



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
Supervisor: A/Prof. Person

Student ID: z5149731
Topic ID: XX00

Abstract

This document describes the requirements to theses submitted for the Bachelor of Engineering in Computer Engineering degree at the School of Computer Science and Engineering. Requirements described are that of both of context and layout of the theses. The document is written using the L^AT_EX template provided by the school.

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 T_EX, for typesetting technical (and non-technical) material with elegance and clarity; and from Leslie Lamport who contributed L^AT_EX, which makes T_EX usable by mortal engineers.

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

Abbreviations

BE Bachelor of Engineering

L^AT_EX A document preparation computer program

PhD Doctor of Philosophy

Contents

1	Introduction	1
2	Background	2
2.1	Profiling	2
2.1.1	CPU Profiling	2
2.1.2	Types of CPU Profilers	3
2.1.3	Statistical Profiling Overview	4
2.1.4	Statistical Profiling Limitations	5
2.2	Performance Monitoring	5
2.2.1	Microarchitecture Fundamentals	6
2.2.2	Performance Monitoring Unit (PMU)	6
2.2.3	Performance Counters	7
2.2.4	Sampling vs Counting	7
2.2.5	Programming the PMU	7
2.3	Platform Support	10
2.3.1	x86 Systems	10
2.3.2	ARM Systems	11
2.3.3	RISC-V Systems	12

3	Related Work	13
3.1	seL4 Benchmarking	13
3.1.1	Benchmarking	13
3.1.2	sel4bench	13
3.1.3	libsel4bench Overview	14
3.1.4	libsel4bench Limitations [TODO: Change title]	14
3.2	Performance Counters for Linux (PCL)	15
3.2.1	The perf utility	15
3.2.2	perf record	15
3.2.3	perf report	16
3.2.4	Event Groupings	17
3.2.5	PCL User-level Interface	18
3.2.6	The perf File Format	19
4	Style and Submission Requirements	21
4.1	Format	21
4.2	Other physical appearance	22
4.3	Submission	22
5	Content Requirements	24
5.1	Structure	24
5.2	Style of writing	24
5.3	Documentation	25
6	Evaluation	26
6.1	Results	26
6.2	Discussion	26

7 Conclusion	27
7.1 Future Work	27
Bibliography	28
Appendix 1	30
A.1 Options	30
A.2 Margins	30
A.3 Page Headers	31
A.3.1 Undergraduate Theses	31
A.3.2 Higher Degree Research Theses	31
A.4 Page Footers	31
A.5 Double Spacing	32
A.6 Files	32
Appendix 2	34
B.1 Data	34

List of Figures

List of Tables

Chapter 1

Introduction

Having a set of clear requirements to their thesis is important to student finalising their BE, or other, degree. Such requirements are both in relation to the physical appearance of the thesis, as well as the writing style and organisation. The present document tries to concisely state the theses requirements while appearing in layout and structure as a thesis

In the context of seL4, because most of the system services do not reside in the kernel, but instead are hoisted into user-space, it is critical that seL4 developers have sufficient tooling such that they can diagnose performance issues. (Could talk about the fact that there is limited support for profiling in the kernel, and so it is no use to developers in user-space)

Chapter 2 explains the background for this document. Chapter 4 states the style and submission related requirements to theses submitted at the school. Chapter 5 explains content related requirements to theses. Chapter 6 evaluates the thesis requirements template. Finally, Chapter 7 draws up conclusions and suggest ways to further improve the thesis requirements template.

Chapter 2

Background

[TODO: Tie the thesis problem statement subsection with the background chapter]

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 [12]) 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` [28].

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 a later section [TODO: Add reference].

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

Suitability for seL4

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 executed¹. 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.

[TODO: Add diagram of these concepts]

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.

[TODO: Add diagram of sampling a thread]

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**,

¹For 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.

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 we will cover shortly. [TODO: Add link]

2.1.4 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 be feasible.

2.2 Performance Monitoring

[TODO: Introduce this section]

2.2.1 Microarchitecture Fundamentals

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 x 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.

[TODO: Include diagram of hypothetical CPU]

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 address 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 [TODO: Find Armv8.1-M that supports branch prediction]). 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 processor² that dictate how the PMU should operate.

Fixed and Programmable Counters

[TODO: Fill this out]

2.2.4 Sampling vs Counting

[TODO: Fill this out]

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

²Typically 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.

may take multiple CPU cycles to execute. This event is also known as instruction retired on other platforms.

Cortex-A15

The Cortex-A15 is a 32-bit multi-core processor, which implements the ARMv7-A architecture, and is well supported by seL4 [31]. 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. 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]. Often, we want

```

#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 PMOVSER_EN_PMCCNTR            (0x1 << 31)
#define PMOVSER_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"(pmcr_config));

```

Listing 2: Writing to the PMCR.

to use performance counters within a C program, and therefore we are using inline assembly, in particular Extended Asm [27].

Now that counters are enabled, 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.

Finally, we need to clear the overflow bit for both of the counters.

```

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

```

```

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

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));

```

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).

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.

The current number of software increment events will now be in the variable `counter_value`.

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

```

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));

```

requirements, but in return offer higher performance over some RISC architectures for certain kinds of workloads, such as: [TODO: Find example]. 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 [22]:

- *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 [35].

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 micro-controllers 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 commercial viability of RISC-V processors, replacing systems that were traditionally dominated by ARM [TODO: Find a citation].

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

[TODO: Tie in the background chapter and describe how it relates to this chapter]

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 [33].

3.1.2 sel4bench

The repo sel4bench provides multiple applications for benchmarking different paths in the kernel [29]. 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 compatible¹.

3.1.3 libsel4bench Overview

Interfacing with the PMU directly via platform dependent assembly instructions and 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) [30].

3.1.4 libsel4bench Limitations [TODO: Change title]

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 defined the difference between sampling and counting to be [TODO: Fill this out]. 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

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

switches control back to the profiler such that it can snapshot the running threads state (i.e PC, call stack, register set).

[TODO: Find another requirement for libsel4bench]

3.2 Performance Counters for Linux (PCL)

[TODO: Motivate why need to review how Linux solves this]

The PCL is a kernel-based subsystem that provides a framework for collecting and analysing performance data [21]. 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 [36] (most recent version is 5.17.4, released 20 April 2022).

3.2.1 The perf utility

The perf utility is the Command Line Interface (CLI) to the PCL subsystem [23]. It is a high level interface 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 ourselves 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

[Todo: Create simpl_prog.c in the appendix and link back here]

To help illustrate how perf report presents the profile data, we will profile a small C program [TODO: link here] 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 following output:

[TODO: Insert output from perf report]

By default, the table 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 came 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

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

Listing 3: User-level syscall to interface with PCL subsystem []

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.

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 3.2.5[TODO: Fix referencing]

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 `ioctl` or `prctl` system calls. The PCL subsystem supports both counting and sampling events from userspace (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 be obtained by reading from the `fd`. This typically is achieved via the `read` syscall, where

```

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 4: Data returned from reading the `fd` for an event

the data read into the provided buffer (if the event is not grouped) will be an instance of `struct read_format`, as seen in Listing [TODO].

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. 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 [24].

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 [24].

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.

[TODO: Insert figure from CERN openlab]

Chapter 4

Style and Submission Requirements

Requirements for other parts of the thesis work can be found on the school web-pages [26]. The requirements below are for the written thesis only.

4.1 Format

The following format specifications must be adhered to for your thesis (the L^AT_EX template available from the school ensures this):

1. The thesis must be written on *A4 size paper*.
2. The thesis must be typed or prepared using a *word-processor*.
 - For Undergraduate theses, you are encouraged to use both sides of the paper.
 - For Higher Degree Research theses, your submitted thesis must be printed single-sided.
3. *Margins* on all sides must be no less than 20 mm (before binding).
4. *1.5 line spacing* (about 8 mm per line) must be used.

5. All sheets must be *numbered*. The main body of the thesis must be numbered consecutively from beginning to end. Other sections must either be included or have their own logical numbering system.
6. The *title page* must contain the following information:
 - (a) University and School names.
 - (b) Title of Thesis/Project.
 - (c) Name of Author and student ID.
 - (d) The degree the thesis is submitted for.
 - (e) Submission date (month and year).
 - (f) Supervisor's name (for undergraduate theses).
7. After the body of the thesis, the thesis *must* contain a Bibliography or References list as appropriate.

Authors should confer with their supervisors and School about the style of their bibliography, as this varies between disciplines.

4.2 Other physical appearance

Other requirements to the physical appearance of your theses are:

1. *Graphs, diagrams and photographs* should be inserted as close as possible to their *first reference* in the text. Rotated graphs etc are to be arranged so as to be conveniently read, with the bottom edge to the outside of the page. *Graphs and diagrams must be legible!*
2. *Supplementary material (for example CFD animations)* may be submitted either online or via external drive, and must be referred to within the text. The text should make sense without the supplementary material available.

4.3 Submission

Finally, here are some requirements to the submission procedure.

1. The *author* of the thesis is *responsible* for the preparation of the thesis before the deadline, proofreading the typescript and having corrections made as necessary.
2. For undergraduate theses, there is a *page limit* of 50 pages for the main body of the thesis.

Chapter 5

Content Requirements

Students should consult the literature (e.g. [32, 34, 13, 19]) and other resources for material on how to write a good thesis. The present document is only a very brief introduction as to what is expected.

5.1 Structure

Most theses are structured very much like the present document. The main part of the thesis can be structured in many different ways, however, but must contain: a *problem definition*; *theory* and *considerations* on how to solve the problem; a description of the *solution method* (dimensioning, construction, etc.); presentation of *results* (measurements, simulations, etc.); a *discussion* of the results (validity, deviations, comparison with previous solutions, etc.); and finally the *conclusions*.

5.2 Style of writing

1. Audience: The thesis must be addressed to engineers at the same level as the student but without the special knowledge gained during the thesis work. Such a third-person must be able to reconstruct the results on the basis of the thesis alone.

2. Every used concept/symbol/abbreviation which is not widely known must be *defined*. The wording should be *short* and *concise*. Readable(!) *figures* and *graphs* enhances comprehensibility.
3. Units. *SI units* must be used.

5.3 Documentation

1. The work must be well documented; i.e. enclosed must be the *complete schematics* of designed electronic circuits/test set-ups and/or a *program listing*, and/or etc. Documentation of *simulation results* and/or *measurement results* likewise.
2. References: For every declaration/equation/method/etc., which is not widely known, a *reference to the literature* must be given (or a ‘proof’ if it is the authors own work). In case material is copied verbatim, quotes must be used. This is also the case when referring to partners work in the case of a Group Thesis.
3. Plagiarism: Failure to give proper references to the literature is *plagiarism*. Plagiarism is considered serious offence and severe penalties may apply.

Chapter 6

Evaluation

This chapter is mainly provided for the purpose of showing a typical thesis structure. There are no more thesis requirements described.

6.1 Results

The result of this work is the present document, being both a L^AT_EX template and a thesis requirement specification.

6.2 Discussion

The Dual function of this document somewhat de-emphasises the primary purpose of the document, namely the thesis requirements. It would be better, if these could be stated on a few concise pages (cf Appendix 1, p30).

Chapter 7

Conclusion

A thesis requirements/template document has been created. This serves the dual purposes of giving students specific requirements to their theses — both style and content related — while providing a typical thesis structure in a \LaTeX template.

7.1 Future Work

Extract the requirements from the template in order to have very concise requirements.

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] BSD. gprof man pages.
- [13] Bruce M. Cooper. *Writing Technical Reports*. Penguin Books, Middlesex, 1964.
- [14] die.net. setitimer(2) - linux man page.
- [15] Urs Fessler. perf file format.
- [16] Jay Fenlason. Implementation of profiling.
- [17] RISC-V Foundation. Counters.
- [18] Brendan Gregg. perf examples.
- [19] GRS. Thesis format guide: A guide for candidates preparing to submit their thesis for examination. https://research.unsw.edu.au/document/thesis_format_guide.pdf, accessed 14/04/2015, 2014. Graduate Research School, UNSW.

- [20] Tawfique Hasan, Torsten Lehmann, and Chee Yee Kwok. A 5V charge pump in a standard 1.8V 0.8 μ m CMOS process. In *IEEE International Symposium on Circuits and Systems*, pages 1899–1902, Kobe, Japan, 2005.
- [21] Red Hat. Performance counters for linux (pcl) tools and perf.
- [22] Intel. Intel® 64 and ia-32 architectures developer’s manual: Vol. 3b.
- [23] Linux. perf(1) - linux manual page.
- [24] linux. perf.data format.
- [25] Jannik Hammel Nielsen and Torsten Lehmann. An implantable CMOS amplifier for nerve signals. *Analog Integrated Circuits and Signal Processing*, 36(1–2):153–164, July-August 2003.
- [26] Saeid Nooshabadi. Bachelor of engineering thesis and project: timetable and notes for students. <http://scoff.ee.unsw.edu.au/document/thesis/thnotes2.pdf>, accessed 14/11/2005, 2005. School of El. Eng. and Telecom., UNSW.
- [27] GCC project. 6.47.2 extended asm - assembler instructions with c expression operands.
- [28] GCC project. gcc man pages.
- [29] seL4. sel4bench.
- [30] seL4. sel4bench.h.
- [31] seL4. Supported platforms.
- [32] Charles H. Sides. *How to Write & Present Technical Information*. Cambridge University Press, Cambridge, 3rd edition, 1999.
- [33] spec. Spec cpu® 2017.
- [34] William Strunk Jr. and E. B. White. *The Elements of Style*. Macmillan Publishing Co., New York, 3rd edition, 1979.
- [35] Jim Turley. The two percent solution.
- [36] 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
<code>\oddsidemargin</code>	40 mm	40 mm
<code>\evensidemargin</code>	25 mm	20 mm
<code>\topmargin</code>	25 mm	30 mm
<code>\headheight</code>	40 mm	40 mm
<code>\headsep</code>	40 mm	40 mm
<code>\footskip</code>	15 mm	15 mm
<code>\botmargin</code>	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	<i>The title of the thesis</i>
---------------	--------------------------------

and on even pages is:

<i>The title of the thesis</i>	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:

<i>The title of the chapter or appendix</i>

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 25, 2022
----	------------------------------

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

Draft: April 25, 2022	14
------------------------------	----

A.5 Double Spacing

Double spacing (actually 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:

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