation, and unfortunately the Ibex CPU core used by Earl Grey does not offer access to performance counters from userspace. To work around this limitation, a new module, Perf, was added to Tock to expose a system call to return the lowest 32 bits of the cycle counter. The bottom 32 bits of the cycle counter is sufficient precision to measure all benchmarks in this paper, as no single benchmark sample window is be more than 4 billion cycles (around 429 seconds for the Earl Grey CPU). Listing 2 is an example of taking a measurement, in Rust:

Listing 2: Timing in Rust

```
let drivers = libtock::retrieve_drivers()?;
let perf = drivers.perf;
let timestamp = perf.cycles()?;

// Code to benchmark

let timestamp_end = perf.cycles()?;
let cycles = timestamp_end - timestamp;
```

And listing 3 is the the same code in C:

Listing 3: Timing in C

```
uint32_t timestamp = perf_cycles();

// Code to benchmark

uint32_t timestamp_end = perf_cycles();
uint32_t cycles = timestamp_end - timestamp;
```

Timing measurement overhead sample is acquired by taking two cycle counts in succession. The following C code shows an example:

Listing 4: Measuring timing overhead in C

```
uint32_t timestamp = perf_cycles();
uint32_t timestamp_end = perf_cycles();
uint32_t cycles = timestamp_end - timestamp;
```

This sample is acquired 1000 times and the results are analyzed to determine the actual overhead. Each sample measures the time it takes for a single Perf system call to complete, and as the Perf capsule system call returns the number of cycles elapsed at the instant the kernel code reaches that particular instruction, the difference in cycles between the first and second cycle counts

encompasses the time of the second half of the first system call execution plus the first half of the second system call. Figure 2 shows a sequence diagram of the process to take a measurement, highlighting the sample window.

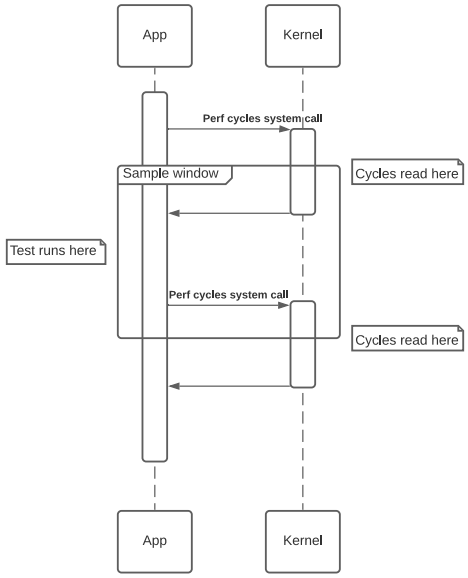


Figure 2: Timing sequence diagram

An identical setup is used to collect the number of instructions retired, by replacing `perf_cycles()` with `perf_instructions_retired()`. This is used to collect information about how many instructions are executed in a test. Collecting cycle and instruction data is done separately, in order to avoid recording the overhead of one in the other. One caveat of this approach that is not explored in this paper is possible discrepancies between the timing and instruction counts due to kernel preemption taking place in the middle of a sample window.

It is estimated that the timing overhead will be dominated by the system call overhead, as the actual performance majority of the time and cycles spent are in the kernel. Per listing 5, the Perf capsule code in Tock appears well optimized, so the capsule should add extremely little to the system call overhead (a few cycles). It is expected that the number of cycles taken should be around 200 (for around 100 instructions), or 20 microseconds, based on a cursory examination of the RISC-V specific Tock trap and system call code.

Listing 5: Tock Perf capsule disassembly

```
2000288e: addi    a0, zero, 2
20002890: beq     a1, a0, 18 <2000288e+0x14>
20002894: addi    a0, zero, 1
20002896: beq     a1, a0, 18 <2000288e+0x1a>
2000289a: bnez    a1, 22 <2000288e+0x22>
```

```
2000289c: mv      a0, zero
2000289e: addi    a1, zero, 1
200028a0: ret
200028a2: csrr    a1, minstret
200028a6: j       6 <2000288e+0x1e>
200028a8: csrr    a1, mcycle
200028ac: mv      a0, zero
200028ae: ret
200028b0: addi    a0, zero, 11
200028b2: ret
```

For all subsequent tests, timing measurement is done in C using a similar structure as listing 3, with the code to be sampled in place of the comment in the listing.

3.1.2 Loop overhead

There are three general loop cases covered by the benchmark, namely:

- An empty loop
- A loop with a not taken conditional within it
- A loop with an alternating conditional within it

The goal of the benchmark is to measure the overhead looping introduces and to observe any difference in timing due to differences in branching timing behavior.

Each test scenario involves acquiring 32 samples, with each sample consisting of running a loop 1000 times. The empty loop test consists of an empty for loop, with the only overhead being maintaining a loop counter and branching. The not taken test loop consists of the same overhead as the empty loop plus additional overhead incurred due to an additional branch that should always be not taken. The alternating loop test should be similar to the not taken test loop, except that the internal branch should alternate between taken and not taken for each loop. This is implemented with a ternary operator that tests if the loop counter is odd or even. Special care is taken in the implementation to ensure that the compiler does not optimize out the loops.

The CPU does not include a branch predictor, but there is a lower performance impact for a branch not taken (at least one less stall cycle), so there is some overhead for every branch taken. It is estimated that the loop overhead on its own (without timing overhead) should be around 4 cycles per loop, or 400 nanoseconds. The second test with the not-taken conditional should add an additional 4 cycles to the loop test overhead for a combined total loop overhead of 8 cycles, or 800 nanoseconds. The third alternating loop test should be similar in structure as the second test, with the added overhead happening due to a cycle stall when branching, so on average this looping should add an additional 2.5 cycles, or 250 nanoseconds, to the overhead, for an overhead of 10.5 cycles

per loop, or 1.05 microseconds. All of these estimates are summarized in Table 4, including the expected measured time for each sample (including timing overhead).

Test	Overhead per loop (µs)	Estimated sample timing (µs)
Empty	0.400	420
Not taken	0.800	820
Alternating	1.050	1070

Table 4: Cycle estimations for loops

3.1.3 Procedure call overhead

The methodology for testing the process call time is to measure the time taken to execute functions with different argument lengths and returns. Specifically, the following set of functions are tested:

- An empty procedure with no arguments and no return value
- An empty procedure with no arguments and returning a value
- An empty procedure with arguments ranging from 1 to 10, returning a value

Testing involves measuring the time a single procedure call takes 1000 times, for each kind of procedure being tested. The procedure calls are marked with compiler attributes (specifically `__attribute__((noinline, optimize(0)))`) to ensure they are not optimized out, so the benchmark actually measures the procedure call cost.

The procedure call overhead should be proportional to the required bookkeeping of registers to save and parameters to pass. Per the RISC-V calling convention [6], the only registers that must 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 (the Earl Grey CPU core does not support floating point instructions). If there are more than eight arguments, the first eight are passed in via registers, and the remainder are passed in via the stack. Memory latency is supposed to be 1 cycle [7], or 100 nanoseconds, thus there should be a small difference between passing arguments via parameters and the stack. Estimates are shown in table 5, assuming that a few instructions are required to save and restore instructions. As with all benchmarks, the measured values will also incur timing overhead per sample, which is estimated to be 10 microseconds per sample in section 3.1.1.

Test	Overhead (µs)	Total overhead (µs)
No returns, no parameters	1	11
Return, n parameter ($n \leq 8$)	$1.4 + 0.4n$	$11.4 + 0.4n$
Return, n parameter ($n > 8$)	$4.6 + 0.5(n - 8)$	$14.6 + 0.5(n - 8)$

Table 5: Procedure overhead estimation

3.1.4 System call overhead

System calls can be made through libtock-c provided functions, or by manually calling `ecall` with the right parameters loaded into registers. libtock-c was modified to allow the system call routines to be inlined when compiled with `-O3`, ensuring that there only overhead being measured is that of the system call itself and not additional overhead introduced by procedure calls.

System call overhead is tested by acquiring 1000 samples, consisting of timestamps before and a system calls to get the current CPU cycle count (with a similar approach for measuring instruction count). The timing overhead is subtracted from the value acquired to get the overhead of a system call. The best system call to use for such a test would be one with minimal overhead, which incidentally would be the same one used for timing as described in section 3.1.1.

As the system call being used to benchmark system call overhead is the same as was used for calculating timing overhead, the estimated cycles counts should be the same between this benchmark and the timing overhead test, or roughly, twice the estimated values from section 3.1.1, which should be around 40 microseconds or 200 instructions per sample.

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. This takes advantage of Tock's IPC functionality. The profiling of the IPC is discussed in section 3.3.

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 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.

3.1.7 Context switching

Tock user-level applications are preempted by the kernel scheduler at regular intervals, and scheduled on the Earl Grey platform based on their priority,

where the priority is assigned based on load order. Userspace applications can communicate with each other with some IPC facilities provided by the kernel, and thus force a context switch through IPC system calls. As a result the timings for a system call and process switch can be found in section 3.1.4 and section 3.3.

3.2 Memory

The OpenTitan Earl Grey microcontroller has three distinct regions of memory all mapped to the CPU address space: SRAM, flash, and ROM. Only SRAM and flash are accessible from userspace. All tests in this section only benchmark SRAM.

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 [3]. 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." [3]

The lmbench paper provides listing 6 as an example of how to measure back-to-back-load latency. Specifically, the code causes back-to-back-loads, possibly forcing successive cache misses. 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." [3] 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.

Listing 6: lmbench back-to-back memory test example

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

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, no data cache. As such, multiple tests are carried out accessing 500 memory locations 16, 32, and 64 bytes apart to show that the behavior does not change regardless of the spacing between data in memory which is 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 200 nanoseconds per memory access due to 100 nanosecond latency to access memory (one cycle latency)[7].

The benchmark test follows the same design as described by the lmbench which essentially traverses a linked list. For each round of tests, a 500 node linked list is constructed with each node separated from the next by the same number of bytes, or distance. The number of cycles required Then we record the number of cycles it takes to traverse the whole linked list. We estimate accessing each linked list node will take 4 cycles for a total of 2200 cycles including the cycles used for timing or 220 microseconds.

3.2.2 RAM bandwidth

Memory bandwidth is measured in four general approaches, three of which are described by the lmbench paper [3]. The first approach measures copy bandwidth (a combination of read and write bandwidth) by using an existing memory copying routine, in this case
textttmemcpy, to copy buffers from one portion of memory to another. The second approach does a similar test using a simple loop and loads and stores. The third approach measures read bandwidth by reading and summing data in a loop, printing the sum of all values read (to prevent the compiler from optimizing out the memory accesses). The fourth approach measures write bandwidth, by writing a constant value over a large memory buffer with the use of a loop.

Similar to the approach taken in the lmbench paper [3], all approaches are tested with buffers of different sizes, specifically, all powers of 2 from 32 to 1024, then from 1024 to 16384 in increments of 1024, inclusive. This will help identify any size dependent changes in memory handling, although none are expected. Kernel preemption might be noticeable with larger tests that lake longer to sample.

Per the OpenTitan SRAM documentation, SRAM access has a latency of once cycle[7], so assuming SRAM reads and writes take two cycle per word access total, optimally, both read and write bandwidths should be close to 20 MB/s or 19.1 MiB/s, as the CPU clock is 10MHz. Estimating overhead is difficult, since SRAM access latency is lower than all sources of overhead. Looking at the generated assembly of the tests indicates that at least some of the measurements are unrolled completely, in which case the only source of overhead would be timing related, which is estimated to be 20 µs. For all other tests involving loops, the overhead is estimated to be 20 µs plus the loop overhead estimation of 0.4 µs times the number of times the loop executes.

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.

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

Testing involves sampling 1000 times, where each sample makes a `setbrk` system call to increase the memory break point by 8 bytes every sample.

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 the overhead in updating kernel internal structures and the memory protection unit registers. This translates to 40 microseconds per `setbrk` system call, or 60 microseconds with timing overhead included per sample.

3.3 Inter-process communication (IPC)

While Tock does support UDP on IPv6, the Earl Grey microcontroller has no networking hardware. In lieu of testing network round trip time, peak bandwidth, and connection overhead, similar tests were implemented over IPC.

The IPC implementation in Tock is based on a client server abstraction. A server can register itself as a service that clients can look up, and both client and server can register callbacks with the IPC driver to be called when notified by the other. In addition, each process can optionally allocate a shared memory buffer which is passed to the callback belonging to the other application.

The methodology to measure the IPC overhead involves taking the time it takes to signal a process from another. This is done by creating a loop where the client notifies the server, and then the server notifies the client again. The server writes the current timestamp to a shared buffer owned by the client and notifies the client. The client then subtracts the current timestamp from the timestamp written into the shared buffer. This measures the time it takes to switch a single context from the server to the client.

The measurements are repeated in four different cases. The first is a client notifying the server to analyze how long it takes to switch from client execution to server execution. The next is a server switching to the client to measure the time it takes to switch from the server execution to the client execution. The third is measuring the round trip time which should be the sum of both of the uni-directional switches. Finally, the overhead of establishing a connection is measured by having the client find (and fail to find) an IPC service.

During most of the testing a shared buffer is allocated on the client side. This results in one extra entry in the memory protection unit for the server. As a result it is expected that switching to the server will take slightly longer than switching to the client. Overall, it is estimated that an IPC invocation will take 20 times as long as a system call, or 800 microseconds. This approximately due to the of the length of code observed in the Tock kernel to handle these tasks.

The round trip time is the amount of time between the client sending and receiving a reply from a server. After a ping is received, the time elapsed is recorded, and then the next ping gets sent. The estimated round trip time is 1.6 milliseconds, or twice the estimate for a one way communication.

The connection overhead is the overhead required by applications to find an existing IPC service running under Tock. The server application registers itself with Tock as an IPC server, and the client application does a IPC service lookup. The benchmark measures the amount of cycles required to find the

server, and the amount of time required to fail to find a server (by using a bad service name).

It is estimated that the connection overhead should be equal to that of a system call plus the cycles required to update kernel structures to establish a client with the server, which should be in the order of another system call. As such, it should take around 400 cycles or 40 microseconds to find an establish a connection with an IPC service.

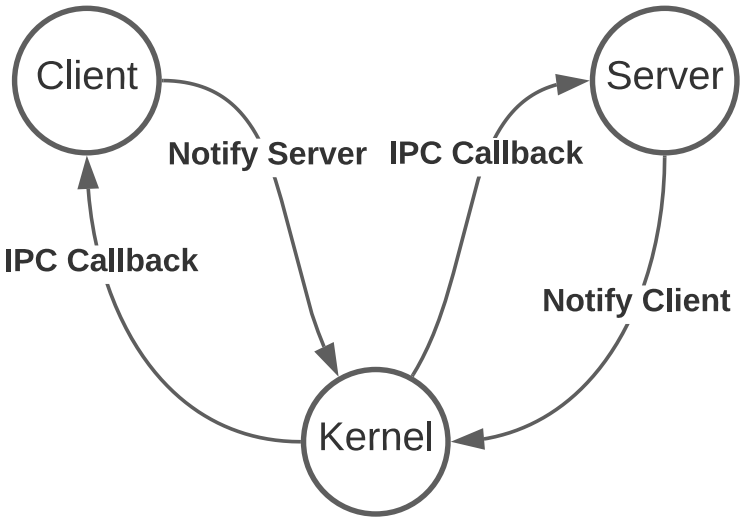


Figure 3: The IPC flow employed for the IPC tests.

3.3.1 Peak bandwidth

The testing methodology involves the client application setting up a 10 kiB buffer, recording an initial timestamp, and then sharing the buffer with the server and sending a notification to the server to wake it up. The server then wakes up, and reads the 10 kiB buffer, and records a timestamp once it is finished. It then sends another notification to the client along with its timestamp, so the client can compute the number of cycles elapsed. Figure 4 shows a graphical representation of the test.

It is estimated that each sample will take 1.884 ms, which consists of the timing overhead, one system call, one context switch, and coping 10 kiB of memory. The without the timing overhead, if everything else is considered part of the bandwidth computation, should take 1.864 ms, which translates to about 5.5 MB/s or 5.2 MiB/s for bandwidth.

3.4 File system

Tock does not provide any native filesystem support, but it does provide applications access to raw flash memory, where program code resides. Each application

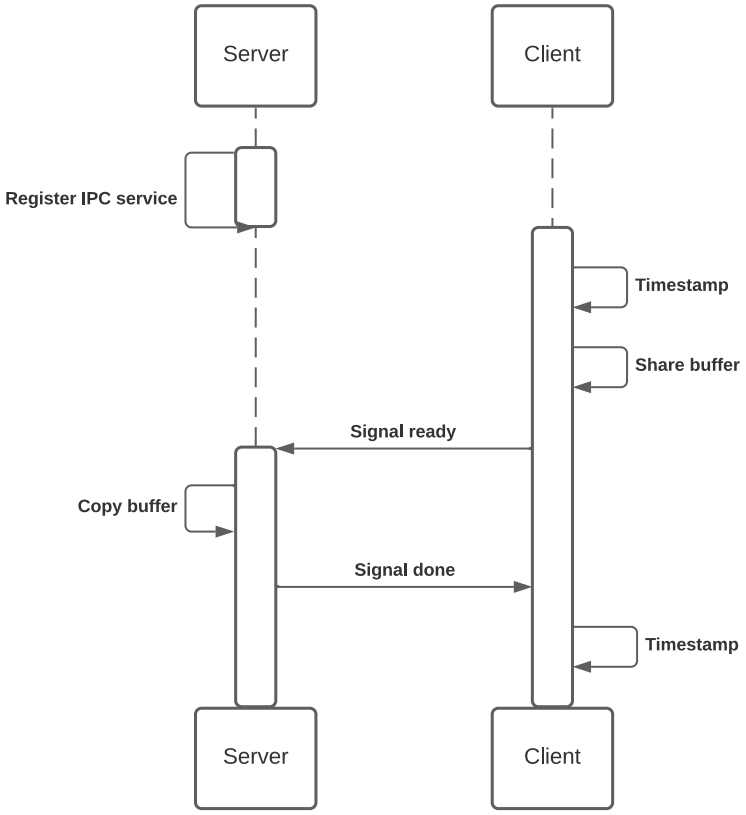


Figure 4: Sequence diagram of client server operations for IPC bandwidth test

is able to read (but not modify) its own program flash memory. However, due to a bug in the flash controller in the Earl Grey processor applications are unable to write to flash.[2]

3.4.1 Size of file cache

Tock does not have a file system and therefore does not have a file cache. However the CPU core does have an instruction cache. Since instructions are fetched directly from flash the effective file cache can be profiled by examining the instruction cache performance.

To measure the size of the instruction cache a loop is created with a variable number of NOPs. By adjusting the number of NOPs generated the time it takes to complete a loop is affected. If the time it takes to load each NOP is the same then the time for a loop iteration will increase linearly. However, there should be a change in performance when the instruction cache is saturated, leading to an increase in the time per instruction, thus yielding a different slope for the time increase data. Figure 5 illustrates the placement of the NOPs and program flow.

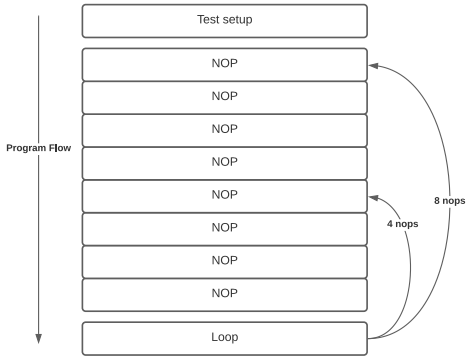


Figure 5: The layout of the test function in memory with two different NOP chain lengths.

Based on reading the parameters used to build the Earl Grey bitstream by the OpenTitan project, the instruction cache is 4 kiB in size. This means that after 2048 NOPs (as each NOP is a two byte instruction) in the loop there should be a change in the time per NOP in each iteration. The slope of the measurements should be one cycle per instruction until the 2048th NOP is reached, at which point we expect every 64 NOPs to evict an existing 64 NOPs. The expected slope would be around two cycles 200 nanoseconds per instruction.

[2]Interrupts do not appear to update status bits: https://github.com/lowRISC/opentitan/issues/4730

3.4.2 File read time

Since Tock does not support a filesystem, but it does expose the memory mapped flash regions of each application to the application as read-only memory. As an alternate test, the flash read bandwidth should provide a proxy of low-level filesystem performance, were once implemented for Tock. A simple loop that reads a continuous region from flash as well as a loop that does random four byte accesses are used to measure flash read performance. The expected throughput for flash operations is the same 20 MiB/s as for SRAM operations as they supposedly share the same SRAM backing in the FPGA. There should also be a 25% overhead for doing the sparse access due to the increased amount of computation per access. It is not possible to test file write as memory mapped regions of flash are mapped read only to userspace applications.

3.4.3 Remote file read time

This test is not done as there is no networking support.

3.4.4 Contention

This test is not done as there is no filesystem to content over.

4 Measurements and discussion

All the tests were implemented in C using libtock-c. Each test consisted of many samples which were then averaged together to produce results.

One of the most striking features of all of the results is that the timing was extremely deterministic. This was expected to some extent, as the Earl Grey microcontroller does not perform speculative execution and only offers a single execution pipeline, but not to the extent manifested. For all tests performed, the difference between samples was zero, meaning the standard deviation of each individual test was zero. It is hypothesized that using `printf` right after each sample window, which invokes a `yield` system call returning control to the kernel for some time, could reset the timeslice allocated for the application, so every sample loop, so long as it's short enough, is able to run uninterrupted.

It was more difficult than expected to interpret measurements due to the Earl Grey microcontroller's CPU implementation enabling a highly experimental feature adding a third pipeline stage for writeback. The documentation for the Ibex CPU core states that all performance descriptions in the documentation are relevant only for a two stage instantiation and that performance will be different with all three stages enabled[5].

4.1 CPU, scheduling, and OS Services

4.1.1 Time measurement overhead

Table 6 shows the summary of the time measurement overhead.

	Time (µs)	Instructions
Estimated	20	100
Actual	226.8	844

Table 6: Time measurement overhead

Timing overhead is measured to be 226.8 microseconds, or 844 instructions. This is significantly higher than the 20 microseconds that was estimated by an order of magnitude. As Listing 5 shows, the actual capsule driver implementation in Tock itself reduces to a fast dispatch and a single instruction for reading from the counter. This implies that something else in the kernel is taking a much larger number of instructions than expected before finally invoking the Perf capsule routines. The original estimate was based on a cursory overview of the Tock trap handler and context switching code for the RISC-V architecture, which have roughly 100 assembly instructions combined. This overhead should be equal to the one calculated in the system call overhead section, in section 4.1.4.

Listing 7 shows the relevant instructions used from userspace to set up the timing system call and to call it once again, which consists of the actual test. Before this code listing the program sets up the s2 register with the Perf capsule driver number to pass to the system call (0x00090004). Of note, the critical test portion is the section between and including the two `ecall` instructions, showing that GCC was able to inline the system calls adequately. This means all of the overhead measured is taking place inside the kernel.

Listing 7: Timing overhead benchmark disassembly

```
20030148:       mv      a1,s2
2003014a:       li      a2,1
2003014c:       li      a3,0
2003014e:       li      a4,0
20030150:       li      a0,2
20030152:       ecall
20030156:       mv      a5,a0
20030158:       li      a0,2
2003015a:       ecall
2003015e:       mv      a1,s0
20030160:       sub     a2,a0,a5
20030164:       addi    s0,s0,1
20030166:       addi    a0,s1,184 # 2003a0b8
2003016a:       auipc   ra,0x7
2003016e:       jalr    42(ra) # 20037194 <printf>
20030172:       bne     s0,s3,20030148
```

4.1.2 Loop overhead

Table 7 details the benchmark results for timing loops with different branches. The estimates of the amount of time taken for each of the loop tests are quite off compared to the actual timings, possibly due to the uncertainty in CPU timing due to the experimental CPU pipeline stage effects. Specifically, the difference in the estimates and the actual measured values is likely due to not understanding the specific delays for loads and stores and for branch taken, even though the documentation suggests SRAM has a latency of one cycle.

Loop Type	Raw time	Estimated overall	Loop portion
Empty Loop	0.7271ms	0.42ms	0.5003ms
Always not taken	1.1269ms	0.82ms	0.9001ms
Alternating Loop	2.2257ms	1.07ms	1.9989ms

Table 7: Time of each loop test (1000 loops)

The documentation states that integer instructions do not stall and that the branch taken will stall by at least once cycle, subject to memory latency for fetching the new instruction. Looking at the generated assembly, for the second and third test there are extra instructions emitted to set up the inner branches that the estimates did not take into account.

The for loop in the empty loop test is compiled to an add immediate and a branch taken. Together these take at least 2 non stalling cycles. The empty loop test reports 0.5003ms which indicates each loop iteration takes approximately 5 cycles or 500 nanoseconds. It is hypothesized that the extra three cycles are due to branch stalling in the pipeline.

The assembly of the for loop for the always not taken test consists of a load word, add immediate, branch not taken, and a branch taken. The always not taken test reports 9001 microseconds which indicates each loop iteration takes approximately 9 cycles to complete, or 900 nanoseconds. Each loop consists of 4 instructions with the branch taken stalling for three cycles. This hints the load stalled for two cycles, in accordance with findings from section 4.2.1.

The alternating loop test alternates based on parity of the loop counter which increments by 1. The test reports it used 19989 cycles or approximately 19 cycles per loop. Unfortunately, it was not possible to correlate each estimated instruction with the cycles per loop. For each loop iteration, regardless of the loop counter's parity, the loop consists of an and immediate, a load word, 2 add immediates, a store word, and a branch taken. This accounts for 13 cycles, 7 of which are stalls. When the loop counter is even, there is an additional branch taken taking 4 cycles. When the loop counter is odd, there is an additional branch not taken taking 1 cycle. This accounts for an average of 15.5 cycles per loop iteration. This leaves 3.5 cycles per loop iteration unaccounted for. It is possible that the load and store word instructions stall for more time, or there are other pipeline effects caused by the experimental writeback stage being enabled.

4.1.3 Procedure call overhead

Figure 6b shows the procedure call measurements as a function of how many arguments and returns are handled, and figure 6a shows the estimated values, for comparison. Note the scale of the y axis is not the same for both graphs. To clarify the values in the x axis, they represent the different tests as explained in section 3.1.3, beginning with an empty procedure, then a procedure with no arguments but now returning a value, and then procedures with the given number of arguments.

The estimated values were offset mostly due to the timing overhead estimation being incorrect (see section 4.1.1). A linear regression on the data used to generate both figures revealed the slope of figure 6b to be 0.375 µs, and of figure 6a to be 0.411 µs, which translates to a difference of less than a cycle per additional parameter between the estimates and the measured values.

Of note, it is not clear why there is a drop in time between the plain or void function test and the return function test, as the only difference is that the latter is actually returning a value and the former is not. In fact, looking at the instruction counts in figure 7, the return function test does execute more instructions. Other than this anomaly, the rest of the graph is more or less as expected. Due to low SRAM latency, there is not a drastic in performance between the 8 parameter and 9 parameter test. In fact the change appears to be in the opposite direction as expected, although slight, showing a smaller increase in time for adding the 9th parameter. Looking at the assembly code generated, that might be simply because the test function does not have to restore any registers related to the 9th and beyond, thus no overhead is generated on a return for those extra arguments, just on procedure call to pass the parameters.

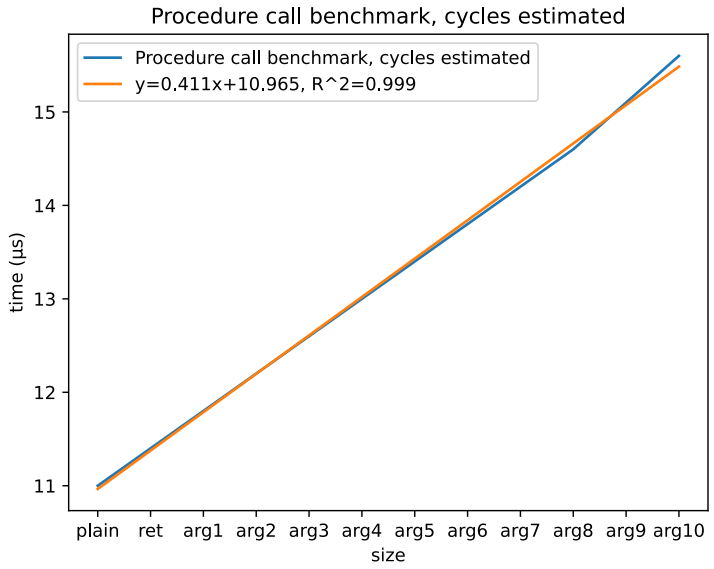
4.1.4 System call overhead

The actual operation values were computed by subtracting the time measurement overhead, shown in table 6 from the measured results. These computed results are shown at the end of table 8.

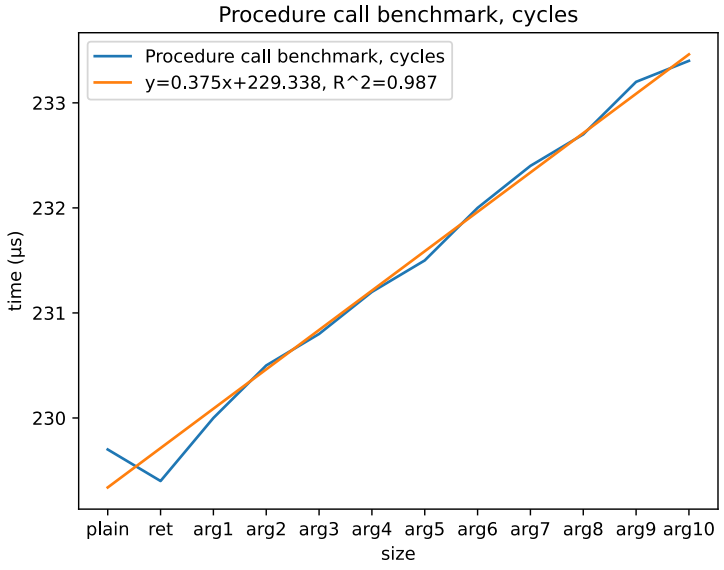
Measurement type	Time (µs)	Instructions
Estimated overhead	20	100
Estimated operation	20	100
Measured	453.5	1690

Table 8: System call overhead

The estimates are as inaccurate as they were for section 4.1.1 and for the same reasons, as the test is using the same system call for the code being tested as is used for timing purposes. The estimate predicted that the overhead and the operation should both be the same amount of time, and that does show in the actual data collected, noting that the measured data is almost exactly double the measured time for the time measurement overhead in section 4.1.1.



(a) Estimates, including overhead



(b) Measurements, with overhead

Figure 6: Procedure call measurements and estimates

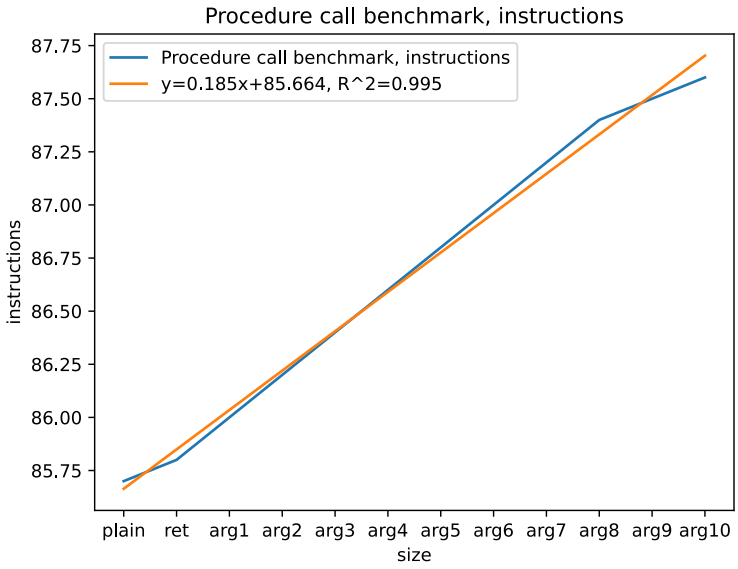


Figure 7: Procedure call measurements, instruction count

4.2 Memory

4.2.1 RAM access time

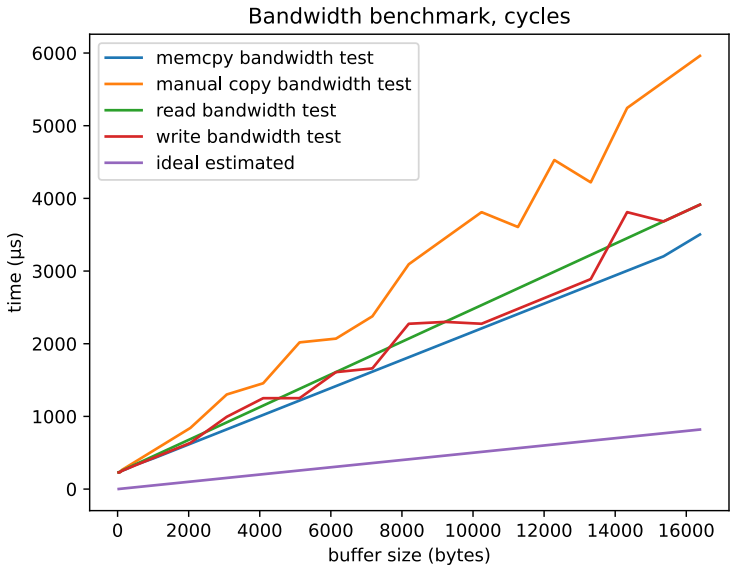
The measurements taken for the RAM access time test are all 627.1 µs exactly, regardless of the back-to-back distance employed. Without the overhead of recording the cycle counts and 500 loops, the actual memory accesses should take somewhere around 150.3 µs for 500 accesses, indicating a single load word instruction is around 300 ns or 3 cycles, implying a latency of two cycles instead of the one cycle latency predicted and expected per the platform documentation.

The back-to-back latency test would have reported a difference in across some of the tests if a small data cache were in use, thus the lack of variability across different distances tested indicates that there is not a data cache employed, consistent with known hardware specifications.

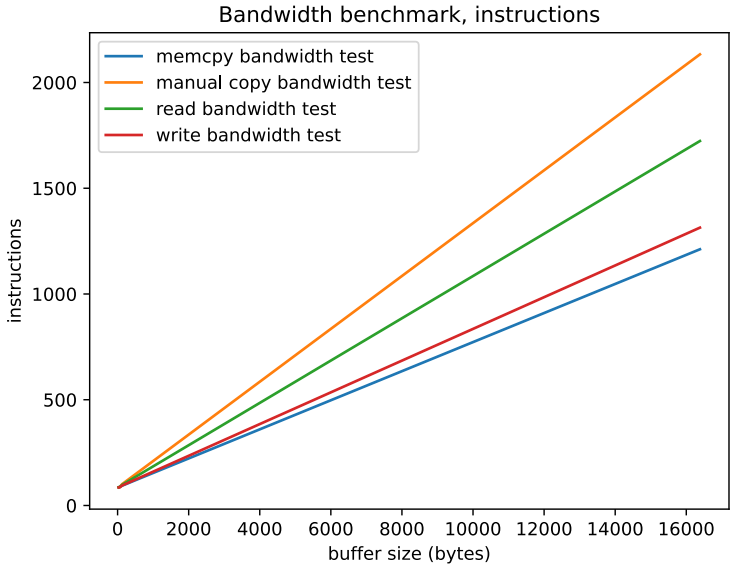
4.2.2 RAM bandwidth

Figure 8a shows the time required to complete each test with different size buffers, along with the estimated time for the ideal case. Figure 8b shows the number of instructions required to complete each test with different size buffers.

Memory bandwidth was difficult to compute from the measured data, as memory overhead is significantly lower than timing overhead and close to the same as looping overhead. As a result, overhead comprises a non-negligible amount of the measurements. Figure 8a shows that all measured tests were



(a) Timing measurements



(b) Instructions measurements

Figure 8: RAM bandwidth measurements and estimates

appreciably slower than the ideal estimate, by comparing the rate of increase in time as the buffer size increased. The effect of the overhead can be seen in figure 8b, where there is a clear correlation between the faster tests in figure 8b requiring a lower number of instructions.

One observation from looking at both figures is that while the instructions required as a function of buffer size appear extremely linear, the times reflected in figure 8a for some tests show some noise or jitter. It is speculated that for the larger buffer sizes each test takes long enough to be occasionally preempted by the operating system kernel, leading to increases in time. It is strange that the spikes did not manifest themselves as noise in the actual data collection, but only when looking at tests of different sizes. This quirk is possibly due to how deterministic the timing of the test environment is due to the test being the only application executing on the operating system, and that there are not many interrupts firing in the background, other than the timer interrupt which should be consistent, if not deterministic.

In lieu of trying to extract a bandwidth measurement from the figures, a study of the the generated assembly for some of the tests discovered that GCC successfully inlined some of the memcpy and manual memory copy loops, and that generally it had more success inlining memcpy than the manual memory copy approach. Moreover, GCC was able to also eliminate all looping for small copy amounts. An upper bound for the memory bandwith was computed using the average of the data from this optimized GCC memcpy, where it copied 32 bytes of data without looping. Table 9 shows the estimates and the measured time of the memcpy test. Assuming timing overhead is consistent and the same as in section 4.1.1, the actual time to copy 32 bytes (read 32 bytes and write 32 bytes) was 4.4 µs, which translates to a bandwidth of 14.5 MB/s or 13.9 MiB/s.

GCC supports pragmas to instruct it to unroll loops explicitly, but it did not seem possible to force GCC to unroll the loops father than some set constant. It is hypothesised that this limit might be related to the immediate offset range in the lw and sw operations, but this was not verified.

Measurement type	Time (µs)	Instructions
Estimated overhead	20	100
Estimated operation	12.8	16
Measured	231.2	860

Table 9: System call overhead

The findings from this test go slightly counter to the observations from section 4.2.1, where if SRAM latency is two cycles instead of one, the theoretical maximum bandwidth should be capped at 13.3 MB/s or 12.7 MiB/s. It is not clear what could be causing this discrepancy and memory benchmarks should definitely be re-visited in the future to help identify the actual hardware performance, as the documentation is incomplete in the presence of the experimental writeback pipeline stage.

4.2.3 Memory Protection Update

The set break system call was used to measure the time a memory protection update takes. This system call allows the program to set the program break, which forces an update of the memory protection regions and therefore allows us to measure the overhead of updating the MPU regions. The predicted a 0.45 milliseconds overhead per call to break was different than the measured amount 0.65 milliseconds. This is reasonably close to the correct value especially since if the actual measurement overhead is subtracted from the actual measurement overhead the result is 0.42 microseconds, which is much closer to the estimated value. If the system call overhead is subtracted from this processed value, the result is 197.4 microseconds to update the MPU regions. It is of note that a quick review of the Tock code to update memory regions appears to indicate a complexity of $O(n)$ per update, where n is the number of regions. This test only operates with $n = 2$. Table 10 shows the measured and estimated times in addition to the measured time minus our measurement overhead.

Timing type	Time (ms)
Estimated	0.4536
Processed	0.4242
Measured	0.6510

Table 10: Time taken by memory system call.

4.3 Inter-process communication

Direction	Raw Time	Processed Time	Estimated Time
Client to Server	2.1600 ms	1.9332 ms	5 ms
Server to Client	2.6595 ms	2.4327 ms	4.5 ms
Round Trip	4.6883 ms	4.4615 ms	9.5 ms
Connection overhead (success)	0.6823 ms	0.4555 ms	0.0400 ms
Connection overhead (failure)	0.6530 ms	0.4262 ms	0.0400 ms

Table 11: IPC Communication overhead

The IPC communication delay was measured to be on average 2.18 ms. This differs from our expected 4.75 ms estimate by over 2.2x. The most likely cause for the discrepancy is poor estimation of the amount of instructions needed to execute. However, another reasonable assumption is that the MPU context switch is faster than expected.

We predicted the client to server communication would take longer as an extra region has to be mapped into the server's address space before the server can execute. However the client to server context switch took less time than the

server to client switch. This might be due to the way Tock prioritizes processes. The server is loaded first which might give it priority over the client. This means that when the client contacts the server, the server runs immediately. But when the server contacts the client, control is transferred back to the server, and only after the server yields does the client run.

The round trip time should be a sum of the client to server and server to client times. However the sum gives 4.3659 ms which is 95.6 µs short. This is at most 956 instructions[5] and represents less than a system call as shown in table 6. As a result we believe the discrepancy is due to some extra checks the scheduler does to determine what process to schedule.

4.3.1 Peak Bandwidth

Table 12 shows the results from measuring IPC bandwidth. Of note, this is the only test where the first data sample differed from all others. This makes sense as the buffer sharing is set up only at the beginning, and due to optimizations deep in the Tock kernel if a requested section is already in place no further work needs to be done to share a buffer.

Measurement type	Time (µs)
Estimated overhead	20
Estimated operation	1864
Measured (first)	4060.0
Measured (steady)	3960.8

Table 12: IPC bandwidth test timing

If overhead is just the timing measurement, using data from section 4.1.1 the actual operation should have taken 3833.2 microseconds for the first sample and 3734.0 microseconds for all others. Both of these are double the estimated operations. The previous IPC results for a one way trip from the client to server takes around 2.1 milliseconds, leaving around 1.6 milliseconds to copy the data on the server side which is likely more than enough to copy 10 kiB of data, even including looping overhead. It appears IPC mechanisms take significantly more than just a context switch. Using the second case as the best case for bandwidth, and after substracting the actual measurement overhead from the measured test, the bandwidth is 2.7 MB/s or 2.6 MiB/s, which is about half of the estimated bandwidth.

4.4 Filesystem

4.4.1 Flash Access

Flash bandwidth was measured as an analog to reading a file from a filesystem. It was expected that the performance would be similarly to SRAM access since it was expected that the FPGA implementation might not be using real flash

Type	Throughput	Estimate	Overhead
Sequential	1.968 MiB/s	20 MiB/s	0.125 instr/load
Sporadic	1.396 MiB/s	15 MiB/s	8 instr/load

Table 13: Flash bandwidth and loop overhead

memory as it is lost on power loss. However, there was a much lower performance measured, possibly due to a difference in micro-architecture for handling flash and memory accesses.

Listing 8: Sequential flash copy disassembly

```
20030198:    4398                lw    a4,0(a5)
2003019a:    43d8                lw    a4,4(a5)
2003019c:    4798                lw    a4,8(a5)
2003019e:    47d8                lw    a4,12(a5)
200301a0:    4b98                lw    a4,16(a5)
200301a2:    4bd8                lw    a4,20(a5)
200301a4:    4f98                lw    a4,24(a5)
200301a6:    4fd8                lw    a4,28(a5)
200301a8:    5398                lw    a4,32(a5)
200301aa:    53d8                lw    a4,36(a5)
200301ac:    5798                lw    a4,40(a5)
200301ae:    57d8                lw    a4,44(a5)
200301b0:    5b98                lw    a4,48(a5)
200301b2:    5bd8                lw    a4,52(a5)
200301b4:    5f98                lw    a4,56(a5)
200301b6:    04078793            addi  a5,a5,64
200301ba:    ffc7a703            lw    a4,-4(a5)
200301be:    fd479de3            bne   a5,s4,20030198
```

A sequential read test was done by unrolling a loop reading 16 values from flash at a time. This unrolled loop was measured by acquiring the number of cycles before and after the loop. The code listing 8 shows the assembly generated by the loop. This shows that there are 2 instructions overhead for every sixteen loads. It was expected that this value to not greatly affect the bandwidth and therefore do not report corrected values.

Listing 9: Sporadic flash copy disassembly

```
2003026c:        054cec63            bltu  s9,s4,200302c4
...
200302c4:        000ca783            lw    a5,0(s9)
200302c8:        0405                addi  s0,s0,1
200302ca:        9d3e                add   s10,s10,a5
200302cc:        07f7f793            andi  a5,a5,127
200302d0:        0785                addi  a5,a5,1
```

```
200302d2:       078a                slli   a5,a5,0x2
200302d4:       9cbe                add    s9,s9,a5
200302d6:       bf59                j      2003026c
```

Accessing flash could go through some sort of cache. To rule this out a test doing sporadic accesses to flash was performed. However due to the extra computation some slowdown was expected. If there were some sort of caching there should be a major slowdown. There was a 30% slowdown which is not insignificant but can be explained by the increased number of instructions executed. The results in table 13 were generated by examining the generated assembly shown in listing 9.

4.4.2 Instruction cache

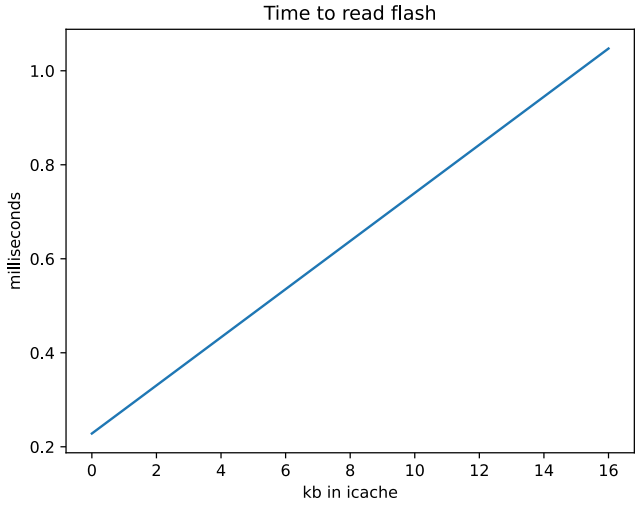


Figure 9: Time taken to read a certain number of bytes from flash. $R^2 = 1.000$

To measure the size of the instruction cache the benchmark repeatedly run a long section of NOPs. As the NOPs section increases in length it was expected to see a linear increase until the cache starts missing due to capacity conflicts. At that point there should be a steeper slope in the data. The results are shown in figure 9. The line is completely flat which indicates that the icache capacity does not affect performance. There are two hypotheses as to why there is no break as predicted: either the cache is able to prefetch the contiguous NOPs and therefore never miss, or the access latency of the icache is the same as the flash due to how they might be implemented in the FPGA.

5 Conclusion

5.1 Final observations and future work

In retrospect, many tests should have been written in assembly in order to reduce the amount of time spent trying to coax the compilers to emit the desired code. While this would make the tests less portable for other architectures (Tock does also run on some ARM platforms), it would make the tests performed on RISC-V systems easier to analyze.

Memory tests should be re-designed to account for the relatively low difference between integer operations and memory accesses. lmbench memory tests assume memory is much slower than the CPU, meaning that computing an approximate bandwidth is possible even if there are some CPU operations per loop, as the overhead each would add to the test would be minimal. This is not the case for the OpenTitan Earl Grey board, where memory latency is very much comparable to many CPU operations, meaning overhead is a significant part of the measured results.

In general, tests should be expanded to cover a wider range of buffer sizes and longer time spans to test the effects of kernel preemption on application timing. Also, there might be value in testing effects of running many applications simultaneously. In addition, testing should be done using some of the different schedulers available to Tock.

See table 14 for a summary of all of the estimates and results computed and measured for this paper.

Operation	Base Hardware Performance	Estimated Software Overhead	Predicted Time	Measured Time
Time measurement	100 ns	20 µs	20.1 µs	226.8 µs
Empty Loop overhead	N/A	400ns per loop	420µs	500.3µs
Always not taken	N/A	800ns per loop	820µs	900.1µs
Alternating	N/A	1050ns per loop	1070µs	1998.9µs
Procedure call overhead	N/A	400 ns per parameter	$14.6 + 0.5n$ µs	$229.3 + 0.375n$ µs
System call overhead	At least 100 ns	20 µs	40 µs	453.5 µs
RAM access time	At least 200 ns	20 µs	220 µs	627.1 µs
RAM bandwidth time	50 ns per byte	0 ns	50 ns per byte	69 ns per byte
Kernel memory operations	1.6 µs	One syscall (20 µs)	Two syscalls (20 µs)	651.0 µs
IPC client to server	N/A	5 ms	5 ms	2.1600 ms
IPC Server to client	N/A	4.5 ms	4.5 ms	2.6595 ms
IPC Round Trip	N/A	9.5 ms	9.5 ms	4.6883 ms
IPC connection success	N/A	0.0400 ms	0.0400 ms	0.6823 ms
IPC connection failure	N/A	0.0400 ms	0.0400 ms	0.6530 ms
IPC Bandwidth	19.07 MiB/s	1.953 ns/B	5.239 MiB/s	2.615 MiB/s
Flash access Sequential	19.07 MiB/s[a]	0.125 instr/load	19.07 MiB/s	1.968 MiB/s
Flash access Sporadic	19.07 MiB/s[a]	8 instr/load	15 MiB/s	1.396 MiB/s
Instruction cache[b]	Inflection at 4KiB	8B	Inflection at 4KiB	No inflection

[a] Earl Grey does not report flash bandwidth, we report the SRAM bandwidth instead.
[b] The instruction read time was 50.00µs/B however we did not see the predicted inflection point. See section 4.4.2 for details.

Table 14: Summary of full results

References

[1] 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.

[2] *libtock-c*. https://github.com/gemarcano/libtock-c. Accessed: 2021-03-11.

[3] 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.

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

[5] *Pipeline Details*. https://ibex-core.readthedocs.io/en/latest/03_reference/pipeline_details.html. Accessed: 2021-03-07.

[6] *RISC-V ELF psABI specification*. https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md. Accessed: 2021-03-09.

[7] *SRAM Controller Technical Specification*. https://docs.opentitan.org/hw/ip/sram_ctrl/doc/index.html. Accessed: 2021-03-09.

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

[9] *Tock*. https://github.com/gemarcano/tock. Accessed: 2021-03-09.

[10] *Tock Architecture*. https://github.com/tock/tock/blob/master/doc/Overview.md. Accessed: 2021-02-01.

[11] 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.