School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Profiling framework for seL4

by

Cameron Bourke

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Computer Engineering

Submitted:	January 1970	Student ID:	z5149731
Supervisor:	A/Prof. Person	Topic ID:	XX00

Acknowledgements

This work has been inspired by the labours of numerous academics in the Faculty of Engineering at UNSW who have endeavoured, over the years, to encourage students to present beautiful concepts using beautiful typography.

Further inspiration has come from Donald Knuth who designed TEX, for typesetting technical (and non-technical) material with elegance and clarity; and from Leslie Lamport who contributed LATEX, which makes TEX usable by mortal engineers.

John Zaitseff, an honours student in CSE at the time, created the first version of the UNSW Thesis LATEX class and the author of the current version is indebted to his work.

Contents

1 Introduction **1**

1.1 Motivation 1

1.1.1 Application to seL4 2

1.2 Thesis Problem Statement 2

2 Background **3**

2.1 Profiling 3

2.1.1 CPU Profiling 3

2.1.2 Types of CPU Profilers 4

2.1.3 Statistical Profiling Overview 5

2.2 Performance Monitoring 6

2.2.1 Microarchitecture 6

2.2.2 Performance Monitoring Unit (PMU) 7

2.2.3 Performance Counters 8

2.2.4 Counting vs Sampling 9

2.2.5 Programming the PMU 9

2.3 Platform Support 13

2.3.1 x86 Systems 13

2.3.2 ARM Systems 13

2.3.3 RISC-V Systems 14

3 Related Work 15

3.1 seL4 Benchmarking 15

3.1.1 Benchmarking 15

3.1.2 sel4bench 16

3.1.3 libsel4bench Overview 16

3.1.4 libsel4bench Extensions 16

3.2 Performance Counters for Linux (PCL) 17

3.2.1 The perf utility 17

3.2.2 perf record 18

3.2.3 perf report 19

3.2.4 Event Groupings 20

3.2.5 PCL User-level Interface 21

3.2.6 The perf File Format 22

4 Approach 24

4.1 Statistical Profiling Limitations 24

4.2 Design Goals 25

4.3 Requirements 25

4.4 Targeted Platform 26

4.5 Design 27

5 Plan 30

5.1 Stages 30

Bibliography 31

Appendix 1 **33**

A.1 Options 33

A.2 Margins 33

A.3 Page Headers 34

A.3.1 Undergraduate Theses 34

A.3.2 Higher Degree Research Theses 34

A.4 Page Footers 34

A.5 Double Spacing 35

A.6 Files 35

Appendix 2 **37**

B.1 Data 37

Chapter 1

Introduction

1.1 Motivation

When designing any system or application, there is how we theorise it will behave, and there is actually how it behaves. It is not uncommon to deploy a new application to production, only to find that it does not perform how we expected. However, diagnosing the performance issue typically cannot be resolved by simply inspecting the source code, looking for clues that may explain the unsatisfactory performance, but rather through measurement.

For example, if a web server takes significantly longer, compared to others, to resolve a particular kind of request, we need to understand which code-paths are being executed, and where in the system are unexpected amounts of time are being spent. Maybe the issue turns out to be that the CPU is stalled on memory I/O, or is spending too much time waiting for disk I/O. It could even be that the TCP connection between the client and the web server is sending too many TCP retransmits. The point is, without the ability to profile our systems or applications, we quickly become blind to the root cause of the performance issues we experience.

1.1.1 Application to seL4

seL4 is a microkernel, designed for building safety- and security-critical systems, due to its comprehensive formal verification and high performance [26]. It has proven to be sought-after in embedded systems, in cases which require a high degree of security or reliability. A core tenant of microkernels is to reduce the Trusted Computing Base (TCB). In seL4 this is achieved by hoisting OS functionality, that traditionally lives in the kernel, up into usermode servers, illustrated in Figure 2.1.

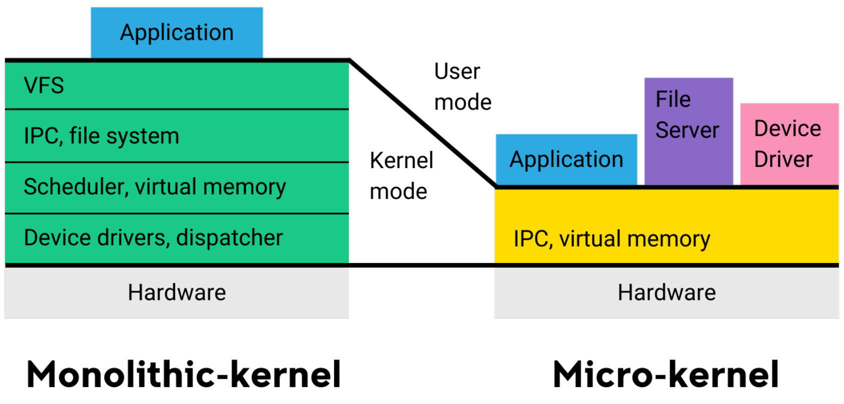


Figure 1.1: Comparison of monolithic and microkernels [20].

Since most system services do not reside in the kernel, but instead are implemented in user-space, it is critical that seL4 developers have sufficient tooling such that they can diagnose performance issues.

1.2 Thesis Problem Statement

In this thesis, we will evaluate the types of CPU profilers for their suitability in seL4, and seek to understand how the `perf` profiler on Linux is implemented. The result will be a statistical and event-based profiling framework for seL4 systems, capable of profiling both kernel and user-level code.

Chapter 2

Background

First, we will need to explore the various types of CPU profiling commonly in use today, in particular statistical profiling, and familiarise ourselves with the available hardware support for performance monitoring. Then, we will begin to look at how the modern profiler on Linux, `perf` is implemented.

2.1 Profiling

In the most general sense, we profile a program or system to gain a deeper insight into its runtime behaviour. The systems that we may want to profile can range from hardware, operating systems, networks and cloud infrastructure. Profiling differs from debugging, in that often we debug a program or system whenever it does not meet its functional requirements, compared to non-functional requirements.

2.1.1 CPU Profiling

In software systems, a large class of performance issues come down to understanding how and where execution time is being spent on the Central Processing Unit (CPU). Without that insight, it can be quite difficult to diagnose exactly where in the program the performance bottlenecks lie.

An example of a CPU profiler that has long been available on UNIX systems (in the case of BSD, since 1983 [13]) is `gprof`.

2.1.2 Types of CPU Profilers

CPU profiling is an umbrella term which encompasses a number of different types and approaches. However, there are three prominent types and techniques used to collect timing data that we should consider when determining which approaches are most applicable in the context of seL4.

Instrumentation

Instrumentation is a profiling technique where trace functions are executed at the start (prologue) and end (epilogue) of each function call. The trace functions are able to collect precise timings during each call, creating a detailed summary of how much time was spent in each function in the program.

With GCC for example, the `-pg` flag will generate trace functions that automatically collect timing information, which by default, is written to a file called gmon.out. The tool `gprof` can then be used to view the data. Alternatively, user defined trace functions can be called instead via the GCC flag `-finstrument-functions` [25].

Statistical

Statistical profiling involves sampling the program counter and call stack running on the CPU at regular intervals. It differs from instrumentation, in that it does not provide a complete picture of the program's execution, but rather an estimation. The trade-off is that statistical profiling allows the program to run closer to full speed, since the cost to profile is not incurred during each function call. We will discuss the mechanics of statistical profiling at greater length in Section 2.1.3.

Event-based

Both instrumentation and statistical profiling are mainly concerned with capturing timing information, such that execution time on the CPU can be attributed to functions within the program. Undoubtedly, this is a valuable tool when trying to quickly understand where in the program an unexpected amount of time is being spent. However, not all performance issues can be resolved in software alone, but rather require a greater insight into the microarchitectural events that are occurring within the CPU to understand the complete picture. These architectural events may be branch misses, cache misses, context switches, cache misses, page faults etc.

2.1.3 Statistical Profiling Overview

Code Execution

In this discussion on code execution, we refer to a thread as the basic unit of CPU utilisation, which consists of a program counter (PC), call stack, and register set. The PC is a register on the CPU that stores the address of the instruction currently being executed[1]. The call stack is a data structure, typically resident in Random Access Memory (RAM), that keeps track of the nested function calls, and store the required state for each function (such as local variables). The register set refers to the current value within each register on the CPU.

Sampling

Following from the brief overview of statistical profiling in 2.1.2, a statistical profiler will sample the PC at each interval. Over time, a number of samples will be collected, which can then be processed to generate an execution profile.

A statistical profiler needs to employ some mechanism in order to probe the CPU state at regular intervals. This is referred to as the profiling interrupt. In the case of `gprof`,

[1]For processors that implement instruction pipelining, when an instruction is in the execution stage (EX), the PC typically will no longer refer to that particular instruction, but rather the instruction in the instruction-fetch (IF) stage of the pipeline.

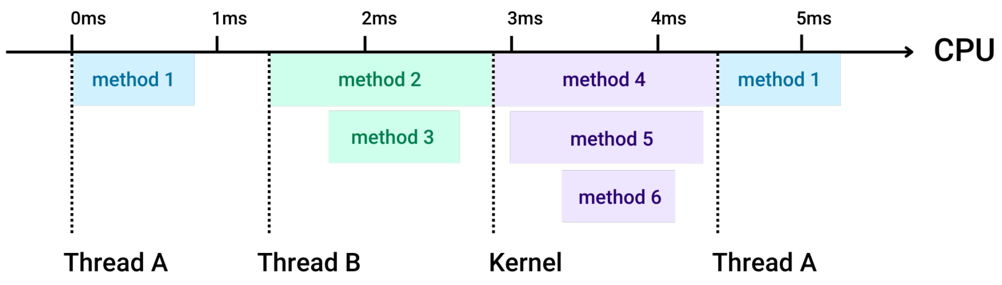


Figure 2.1: Sampling the CPU every 1ms. In this example, profiling begins at 0ms, where the first sample is recorded with thread A on the CPU executing method 1. The second sample occurs at 1ms, where there is no activity on the CPU. At 3ms, thread B is active and the call stack contains methods 2 and 3. The fourth sample at 3ms, records method 4 executing within the kernel.

initially on Linux (v2.0 and earlier) it used the syscall `setitimer` [16], which permitted it access to the underlying hardware timers. Then later it migrated to the more efficient `profil` syscall where the kernel could perform the probe on behalf of the user program, and therefore did not require two mode switches when the timer interrupt occurred. While hardware timers are still extensively used in modern processors, there is now dedicated hardware for performance profiling, which is covered in Section 2.2.2.

2.2 Performance Monitoring

2.2.1 Microarchitecture

The Instruction Set Architecture (ISA) is the interface between software and hardware. It is part of the abstract model of a computer, and defines how the CPU is controlled by software [8]. The term microarchitecture refers to how a given Instruction Set Architecture (ISA) is implemented in a particular processor. A processor (CPU) consists of a number of components with different responsibilities, that harmoniously coordinate to execute each instruction. To help illustrate how these components fit together, Figure 2.2 shows a hypothetical processor, where the RAM and system clock are not part of the CPU, but rather shared with the rest of the system.

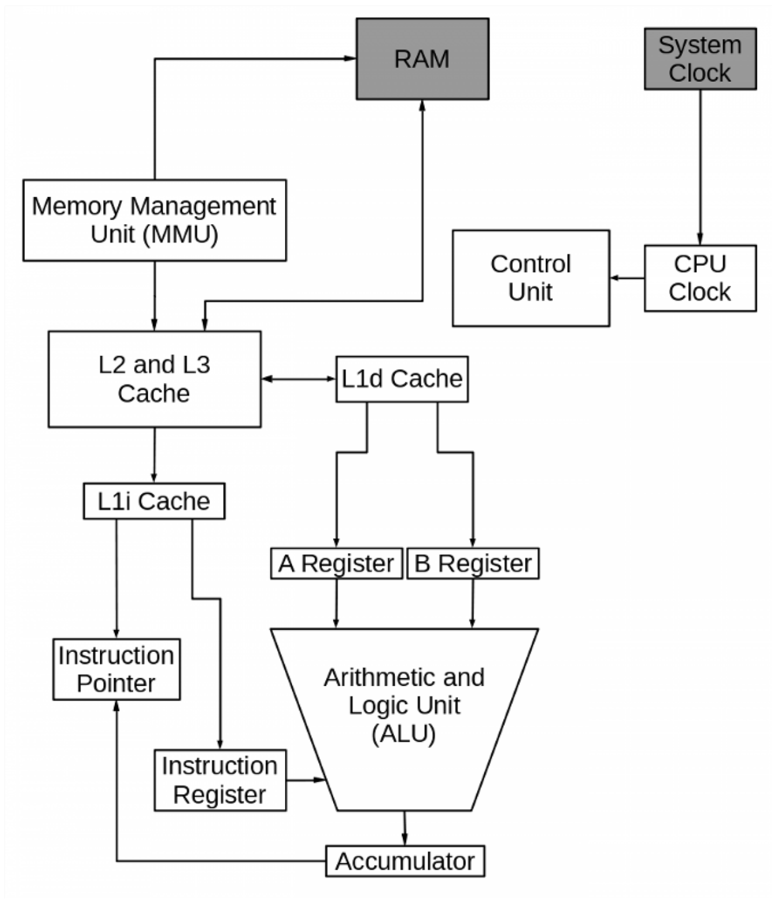


Figure 2.2: A simplified conceptual diagram of a typical CPU. The reality is much more complex than this, however a typical CPU will have: multiple layers of caching (L1, L2, etc), distinct L1 cache for instructions (L1i) and memory (L1d), a Memory Management Unit (MMU) to support virtual addressing, a control unit to coordinate the data-path and pipelining, and an Arithmetic Logic Unit (ALU) to execute arithmetic and logical instructions. Taken from a Red Hat blog post titled "The central processing unit (CPU): Its components and functionality" [12]

2.2.2 Performance Monitoring Unit (PMU)

A statistical profiler can help determine where in the program is an unexpected amount of time being spent, but unless the root cause is solely in software, it is not able to provide any insight into where time is being spent within the microarchitecture itself.

The Performance Monitoring Unit (PMU) is a non-invasive debug component, which can addresses this limitation, by providing a fixed number of counters that can count various useful events that regularly occur with the CPU itself. Note, while the PMU provides greater capabilities for software based profilers, it is also used by hardware engineers to debug potential hardware issues and perform microarchitecture benchmarks.

Hardware Events

The events available to be counted on the PMU vary from processor architecture (e.g x86 vs ARM), but they also can vary between processor to processor within the same architecture (e.g Cortex-M55 vs Cortex-M33). However PMU events can be typically grouped into the following event type categories [7]:

- Instruction execution
- Instruction speculation
- Cache behaviour
- External memory accesses
- Memory errors
- Branch prediction
- Exceptions
- Pipeline stalls
- CPU cycles
- Debug and trace events

2.2.3 Performance Counters

At a high level, the PMU allows a single, supported event to be assigned to one of the available performance counters. Commonly, the PMU can be interfaced via Model Specific Registers (MSRs), which are a set of registers on the processor[2] that dictate how the PMU should operate.

[2]Typically the registers are not located on the CPU itself, but rather on a coprocessor. This decouples the PMU from main processor since on most architectures the PMU is an optional extension.

Fixed and Programmable Counters

In practice, most PMUs offer a mix of both fixed and programmable counters. Fixed performance counters always count the same event within the CPU. They are used for common used events, such as CPU cycles and instructions retired. Programmable counters, as the name suggests, can be programmed to count any supported[3] event.

2.2.4 Counting vs Sampling

Generally, there are two distinct use cases when wanting to utilise the PMU. The first, known as *counting*, is when we want to record the number of times a particular event has occurred during a specific operation. To do so, we reset the value of the counter to 0, perform the operation, and then read the value of the counter again.

The second use case, known as *sampling*, is where we want the PMU to notify us once the count for a certain event reaches a particular value. When this occurs, known as a counter *overflow*, the PMU signals an interrupt informing us that the overflow has occurred.

2.2.5 Programming the PMU

To help illustrate how to interface with the PMU, we will demonstrate how to program the PMU to count the number of instruction software increments on an ARMv7-A processor that uses the Performance Monitors Extension version 2 (PMUv2). The instruction software increments refers to when the instruction has been architecturally executed. This will differ from the number of CPU clock cycles, since a given instruction may take multiple CPU cycles to execute. This event is also known as instruction retired on other platforms.

[3]On x86, some events can only assigned to a subset of the available performance counters, and therefore are considered semi-programmable. In order to determine which events can be assigned to which counters, x86 provides a mask for each event which specifies the supported counters.

Cortex-A15

The Cortex-A15 is a 32-bit multi-core processor, which implements the ARMv7-A architecture, and is well supported by seL4 [29]. The PMU on the Cortex-A15 provides implements the PMUv2 architecture and provides six programmable counters [2] and one clock cycle counter (PMCCNTR). The ARMv7-A architecture defines 12 performance monitor registers (MSRs) which are used to configure the behaviour of the PMU [11]. The PMU registers reside on Coprocessor 15 (CP15). For this example, inspired by a similar example from the Resource Allocation and Scheduling Group [1], only the following six registers are required:

- Performance Monitor Control Register (PMCR)
- Count Enable Set Register (PMCNTENSET)
- Overflow Flag Status Register (PMOVSR)
- Event Count Selection Register (PMSELR)
- Event Type Select Register (PMXEVTYPER)
- Event Count Register (PMXEVCNTR)

To begin, we will define a series of #defines to provide greater clarity regarding to the value being written to a particular PMU register (defined in Listing 1). Note, the constants are directly from each respective PMU register section in the ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition [6].

Initialising the PMU

When the processor is first powered on, we have to assume that the PMU registers are in an undefined state and therefore reinitialise them. This involves enabling counters, resetting them to 0, as well as the clock counter. To write to one of the PMU registers on ARMv7, we use the Move to Coprocessor (MCR) instruction [9] (Listing 2). Often, we want to use performance counters within a C program, and therefore we are using inline assembly, in particular Extended Asm [24].

Now that counters are initialised, we need to enable the specific counters that we want to use. In this case, we will enable counter 0, as well as the clock counter (Listing 3.

```
#define PMU_SOFTWARE_INC_EVT_ID          0x00

#define PMCR_EN_CTRS                    (0x1 << 0)
#define PMCR_EN_RESET_CTRS              (0x1 << 1)
#define PMCR_EN_RESET_CLK_CTR           (0x1 << 2)
#define PMCR_EN_CLK_DIV                 (0x1 << 3)
#define PMCR_EN_EXPORT_EVTS             (0x1 << 4)
#define PMCR_DD_CLK_CTR_PROB_REG        (0x1 << 5)

#define PMCNTENSET_EN_PMCCNTR           (0x1 << 31)
#define PMCNTENSET_EN_CTRS               0x3F

#define PMOVSR_EN_PMCCNTR               (0x1 << 31)
#define PMOVSR_EN_CTRS                   0x3F
```

Listing 1: Flags for PMU MSRs on ARMv7.

```
uint32_t pmcr_config = PMCR_EN_CTRS
                     | PMCR_EN_RESET_CTRS
                     | PMCR_EN_RESET_CLK_CTR;
asm volatile ("MCR p15, 0, %0, c9, c12, 0\t\n" :: "r"(pmrc_config));
```

Listing 2: Initialising the counters via the `PMCR`.

```
uint32_t pmcntenset_config = PMCNTENSET_EN_CTRS | PMCNTENSET_EN_PMCCNTR;
asm volatile ("MCR p15, 0, %0, c9, c12, 1\t\n" :: "r"(pmcntenset_config));
```

Listing 3: Enabling counter 0 and the CPU clock counter via the `PMCNTENSET` register.

```
uint32_t pmovsr_config = PMOVSR_EN_CTRS | PMOVSR_EN_PMCCNTR;
asm volatile ("MCR p15, 0, %0, c9, c12, 3\t\n" :: "r"(pmovsr_config));
```

Listing 4: Clearing the overflow bit (in case it was previously enabled) via the PMOVSR.

```
asm volatile ("MCR p15, 0, %0, C9, C12, 5" :: "r"(0x00));
asm volatile ("MCR p15, 0, %0, C9, C13, 1" :: "r"(PMU_SOFTWARE_INC_EVT_ID));
```

Listing 5: Assigning the software increment event to counter 0 via PMSELR and PMXEVTYPER.

Finally, we need to clear the overflow bit for both of the counters (Listing 4).

Counting an event on the PMU

Once we have initialised the PMU, we can assign the software increment event to counter 0. This is a two step process. First we specify which counter via PMSELR, and then we specify which event via PMXEVTYPER (Listing 5).

Reading an event count on the PMU

The PMU will begin to count the number of software increment events as the program continues to execute on the CPU. At any stage, the current count can be obtained from the PMU register PMXEVCNTR. In this case, we use the Move to ARM Register [10] instruction to move the value from a Coprocessor register to a core (CPU) register, which will be written into the variable counter_value (Listing 6).

```
uint32_t counter_value;
asm volatile ("MCR p15, 0, %0, C9, C12, 5" :: "r"(config.counter));
asm volatile ("MRC p15, 0, %0, C9, C13, 2" : "=r"(counter_value));
```

Listing 6: Reading the number of software increments in counter 0 via the PMXEVCNTR.

2.3 Platform Support

2.3.1 x86 Systems

x86 systems are ubiquitous in desktop computing, workstations, servers. A majority of CPUs in the TOP500 project based on the x86-64 instruction set architecture [TODO: Need to cite]. The x86-64 architecture is the 64-bit version of the x86 ISA. The x86 ISA is a Complex Instruction Set Architecture (CISC), which tend to have higher power requirements, but in return offer higher performance over RISC architectures for certain kinds of workloads, such as vector operations since x86 supports Single Instruction, Multiple-Data (SIMD) instructions. Both Intel and AMD produce processors that implement the x86 ISA.

Performance monitoring was first introduced in the Intel Pentium processor with a set of model specific performance monitoring counter MSRs. Intel refers to its x86 (32-bit) and x86-64 architecture implementations as Intel 64 and IA-32 respectively. There are two classes of performance monitoring capabilities offered by the Intel architectures [21]:

- *architectural* - visible behaviour of events are consistent across processor implementations.
- *non-architectural* - events are specific to the microarchitecture and vary from processor to processor.

The number of general-purpose performance monitoring counters can vary across processor generations within a processor family, across processor families, or could be different depending on the configuration chosen at boot time in the BIOS regarding Intel Hyper Threading Technology. Typically there are between 2-8 programmable performance counters, and three fixed counters.

2.3.2 ARM Systems

The ARM processors are frequently utilised in embedded systems, due to their low cost, minimal power consumption and lower heat generation compared to their competitors

[32].

The ARM processor family is divided into three architecture profiles [5]:

- Application (A-profile): Highest performance, targeted towards operating systems. The distinguishing feature from the other two profiles is that supports virtual memory via a Memory Management Unit (MMU).
- Real-time (R-profile): Fast response, targeted towards high-performance, hard real-time applications.
- Microcontroller (M-profile): Lowest power consumption, targeted towards microcontrollers and discrete processing. It is designed to be integrated into an FPGA.

The ARMv8 architecture (most recent is ARMv9, announced March 2021) supports up to 31 programmable performance counters, however in practice, processors that implement the ARMv8 architecture only provide between 4-8 counters [3] [4].

2.3.3 RISC-V Systems

RISC-V is royalty free, open-source ISA and processor specification for a Reduced Instruction Set Computer (RISC) architecture. Historically, RISC-V was prominently used in academia, since researches are able to change and experiment with the architecture. However, in recent years, there has been a sharp rise in the commerical viability of RISC-V processors, replacing systems that were traditionally dominated by ARM [31].

The RISC-V ISA specification defines three fixed performance counters (`hpmcounter0` to `hpmcounter2`) which are dedicated for cycle count, real-time clock and instructions-retired respectively. It also supports up to 29 programmable performance counters (`hpmcounter3-hpmcounter31`) [17].

Chapter 3

Related Work

We can now turn our attention to the current support for interfacing with the PMU in seL4, which prominently facilitates benchmarking in the kernel. Then, we will explore the performance subsystem on Linux, looking at both the user-space API, and how it is implemented.

3.1 seL4 Benchmarking

3.1.1 Benchmarking

The aim of benchmarking is twofold: (1) to detect performance regressions and (2) to identify opportunities for improvement. Benchmarking often takes the form as a suite of tests that are able to measure the performance of a given system. Benchmarking can be divided into standard and ad-hoc benchmarks. Standard benchmarks are designed by experts in the industry and tend to be based on macro-benchmarks, which attempt to represent real-world performance. The benefit of standardised benchmarks is that it provides a common standard, such that results can be compared. One example of a standardised benchmark is SPEC CPU 2017, which is designed to measure compute-intensive performance on a CPU [30].

3.1.2 sel4bench

The repo sel4bench provides multiple applications for benchmarking different paths in the kernel [27]. Most notably it contains an application that benchmarks the Inter-process Communication (IPC) mechanism in seL4. The benchmarks are ad-hoc, since seL4 is an experimental system and therefore none of the standardised benchmarks are compatible[1].

3.1.3 libsel4bench Overview

Interfacing with the PMU directly via platform dependent assembly instructions sand MSRs (see Section 2.2.5), while illuminating, requires further layers of abstraction before it can be useful to developers on seL4.

The seL4 benchmarks require access to the PMU counters, specifically to count the number of CPU cycles required to perform a particular operation. How CPU cycles, or any other PMU event is counted, depends on the underlying platform (e.g x86, ARM, RISC-V), as well as the specific architecture (e.g for ARM, this could be ARMv6, ARMv7, ARMv8 etc.) The libsel4bench library is designed to abstract over the performance monitoring counters (PMCs) [28].

3.1.4 libsel4bench Extensions

In order for libsel4bench to also support benchmarking and profiling on seL4, support for additional PMU functionality will be required.

PMU Sampling Support

Earlier, in Section 2.2.4, we saw that counting is when we are only interested in reading the PMU counters, compared to sampling where we configure the PMU to notify us

[1]Most benchmarking suites designed for operating systems target UNIX/POSIX like systems.

when a counter has overflowed. While libsel4bench provides extensive support for counting, it does not provide any support for sampling the performance counters.

Sampling is not necessarily required for benchmarking in seL4, since often it is suffice to sample the counter value directly before and after the operation to be benchmarked. However, for a statistical profiler, sampling support is fundamental since it is the mechanism that switches control back to the profiler such that it can snapshot the running threads state (i.e PC, call stack, register set).

3.2 Performance Counters for Linux (PCL)

The PCL is a kernel-based subsystem that provides a framework for collecting and analysing performance data [19]. It is also commonly known in the open source community as Linux perf events (LPE), or perf_events [18]. The subsystem was merged into the Linux kernel in version 2.6.31 [33] (most recent version is 5.17.4, released 20 April 2022).

3.2.1 The perf utility

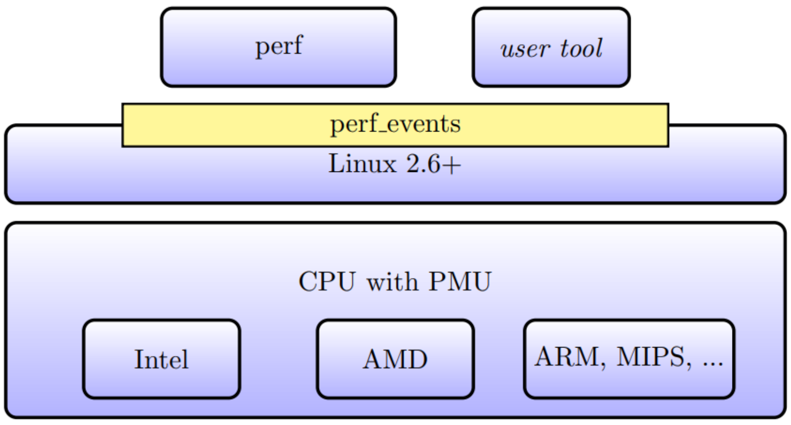


Figure 3.1: How how the perf utility relates to the PCL subsystem. Taken directly from Figure 1 in the CERN openlab, titled "perf file format" [15].

The perf utility is the Command Line Interface (CLI) to the PCL subsystem (perf_events) [22]. It is a high level interface (see Figure 3.1) that acts as an entry point for a number of commands, such as:

- `perf stat` - instruments and summarises key CPU counters (PMCs)
- `perf record` - records PMU events (to perf.data) which can be later reported
- `perf report` - breaks down events by process, function, etc and allows user to filter events
- `perf annotate` - annotate assembly or source code with event counts
- `perf top` - view live event count (in realtime)
- `perf bench` - run benchmarks for different kernel subsystems

3.2.2 perf record

The `perf record` command invokes the statistical profiler (see Section 2.1.3 for an overview of statistical profiling).

Example Usage

To familiarise ourself with the perf API, we will sample the CPU every 10000 instructions, such that we include the call stack in the sample, and filter out samples that were taken while the CPU was executing in kernel mode. We can specify with the shell command `perf record -g -e cycles:u -c 10000` where the argument:

- `-g` specifies that call stack should be included,
- `-e` specifies the sampling event,
- `-c` specifies the count at which a sample occurs.

However, when CPU cycles is the sampling event, it is often more convenient to sample based on a sampling frequency (in Hz), `perf record -g -F 99 -all-user`, where the argument `-F` specifies the frequency to sample and `-all-user` specifies that to sample whilst CPU is in user-mode.

3.2.3 perf report

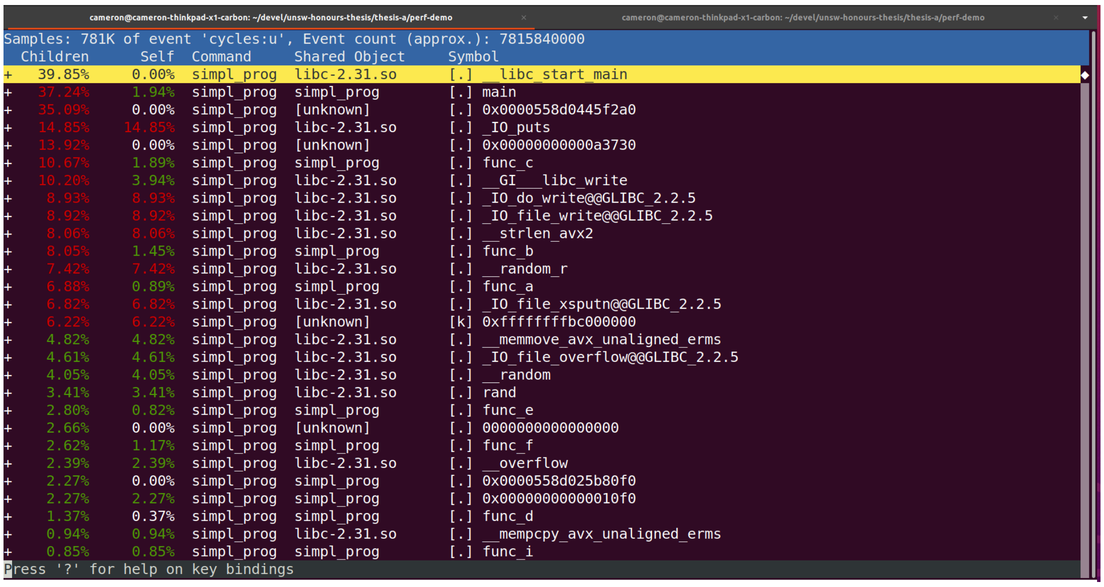
The perf report command is responsible displaying the profile data generated by perf record. Note, the term *profile* refers to the data generated by the profiler. In the case of a statistical profiler, a profile is a sequence of samples.

Example Report

To help illustrate how perf report presents the profile data, we will profile a small C program (see Appendix TODO) that has a sufficiently complex call stack to demonstrate the nature of sampling. We can run perf record, with the same arguments as before, but we also pass the program to profile with `perf record -g -e cycles:u -c 10000 simple_prog`.

Interpreting the Report

Once the program has finished executing, we can run `perf report`. This displays the output depicted in Figure 3.2:


cameron@cameron-thinkpad-x1-carbon: ~/devel/unsw-honours-thesis/thesis-a/perf-demo
cameron@cameron-thinkpad-x1-carbon: ~/devel/unsw-honours-thesis/thesis-a/perf-demo

```
Samples: 781K of event 'cycles:u', Event count (approx.): 7815840000
  Children      Self  Command    Shared Object     Symbol
+   39.85%     0.00%  simpl_prog  libc-2.31.so      [.] __libc_start_main
+   37.24%     1.94%  simpl_prog  simpl_prog        [.] main
+   35.09%     0.00%  simpl_prog  [unknown]         [.] 0x0000558d0445f2a0
+   14.85%    14.85%  simpl_prog  libc-2.31.so      [.] _IO_puts
+   13.92%     0.00%  simpl_prog  [unknown]         [.] 0x00000000000a3730
+   10.67%     1.89%  simpl_prog  simpl_prog        [.] func_c
+   10.20%     3.94%  simpl_prog  libc-2.31.so      [.] __GI___libc_write
+    8.93%     8.93%  simpl_prog  libc-2.31.so      [.] _IO_do_write@@GLIBC_2.2.5
+    8.92%     8.92%  simpl_prog  libc-2.31.so      [.] _IO_file_write@@GLIBC_2.2.5
+    8.06%     8.06%  simpl_prog  libc-2.31.so      [.] __strlen_avx2
+    8.05%     1.45%  simpl_prog  simpl_prog        [.] func_b
+    7.42%     7.42%  simpl_prog  libc-2.31.so      [.] __random_r
+    6.88%     0.89%  simpl_prog  simpl_prog        [.] func_a
+    6.82%     6.82%  simpl_prog  libc-2.31.so      [.] _IO_file_xsputn@@GLIBC_2.2.5
+    6.22%     6.22%  simpl_prog  [unknown]         [k] 0xffffffffbc000000
+    4.82%     4.82%  simpl_prog  libc-2.31.so      [.] __memmove_avx_unaligned_erms
+    4.61%     4.61%  simpl_prog  libc-2.31.so      [.] _IO_file_overflow@@GLIBC_2.2.5
+    4.05%     4.05%  simpl_prog  libc-2.31.so      [.] __random
+    3.41%     3.41%  simpl_prog  libc-2.31.so      [.] rand
+    2.80%     0.82%  simpl_prog  simpl_prog        [.] func_e
+    2.66%     0.00%  simpl_prog  [unknown]         [.] 0000000000000000
+    2.62%     1.17%  simpl_prog  simpl_prog        [.] func_f
+    2.39%     2.39%  simpl_prog  libc-2.31.so      [.] __overflow
+    2.27%     0.00%  simpl_prog  simpl_prog        [.] 0x0000558d025b80f0
+    2.27%     2.27%  simpl_prog  simpl_prog        [.] 0x00000000000010f0
+    1.37%     0.37%  simpl_prog  simpl_prog        [.] func_d
+    0.94%     0.94%  simpl_prog  libc-2.31.so      [.] __mempcpy_avx_unaligned_erms
+    0.85%     0.85%  simpl_prog  simpl_prog        [.] func_i
Press '?' for help on key bindings
```

Figure 3.2: Example output from `perf report`.

By default, the table in the report is separated into five columns:

- *Children* is the percentage of overall samples that were collected exclusively within a descendant function.
- *Self* is the percentage of overall samples that were collected within the function itself (i.e ignoring descendant functions).
- *Command* is the process the samples were collected from.
- *Shared Object* displays the name of the Executable and Linkable Format (ELF) image where the samples same from.
- *Symbol* displays the name of the function that was executing when the sample was taken.

In the example report, there are a number of notable points:

1. The "Children" and "Self" columns are percentage values, and do not represent time, but rather percentage of overall samples. If the total execution time for the profile is known, the cumulative execution time for a given function can be approximated using the number of samples where the function appears.
2. The "Shared Object" column refers to ELF images other than simple_prog. This is because there are calls to libc functions, namely `time`, `srand`, `getpid` and `printf` which are still executing within the simple_prog process.
3. Instances where the value for "Shared Object" show [unknown] refer to dynamic shared objects (DSO), where the object name could not be resolved.
4. The cases where the value for "Shared Object" is simple_prog, but the corresponding symbol is a raw address, is due to the profiler not being able to find an entry in the ELF image for that particular address.
5. Lastly, if we were to count all (CPU) cycles, instead of only user cycles, we would expect to see kernel symbols also appearing in the report.

3.2.4 Event Groupings

The PCL subsystem allows events to be grouped. A group of events are atomically assigned to PMU counters, in that at any given point in time, either they are all actively being counted by the PMU, or none are. This ensures that each event in the group was active for the identical number of CPU instructions, which is required in order to correctly compute ratios being two events. For example, number of L1 cache misses per instruction software increment.

```
int syscall(SYS_perf_event_open, struct perf_event_attr *attr,
            pid_t pid, int cpu, int group_fd, unsigned long flags);
```

Listing 7: User-level syscall to interface with PCL subsystem

3.2.5 PCL User-level Interface

On Linux, the `perf` utility somehow needs to interact with the PCL subsystem, which it does so via the `perf_events_open` syscall in Listing 7.

The arguments for the syscall are:

- `SYS_perf_event_open` is an integer defined in `<sys/syscall.h>` that uniquely identifies the syscall within the kernel.
- `struct perf_event_attr *attr` is a pointer to the `struct` that provides detailed configuration information for the event to be measured.
- `pid` is the process ID to measure.
- `cpu` is the CPU to measure.
- `group_fd` allows event groups to be created (see Section 3.2.4).
- `flags` is formed by ORing a number of flags that prominently alter how the returned file descriptor should behave.

On success, the syscall returns a file descriptor (fd), which directly corresponds to a single event being measured. Events can then be enabled or disabled via the `ioctrl` or `prctrl` system calls. The PCL subsystem supports both counting and sampling events from user-space (see Section 2.2.4 for an overview on the difference).

Counting Events

Once the file descriptor (`fd`) for an event has been opened, the current value can obtained by reading from the `fd`. This typically is achieved via the `read` syscall, where the data read into the provided buffer (if the event is not grouped) will be an instance of `struct read_format` (defined in Listing 8).

```
struct read_format {
    u64 value;          /* The value of the event */
    u64 time_enabled;   /* if PERF_FORMAT_TOTAL_TIME_ENABLED */
    u64 time_running;   /* if PERF_FORMAT_TOTAL_TIME_RUNNING */
    u64 id;             /* if PERF_FORMAT_ID */
};
```

Listing 8: Data returned from reading the `fd` for an event.

Sampling Events

Sampling an event differs from counting an event, in that the kernel will notify userspace every N events, where N is determined by the sampling frequency. The mechanism for notifying is via either a signal sent to the user process or by polling the file descriptor (e.g using `epoll`). Instead of reading directly from the file descriptor, the kernel will write records to a shared ring-buffer, which can then be accessed in userspace via calls to `mmap`. A record, which is perhaps confusingly referred to as an asynchronous event in the man pages, can be one of 21 types. Four common record types are defined below:

- `PERF_RECORD_MMAP` indicates the `PROT_EXEC` mappings.
- `PERF_RECORD_COMM` indicates a change in the process name.
- `PERF_RECORD_LOST_SAMPLES` indicates some number of samples may have been lost.
- `PERF_RECORD_SAMPLE` indicates a sample, which may contain the program counter, call stack, cpu, recent branch stack, register set etc.

3.2.6 The perf File Format

The `perf record` command, by default, will write the profile data out to file called `perf.data`, which is then consumed by other perf tools (e.g `perf report`). In order to ensure interoperability, there is a perf file format which specifies the layout of data within the `perf.data` file, and is designed in such a way that it provides forwards and backwards compatibility [23].

The CERN openlab technical report from 2011 [15] covers the perf file format in depth. The format, as of April 2022, still largely resembles the format described in the report and therefore provides an excellent reference explanation, complimenting the documentation within the Linux repo [23].

The file format begins with a header, and defines an offset into the file for sections that store the event types that make up the profile, the attributes for each sampled event, and the recorded samples for each event. The file format essentially defines a way to serialise the event configuration from each `perf_event_open` syscall, as well as each sample that is placed into the shared buffer (see Figure 3.3).

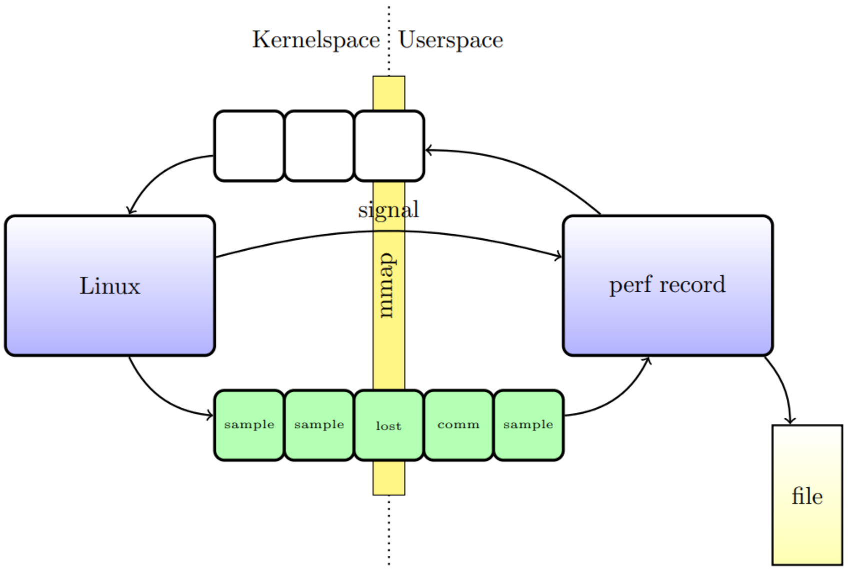


Figure 3.3: Overview of how the PCL subsystem (labelled as Linux) and the perf utility interact. Taken directly from Figure 2 in the CERN openlab, titled “perf file format” [15].

Chapter 4

Approach

The thesis problem statement of developing a profiling framework for seL4 is well defined since we have a comparative tool on Linux in the form of the `perf_event_open` syscall that sets expectations around the desired functionality. However, there is still considerable room for how exactly the profiler should be implemented on seL4. To begin, we will look at some of the limitations of statistical profiling, followed a set of design goals to guide the implementation, and then we will discuss a potential solution.

4.1 Statistical Profiling Limitations

Statistical profiling offers a low overhead approach to profiling, however there are a number of limitations or scenarios where it is not suitable:

- When 100% instruction accurate profiles are required. Due to the latency of the polling interrupts, the address read from the PC may refer to a more recently executed instruction. An illuminating example of this mis-attribution can be seen in a loop where the last instruction is relatively expensive, but the time is attributed to incrementing the loop variable [14].
- When the function being profiled is called infrequently. If there are not a sufficient number samples, the profiler may not be able to provide any useful insight into its runtime behaviour.

- When no disturbances to the system whatsoever can be tolerated. Sampling requires hardware interrupts to be handled, which may not be suitable for real time applications.
- When an interstitial profiling API is required. Due to the nature of sampling, it is imperative that work performed during each profiling interrupt is minimal, to reduce the overhead on the system, and therefore calling user-defined functions during the interrupt handling would not feasible.

4.2 Design Goals

1. **Performance**. The profiler should not impose a significant overhead on the system. Further, the overhead should be directly proportional to the sampling frequency selected by the user.
2. **Extensible**. The design should be open for extension, such that new PMU events or configuration options can be added without refactoring the existing codebase.
3. **Portable**. The implementation should not be dependent on any platform and architecture specific details.
4. **Configurable**. The design should provide an expressive API to user-space, such that the profiler can be configured to sample select events.
5. **Verification**. seL4 is a formally verified microkernel and so any extension to the kernel, requires that it be verifiable. While full verification is outside the scope of this thesis, the implementation footprint should be as minimal as possible, and amenable to verification.

4.3 Requirements

Following on from the design goals, we can now list concrete requirements for the implementation of a statistical profiler.

Requirement 1. Supports both statistical and event-based profiling.
The additional expressivity that arises when supporting both statistical and event-based profiling, is evident with the PCL subsystem (see Section 3.2), as it allows the user to configure sampling based on (microarchitecture) events. This means that users are not constrained to only sampling based on CPU cycles.

Requirement 2. Supports non-destructive profiling.
The term *non-destructive* refers to a type of profiling which does not require the program to be recompiled in order to be profiled.

Requirement 3. Symbol resolution.
The profiler should resolve raw addresses, obtained as part of the sample, to their corresponding symbol, as it is critical to the usability of the tool. This includes resolving symbols for the kernel, shared libraries, and user-space programs.

Requirement 4. Supports system-wide profiling.
The ability to monitor the whole system allows mode switches between kernel-mode and user-mode to be observed. To profile IPC for example, requires system-wide profiling be supported.

Requirement 5. Transport independent.
Unlike the `perf` utility on Linux, we cannot assume a filesystem will be present to write the profile data to disk. Therefore, where the profile data is written should be provided by the user. This decouples the responsibility of collecting the data from storing the data.

Requirement 6. Addresses lockstep sampling.
Lockstep sampling occurs when the profiling samples occur at the same frequency as a loop in the program. The result is that each sample is taken at a similar location in the loop, therefore giving the appearance that it is the most common operating, and possibly a performance bottleneck.

Requirement 7. Interoperability with `perf report`.
We should not reinvent the wheel, but instead look to leverage the existing ecosystem of tooling built around the PCL subsystem.

4.4 Targeted Platform

All major platforms support performance monitoring (see Section 2.3). The interface for accessing the PMU counters differs on each platform. While libsel4bench (see Section 3.1.3) defines an interface for accessing and configuring the PMU counters [28] which abstracts over platform specifics, to begin we want to focus on a single platform and architecture. Considering that ARM is the most supported platform on seL4 [29], we should start with ARM, and in particular ARMv8-A (A refers to the A-profile, which is targeted towards operating systems).

4.5 Design

At a conceptual level, the PCL subsystem is a suitable reference approach for abstracting over the PMU counters. It is also an example of a working implementation. However, there are a number of implementation specific details that limit its usefulness in the context of seL4:

- **Requires a filesystem**. The `perf record` command assumes that the output will be written to a file. There is no requirement for an seL4 system to have a mounted filesystem, and therefore we cannot make this assumption. However, the `perf_event_open` Linux syscall (see Section 3.2.5) writes sample data to a shared memory, which is only possible on seL4.
- **Inherit complexity**. Linux is deployed everywhere, and therefore the PCL subsystem was designed and built to support all possible uses cases, since it could not predict how users might want use the subsystem. For seL4, we can limit the inherit complexity by focusing only on the use cases relevant to seL4 users.
- **Interrupt handling**. The PCL subsystem will handle the interrupt and generate a sample within the kernel itself. For seL4, there is no evidence that this needs to be happen with the kernel, and therefore should be moved to userspace. As a consequence, the shared ring-buffer between the user thread and the kernel not required for seL4.
- **POSIX/Linux APIs**. The PCL subsystem depends on either POSIX or Linux specific APIs. seL4 does not implement a POSIX/UNIX like interface and therefore either syscalls in the PCL subsystem need to be mapped to corresponding syscalls in seL4, or another approach needs to be devised where no mapping exists.

Thus our proposed design looks to borrow concepts from PLC where possible, but is congruent with an seL4 environment. The design is comprised of the following components:

- *Profiling server* is responsible for configuring the PMU counters, handling interrupts from the PMU and generating samples. If a thread wants to start or stop profiling, it does so by sending a request to the profiling server.
- *Sample ring-buffer* is a ring-buffer that stores the sample generated during each PMU interrupt.
- *Debug seL4 kernel* is required since on ARM, user-mode access to PMU counters must be enabled in kernel-mode. Note, there already exists the ability to compile a debug version of seL4.

When the ring-buffer within the profiling server reaches capacity, the buffer needs to be flushed. The server does not make any assumptions as to where the data in the buffer should be written, and instead invokes a capability supplied from the requesting thread to flush the buffer. To help illustrate how this would work in practice, we will walk through an scenario where the profiler is running on an embedded device, and the profile data is being sent to a Linux machine via the network (Figure 4.1).

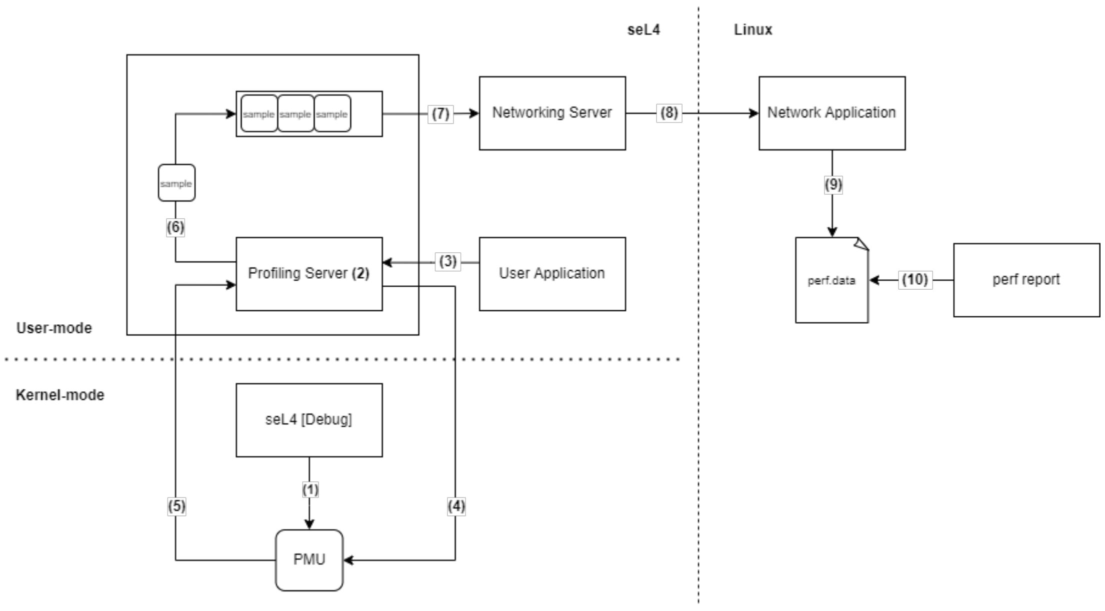


Figure 4.1: Design overview for a profiler in seL4, where the transport layer been configured such that write measurement data is sent via the network and stored on the receiving Linux machine.

1. The debug seL4 kernel enables user-mode access to the PMU counters.
2. The profiling server is provided a capability in order to service interrupts from the PMU.

3. The user application that would like to start profiling sends a request to the server (via IPC).

4. The profiling server then configures the PMU via MSRs (see Section 2.2.5).

5. Once the PMU counter overflows, sends an interrupt, which is ultimately handled by the profiling server.

6. The profiling server reads the appropriate PMU counter and generates a sample which is stored in the ring-buffer.

7. when the ring-buffer reaches capacity, the capability provided by the user application was a capability to send a packet via the networking server. the profiling server then invokes the capability, which results in the samples in the buffer being transferred across the network.

8. A network application on the Linux machine receives the profile data, which conforms to the perf file format (see Section 3.2.6), and is written out to a file on disk.

9. Lastly, the `perf record` command on Linux is used at a later stage to display the profile data.

Chapter 5

Plan

5.1 Stages

Bibliography

[1] Resource Allocation and Scheduling Group (RASG) @SCI-Pitt. How to use arm performance monitoring units(pmu) in armv7.

[2] ARM. About the pmu.

[3] ARM. About the pmu.

[4] ARM. About the pmu.

[5] ARM. Arm architecture profiles.

[6] ARM. Arm architecture reference manual armv7-a and armv7-r edition.

[7] ARM. Armv8.1-m performance monitoring user guide.

[8] ARM. Instruction set architecture (isa).

[9] ARM. Mcr, mcr2, mcrr, and mcrr2.

[10] ARM. Mrc, mrc2, mrrc and mrrc2.

[11] ARM. Performance monitor registers.

[12] David Both. The central processing unit (cpu): Its components and functionality.

[13] BSD. gprof man pages.

[14] die.net. setitimer(2) - linux man page.

[15] Urs Fassler. perf file format.

[16] Jay Fenlason. Implementation of profiling.

[17] RISC-V Foundation. Counters.

[18] Brendan Gregg. perf examples.

[19] Red Hat. Performance counters for linux (pcl) tools and perf.

[20] Germot Heiser. Microkernels: Reducing the trusted computing base.

[21] Intel. Intel® 64 and ia-32 architectures developer's manual: Vol. 3b.

[22] Linux. perf(1) - linux manual page.

[23] linux. perf.data format.

[24] GCC project. 6.47.2 extended asm - assembler instructions with c expression operands.

[25] GCC project. gcc man pages.

[26] seL4. About sel4.

[27] seL4. sel4bench.

[28] seL4. sel4bench.h.

[29] seL4. Supported platforms.

[30] spec. Spec cpu® 2017.

[31] Dean Takahashi. Risc-v grows globally as an alternative to arm and its license fees.

[32] Jim Turley. The two percent solution.

[33] Vince Weaver. The unofficial linux perf events web-page.

Appendix 1

This section contains the options for the UNSW thesis class; and layout specifications used by this thesis.

A.1 Options

The standard thesis class options provided are:

undergrad	default
hdr	
11pt	default
12pt	
oneside	default for HDR theses
twoside	default for undergraduate theses
draft	(prints DRAFT on title page and in footer and omits pictures)
final	default
doublespacing	default
singlespacing	(only for use while drafting)

A.2 Margins

The standard margins for theses in Engineering are as follows:

	U'grad	HDR
`\oddsidemargin`	40 mm	40 mm
`\evensidemargin`	25 mm	20 mm
`\topmargin`	25 mm	30 mm
`\headheight`	40 mm	40 mm
`\headsep`	40 mm	40 mm
`\footskip`	15 mm	15 mm
`\botmargin`	20 mm	20 mm

A.3 Page Headers

A.3.1 Undergraduate Theses

For undergraduate theses, the page header for odd numbers pages in the body of the document is:

Author's Name	*The title of the thesis*

and on even pages is:

The title of the thesis	Author's Name

These headers are printed on all mainmatter and backmatter pages, including the first page of chapters or appendices.

A.3.2 Higher Degree Research Theses

For postgraduate theses, the page header for the body of the document is:

The title of the chapter or appendix

This header is printed on all mainmatter and backmatter pages, except for the first page of chapters or appendices.

A.4 Page Footers

For all theses, the page footer consists of a centred page number. In the frontmatter, the page number is in roman numerals. In the mainmatter and backmatter sections, the page number is in arabic numerals. Page numbers restart from 1 at the start of the mainmatter section.

If the **draft** document option has been selected, then a "Draft" message is also inserted into the footer, as in:

	14	**Draft:** April 27, 2022

or, on even numbered pages in two-sided mode:

Draft: April 27, 2022	14	

A.5 Double Spacing

Double spacing (actualy 1.5 spacing) is used for the mainmatter section, except for footnotes and the text for figures and table.

Single spacing is used in the frontmatter and backmatter sections.

If it is necessary to switch between single-spacing and double-spacing, the commands `\ssp` and `\dsp` can be used; or there is a `sspacing` environment to invoke single spacing and a `spacing` environment to invoke double spacing if double spacing is used for the document (otherwise it leaves it in single spacing). Note that switching to single spacing should only be done within the spirit of this thesis class, otherwise it may breach UNSW thesis format guidelines.

A.6 Files

This description and sample of the UNSW Thesis LaTeX class consists of a number of files:

unswthesis.cls	the thesis class file itself
crest.pdf	the UNSW coat of arms, used by `pdflatex`
crest.eps	the UNSW coat of arms, used by `latex` + `dvips`
dissertation-sheet.tex	formal information required by HDR theses
pubs.bib	reference details for use in the bibliography
report-a.tex	the main file for the thesis

The file report-a.tex is the main file for the current document (in use, its name should be changed to something more meaningful). It presents the structure of the thesis, then includes a number of separate files for the various content sections. While including separate files is not essential (it could all be in one file), using multiple files is useful for organising complex work.

This sample thesis is typical of many theses; however, new authors should consult with their supervisors and exercise judgement.

The included files used by this sample thesis are:

definitions.tex	mywork.tex
abstract.tex	evaluation.tex
acknowledgements.tex	conclusion.tex
abbreviations.tex	appendix1.tex
introduction.tex	appendix2.tex
background.tex	

These are typical; however the concepts and names (and obviously content) of the files making up the matter of the thesis will differ between theses.

Appendix 2

This section contains scads of supplimentary data.

B.1 Data

Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data.

Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data.

Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data.

Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data.

Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data.

Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data.

Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data.

Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data.

Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data.

Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data.

Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data. Heaps and heaps and heaps and heaps and heaps and heaps of data.