CS6354 Microbenchmarking Project

Group Members: nfz5mv, szz5ft

Environment

- macOS (Apple Silicon)
- Apple M1 Pro w/10 Cores
- Clang 17.0.0
- C11 standard
- No external dependencies

Directory Structure

- `include/` – common header file (`harness.h`)
- `src/` – source code:
  - `harness.c` – timing utilities
  - `00_function_call.c` – benchmark for function call overhead
  - `01_context_switch.c` – benchmark for syscall and thread context switches
- `scripts/` – helper scripts (`run_all.sh`)
- `report/` – report files
- `bin/` – compiled binaries (created by `make`)

Build Instructions

```
make
```

Run Instructions

Run each benchmark manually:

```
./bin/00_function_call
./bin/01_context_switch
```

Or run everything at once:

```
./scripts/run_all.sh
```

Methodology

0.1 Part 1: Function Call Overhead

0.1.1 Functions Under Test

We implemented a suite of functions designed to represent different cases in the calling convention:

1. No arguments (`f0`)
2. Single integer argument (`f1i`)
3. Two integer arguments (`f2ii`)
4. Single double argument (`f1d`)
5. Eight integer arguments (`f8iiiiiiii`) — forces stack spilling beyond register arguments
6. Struct argument (`f1s`) — tests by-value struct passing
7. Struct return (`f_ret_s`) — tests hidden pointer return mechanism

All functions have trivial bodies (ignoring inputs or returning a fixed struct) to isolate call overhead.

0.1.2 Measurement Procedure

1. **Baseline loop timing**: Execute an empty loop with a trivial sink operation and record duration *base_ns* to measure loop control overhead.
2. **Loop with function calls**: Execute the same loop but invoke the test function in each iteration and record duration *with_call_ns*.
3. **Overhead isolation**: Compute net function call time as:

   $$\text{diff} = (\text{with\_call\_ns} - \text{base\_ns}) - 2 \times \text{timer\_overhead}$$

   Divide by iteration count to yield nanoseconds per call.
4. **Repetition**: Repeat each measurement 21 times. Sort the samples and report the **median** as the primary metric, since the median is robust to OS jitter and outliers.
5. **Rounding**: Results are rounded to one decimal place for reporting clarity.

0.2 Part 2: Context Switch Overhead

0.2.1 Measurements Performed

We implemented two complementary experiments to quantify context switching costs:

1. **System call round-trip**: Measuring the cost of entering and exiting the kernel using a trivial syscall (`getpid()`).
2. **Thread ping-pong**: Measuring thread scheduling and context switch latency by synchronizing two threads with semaphores.

0.2.2 Measurement Procedure

1. **Baseline loop timing**: For each experiment, execute an equivalent empty loop structure (same nesting and iteration counts) and record duration $base\_ns$.

2. **System call timing**: In the syscall test, each iteration performs $K = 32$ consecutive calls to `getpid()` to amplify the signal. Record duration $with\_syscall\_ns$.

3. **Thread ping-pong timing**: A main thread and worker thread exchange control via semaphores for R round-trips. Each round-trip involves two context switches (main $\rightarrow$ worker, worker $\rightarrow$ main). A warm-up phase of 200 iterations is executed before measurement to remove startup effects.

4. **Overhead isolation**: For syscalls:

$$\text{diff} = (\text{with\_syscall\_ns} - \text{base\_ns}) - 2 \times \text{timer\_overhead}$$

   Divide by $(iters \times K)$ to obtain nanoseconds per system call.

   For thread ping-pong:

$$\text{cs\_overhead} = \frac{(\text{pingpong\_ns} - 2 \times \text{timer\_overhead})}{2R}$$

   where $pingpong\_ns$ is the timed duration of R round-trips.

5. **Repetition**: Each test is repeated (21 runs for syscalls, 7 runs for ping-pong). Samples are sorted, and the **median** is reported as the primary metric.

1 Results

1.1 Part 1: Function Call Overhead

Table 1 summarizes the measured overhead for different function signatures. For all simple cases (no arguments, integer arguments, double arguments, eight integer arguments, and struct-by-value), the overhead was consistently around **0.8 ns per call**. Returning a struct incurred a higher cost of **2.2 ns per call**, due to the hidden return pointer and additional data movement.

Function	**Overhead (ns/call)**
f()	0.8
f(int)	0.7
f(int, int)	0.8
f(double)	0.7
f(int $\times$ 8)	0.8
f(struct)	0.6
f() $\rightarrow$ struct	2.2

Table 1: Function call overhead across different signatures.

These results indicate that modern compilers and calling conventions keep function call overhead extremely low (typically within 2–3 CPU cycles), with the only notable

increase observed for struct returns. On repeating experiments we found out the overhead might vary, but all functions, excluding struct, still demonstrated consistency around 0.8 ns/call, with some test instances reporting consistent 0.8 across the board. One notable finding is when testing on a better CPU, the Apple Silicon M2 Max, the overhead is around 0.6, which is 25 percent faster.

1.2 Part 2: Context Switch Overhead

Table 2 reports the measured costs of system call round-trips and thread context switches.

Measurement	Overhead
System call round-trip (`getpid()`)	187.2 ns/call
Thread ping-pong context switch	1637.2 ns/switch

Table 2: System call and thread context switch overhead.

The system call round-trip was measured at approximately **187 ns**, corresponding to around 600 CPU cycles. Thread ping-pong switching was measured at approximately **1637 ns** ($\approx 1.6\,\mu$s), or roughly 5200 CPU cycles per switch.

1.3 Discussion of Results

The experiments confirm a strong hierarchy of overhead:

- Function calls are effectively negligible, except when returning complex types.
- Crossing the user-kernel boundary incurs roughly **200**$\times$ higher cost than a function call.
- Thread context switches are an order of magnitude more expensive than syscalls, reflecting scheduler involvement, synchronization, and state save/restore overhead.

Overall, the measurements align with values reported in prior work, validating the methodology.

Notes

- On macOS, `syscall(SYS_getpid)` shows a deprecation warning; this is expected and does not affect correctness.
- `bin/` and `build/` directories are generated by compilation and do not need to be submitted.
- During part 1, we run $\text{sink}_u 64+ = iinourbodylooptokeepthecompileroptimizersfromoptimizinga$