Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

You also want to explore other security leaks? Building an easily extendable application library for security leak research.

Semester Thesis

Bruno Klopott

klopottb@ethz.ch

Computer Engineering and Networks Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:
Philipp Miedl
Prof. Dr. Lothar Thiele

August 3, 2018

Abstract

This report describes the work carried out during the semester project, which focused on transforming an existing framework for researching covert channels into an extendable library. In a covert channel attack, processes communicate by exploiting accidental side effects of computation, such as heat dissipation.

The new library provides a core framework making use of the *process networks* model of computation. The new architecture defines a clearer structure for constructing covert channel applications, and allows considerable degrees of reuse and extendability. Additionally, the library features a breadth of utilities, which speed up the development process, reduce code duplication, and improve the maintainability of the codebase. The built process is also significantly overhauled, and makes it easier to build new applications. The work addresses the shortcomings of the legacy software. Preliminary evaluation also suggests that the new framework offers satisfactory performance on par with prior work.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

"Invention of any new kind of technology is also simultaneously the invention of a new kind of accident" [Paul Virilio 1, p.39]

Computing platforms, ranging from simple micro-controllers to modern processors with an array of advanced micro-architectural elements, by virtue of the operations they process produce a variety of intentional and accidental side-effects. Along with execution characteristics they form so-called side-channels, which can be exploited to break the security assumptions of a system.

Providing a comprehensive taxonomy of side-channels can be exceedingly difficult. The research work in the field is advancing rapidly and has progressed far beyond the seminal work of Paul Kocher on power [2] and timing analysis [3] of cryptographic primitives. Nowadays even LED status indicators can be used to exfiltrate data [4, 5]. A side-channel attack can exploit persistent or transient state shared at some level of the computer architecture (e.g., a single thread, a multi-core processor package), and extract sensitive information based on timing information, a trace of discrete state changes, or information about access to shared resources. To date, many of the most dangerous side-channels exploit last-level cache inclusion policies [6, 7, 8, 9] or cache coherence protocols [10], and are able to violate the spatial confinement of processes, including Intel's and ARM's trusted computing environments [11, 12].

Covert channels are descendants of side-channel attacks. In their case an unintended communication channel is explicitly established between cooperating entities by means of an unmitigated side-channel. Thanks to the newly created transmission medium malicious applications can circumvent the isolation imposed by the hardware and the operating system. The exchange of information without the knowledge of the system users can be achieved not only through the aforementioned micro-architectural side-channels, but also through more physical media, like sensors and actuators [13, 14], device system settings [15], and more fundamental byproducts of computation, like energy dissipation [16].

A number of researchers in the field of covert and side channel analysis have openly published the source code used for their exploits. Most of the related work focuses primarily on low-level primitives, and judging by the available code the authors rarely focus on repeatability and reproducibility of their experiments. The following are examples of the related work, which is directly related or somewhat applicable to covert channels:

- *Institute of Applied Information Processing and Communications at Graz University of Technology* The academic team at the IAIK have been quite prolific in producing side-channel research. Some of the publicly available source code accompanies the papers [6, 17, 18] , and the work most relevant to covert channels "CJAG: Cache-based Jamming Agreement" [19]. Most of the publications are accompanied by rather informative README files, but usually provide very scant documentation. For example, Flush/Flush source code features 56 comments in 1316 lines of code, and CJAG has an even worse statistics of 23/1557. Some attempts were made at systematising the developed functionality in ARMageddon's `libflush` library [20].
- *Flush+Reload* A repository [21] that shows greater attention to the scientific quality of the results has been provided by Hornby [22], who builds upon the Flush+Reload last-level cache side channel attack from [7]. The source code might not be the best documented one, but it provides a suite of tools for performing and automating experiments. Moreover, unlike other projects, the author has also published all experiment data.
- *Micro-architectural channels* An evaluation of a number of existing side channels was published by Hunger *et al.* [23], who also published the associated source code [24]. The source code is more extensive then the above-mentioned work, with around 8000 lines of code. Channels are provided as separate classes with a somewhat uniform calling interface. Sender and receiver applications are spawned by simple run scripts, which evaluate a range of parameters to estimate the capacity of the covert channels.

The project presented in this report builds upon the research carried out at the *Computer Engineering Group* into covert channels on modern computing platforms. The novel communication methods exploit thermal, energy, and frequency information on desktop and mobile devices based on Intel and ARM processors. The work published to date includes one master and three semester theses [25, 26, 27, 28] as well as three more rigorous publications on the risk of thermal [29], frequency [30], and power measurements [31] and the associated covert channel capacity bounds and achievable practical throughputs. What distinguishes the prior work is not only the novelty of the covert channels, but also how systematically they were evaluated.

Over the years an extensive codebase has been accumulated. The goal of this project is to refactor and rework the legacy software such that a reusable and extendable library is created. The primary aim is to facilitate the development and evaluation of existing and new covert or side channels.

In the following report, dots on the margin indicate which particular goal or improvement a section or paragraph targets. The targets include:

- *extendability*: indicates that a particular solution facilitates extensions to the software framework;
- *functionality*: signals that the presented solution extends the functional capabilities of the existing software;
- *portability*: indicates that the implemented idea improves cross-platform compatibility of the framework;
- *quality*: indicates that the implemented changes improve the correctness of program execution and quality of the source code.
- *reproducibility*: states that the described work improves the reproducibility of experiments or the software build process.
- *reusability*: signals that the presented concept allows for greater code reuse;
- *usability*: indicates that the ease-of-use or quality-in-use is improved by the presented solution;

This report is structured as follows:

- Chapter 2 describes the functionality provided by the legacy framework and reviews the available codebase.
- Chapter 3 lists some of the sources of inspiration for the new library, provides a high-level overview of the developed framework, and describes some of the software engineering decisions made to satisfy the project requirements.
- Chapter 4 deals with the implementation of the new library and gives a fairly comprehensive overview of the available functionality as well as some technical considerations.
- In chapter 5 an evaluation of the new framework is given, both in qualitative and quantitative terms.
- Possible future work, including functional extensions and software engineering improvements, are listed in chapter 6.
- Chapter 7 concludes the report.

Chapter 2

Legacy framework

2.1 Functionality

In the course of the investigation into covert channels a number of programs have been developed. From a communication-oriented perspective these have been arranged into *source* and *sink* applications. The objective of the former is to change the state of the computing platform using a provided sequence of desired , while the task of the latter is to acquire the measurements one or more observable states. A *source* commonly has its *sink* counterpart, and both are a part of a larger data generation and processing framework. The scope of the applications is therefore quite narrow; *source* and *sink* applications are concerned only with the operations closest to the covert channel.

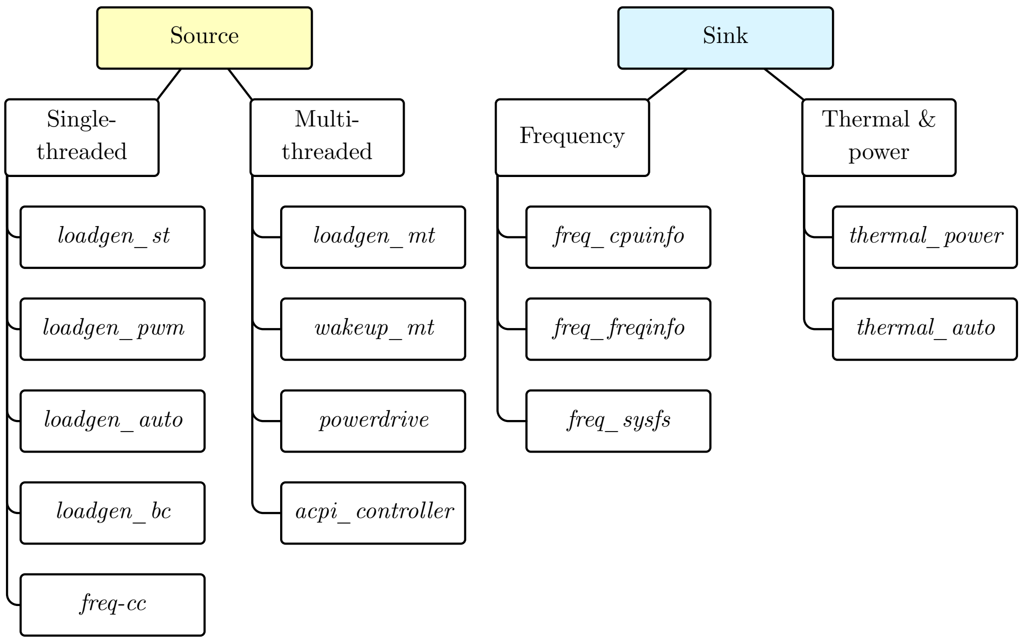

Figure 2.1: *Overview of the available applications*

The channel model is roughly shown in fig. 2.2. The *source* and *sink* applications are precended and succeded by other parts of the framework, which are responsible for producing encoded representation of some input data, and for analysing the measured state. Figure 2.1 shows an overview and an informal classification of the applications available in the legacy framework.

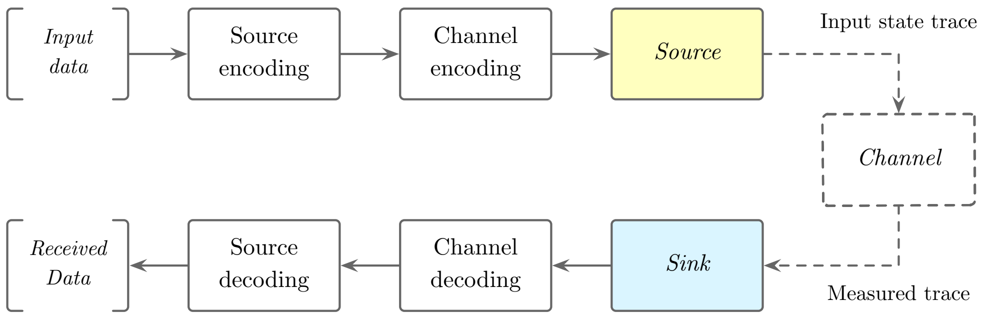

Figure 2.2: *The covert channel model*

Sources The available *source* applications are:

`loadgen_st` A single-threaded load generator that generates a binary utilisation trace (the load is either on or off) from a supplied symbol trace, which also provides the duration at which a particular state is to be maintained. This source can be pinned to a single core, and uses core 0 by default. The program also has its Android-specific version.

`loadgen_mt` A multi-threaded load generator that can be pinned to multiple cores, where core activation is instructed by an integer input symbol. All configured cores up to the provided number are activated. This source can produce a coarse multi-level utilisation trace.

`wakeup_mt` A load generator nearly identical to `loadgen_mt`, with the only difference being a single invocation of the `usleep` POSIX function [32] in the main work loop.

`loadgen_pwm` A single-threaded load generator that allows multiple utilisation levels by duty cycling the intensive work.

`loadgen_auto` A single-threaded load generator that takes a binary file instead of a sequence of symbols. A pre-processing step packetises the binary input, applies Manchester line coding, and constructs a symbol sequence much alike that used in the load generators above.

`powerdrive` A simple multi-threaded load generator that applies a configurable number of iterations of a slightly modified processor stress procedure from the `cpuburn` suite [33], with a configurable constant duty cycle.

`loadgen_bc` A single-threaded load generator that can operate in two modes. One mode provides the functionality of `loadgen_pwm`. The other mode implements a back-channel relative frequency measurement and a generator that will attempt to reach a desired utilisation level using the back-channel. The program also has its Android-specific version.

`freq-cc_conservative` An advanced single-threaded load generator that tries to cause the processor core to reach a specific operating frequency. A detailed description of the execution mechanism is presented in [30]. The main idea is to use the back-channel relative frequency measurement to either perform or suspend computation in an attempt to reach and hold a frequency target.

`freq-cc_ondemand` A load generator identical to `freq-cc_conservative`.

`acpi_controller` A multi-threaded load generator that accesses the interface to the Linux kernel's CPU clock scaling [34] exposed via the `sysfs` pseudo-filesystem [35] to directly set the CPU frequency.

Sinks The *sink* applications available in the repositories are:

`freq_cpuinfo` A multi-core meter that parses the operating frequency information reported via the Linux's `procfs` pseudo-filesystem [36] to obtain the current frequency of specific processor cores. On ARM-based platforms the meter accesses the `sysfs` interface.

`freq_freqinfo` A single-core meter that provides relative frequency readings. The program also has its Android-specific version.

`freq_sysfs` A multi-core frequency meter that accesses the Linux kernel's `sysfs` interface to processor frequency scaling.

`freq_sysfsdbg` A meter that extends the previous one with access to additional debugging interface exposed via `sysfs`.

`thermal_power` A meter that accesses processor core temperatures and energy consumption information, either via reads from model specific registers on Intel processors, or through access to sensor interfaces exposed in the `/dev` pseudo-filesystem.

`thermal_auto` A meter that is the sink counterpart of the `loadgen_auto` source, but provides no functionality other than the `thermal_power` above.

In addition to the *sinks* and *sources*, a number of general-purpose library source headers are provided, which:

- perform command line parsing (using POSIX `getopt` [37]),
- wrap access to Intel processors' time stamp counters,

- contain debug logging macros,
- define timing operations on `timespec` and `timeval` structures [38],
- define signal handlers for modifying global state variables,
- estimate the frequency of the time stamp counter,
- define workload generation functions,
- implement some of the POSIX threads functionality missing on Android.

2.2 Review

It is inevitable that source code produced by different people of various skill level over an extended period of time will accrue some technical debt, especially considering the tight schedules associated with academic work and the explorative character of the performed research. The book *Refactoring for Software Design Smells* defines some of the different dimensions of technical debt, many of which are present to various extent in the project's repositories [39, p. 2]:

- *Code debt*: Static analysis tool violations and inconsistent coding style.
- *Design debt*: "Design smells" and violations of design rules.
- *Test debt*: Lack of tests, inadequate test coverage, and improper test design.
- *Documentation debt*: No documentation for important concerns, poor documentation, and outdated documentation.

What follows is a non-exhaustive list of aspects of the software framework that could benefit from refactoring or improvement.

Documentation The comments in the source code are rather scarce, and the excessive brevity in variable naming do not help the understandability of the software (especially that different names are sometimes used in the source code and publications). Analysis with the help of a source code line counting tool [40] revealed that in the case of *sink* applications there are 459 comment lines for 3378 source code lines, and respectively 605 for 4339 lines in the case or *source* applications. It is worth noting that many of the comment lines are actually commented-out pieces of code, and provide no documentation. No file headers are given, and many "TODO" sections are not filled out.

Code duplication To start with, code duplication analysis was performed using the Simian similarity analyser [41]. While the results have to be taken with a grain of salt, they do indicate that there are large portions of code copied-and-pasted around the source files. The analysis of *source* applications resulted in 2196 duplicate lines in 102 blocks in 38 files (out of 4035 significant code lines in 59 files), and 1111 duplicate lines in 54 blocks in

14 files (out of 3248 significant code lines in 53 files) in the case of *sink* applications.

Although code duplication is not necessarily a bad thing in itself [42, 43, p. 57], some of the repeating code lines are obvious copy-paste duplicates, resulting from a lack of framework support for some commonly used procedures. In particular, the meaningful duplicates were related to signal handling, setting thread attributes, initialising and performing timing operations, and writing readings to the log file. Moreover, some functions that are present in the library have been redeclared in certain applications.

Separation of concerns Some of the important software qualities are changeability and extensibility, which are defined as "the ease with which a design fragment can be modified [and] enhanced without ripple effects", respectively [39, p. 10]. With those in mind, one quickly notices that there could be better separation of concerns in the existing software framework. For example, the `meter` and `loadgen` functions are also responsible for reading the input trace/schedule, validating input, and producing log output. Extracting such functionality would both allow greater code reuse and allow each part to be improved or enhanced without significantly affecting the other parts.

Potential misuse Some software components pose some risk of unintentional misuse. Command line parsing and passing of configuration parameters is one such component. In my personal view, while the interface is quite consistent, parameters are represented by integer indexes to arrays storing readings. The range of correct parameters are not indicated in the help text and are not necessarily apparent in the source code. Passing a wrong sequence of numerical options does not produce meaningful error messages. Maintaining the same command line options also constraints the range of possible interfaces, which is reflected in the fact that some applications manually parse them instead of using the library function. Another aspect that can overburden a potential developer is the frequent reliance on external linkage for variables holding global state and arrays.

Safety Some programs have been casually investigated by instrumenting them with sanitiser runtimes, in particular the LLVM's ThreadSanitizer [44], which detects data races. In some cases data races were detected on global flags and arrays for holding readings, indicating that barrier synchronisation primitives do not guarantee thread-safety. Only one component, the `powerdrive`, uses compiler-specific atomic operations `__sync_fetch_and_add` to modify shared variables.

Cumbersome build process The software build process relies on standard Unix Makefiles, where each application, module, and library code is provided with a separate Makefile. Individual build files rely on glob patterns for gathering include files and producing object code. Such an organisation has its limitations. First of all, the build process relies heavily on the directory hierarchy used by the project, which has to be navigated to produce particular compilation targets. As a result`make` is invoked recursively, which tends to be problematic [45].

It is difficult to globally update compilation flags and unexpected results can occur if specific subparts are given different compilation options. Moreover, all Makefile variables have global scope, increasing the chance of conflicts [46, p. 146]; currently, the compilation flags are redefined in each ancillary Makefile.

Dispersed Makefiles also hamper the cross-compilation potential and make it difficult to use modern compiler and analysis facilities. TOols such as static analysers like Cppcheck [47] or Clang-tidy [48] can help detect unsafe, unreadable, and poorly-performing code constructs. Moreover, the software relies on some non-standard preprocessor facilities for achieving multi-architecture code.

Chapter 3

Design

3.1 Inspiration

Looking at the covert channel model in fig. 2.2, what kind of software architecture could arise from such an organisation of rather self-contained components? One of the first places to look for inspiration were digital communication systems. In particular, the GNU Radio project shows how complete communication systems can be built from reusable blocks. An example of such a block is shown in fig. 3.1. Each block contains endpoints, which are connected together by edges. The blocks operate on streams of samples, with processes encapsulated in single work functions. The flow of data streams through the edges (which are built using circular buffers) is performed via asynchronous message passing and orchestrated by a scheduler [see 49].

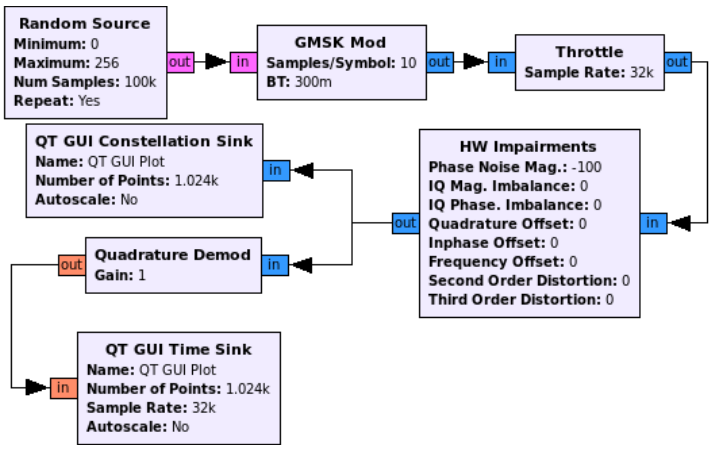

Figure 3.1: *An example of a GNU Radio flow graph*

Source: https://wiki.gnuradio.org/index.php/Guided_Tutorial_Introduction

The analogy between our channel model and the GNU Radio project prompted further investigation into more formal models of computation. The aforementioned flow graph is a variation of the topic of data-flow models. In our case, we also are dealing with individual blocks, which process information at different levels: be it a data bitstream, symbol stream, or a low-level representation of the signal, fed to the blocks closes to the channel boundary. It is important to mention that many of those parts might have different runtime requirements, in terms of synchronicity and use of resources. For example, *sinks* and *sources* need to run with fairly strict timing constraints, but the preceding and succeeding parts of the channel model can usually perform their actions asynchronously, or entirely during a pre-/post-processing step. Therefore, the more real-time MoCs (like synchronous data-flow, or synchronous reactive models) were not considered, also due to the increased complexity arising from the need of producing a static schedule before runtime.

The *Kahn Process Networks* model was chosen as the conceptual underpinning of the new framework. First of all, like all data-flow models, it allows for great extendability and reusability, because the individual nodes of the network are self contained, and communicate only through well defined interfaces. Any number of nodes can be introduced without any impact to the existing functionality. Secondly, the model is simple yet expressive. Processes in *process network* nodes can communicate only via unbounded first-in, first-out queues, but can perform computation of any degree of complexity.[1] The message passing semantics are rather straightforward, requiring blocking read accesses,[2] and non-blocking writes to the queues. Such a model can describe systems that process streams of data, which can run sequentially or concurrently. Further details about the model of computation, its extensions, and implementation requirements can be found in [51, 52, ch. 2, 53, ch. 3].

Related work

Before embarking on the process of designing and implementing own *process network* framework, existing solutions were sought. These could have either served as foundation for higher-level functionality, or as inspiration for a custom framework.

A number of potential candidates for core *process network* support were identified: Computational Process Networks (CPN) by Allen [52], RaftLib++ [54], FastFlow [55], Yapi [56], and Threading Building Blocks (TBB) by Intel [57]. Nornir [53] was also investigated, but the implementation has not been made publicly available. Task-based models were also briefly explored; a survey is given in [58].

[1] Data-flow graphs "have computational capability equivalent to a universal Turing machine" [50, p. 32].

[2] When a process attempts to read from an empty queue, it will be suspended until a token is available.

The code samples which demonstrate the interaction with each of the above-mentioned libraries are given in appendix A.1. The TBB library would have been the best candidate, since it provides the cleanest and most declarative style, at the same time having the most functionality and the widest support. However, the library does not support the 64-bit ARM architecture (ARMv8 ISA), which is used in one of our target platforms. Moreover, such a considerable dependency would possibly affect the build process and impair the portability of the new framework. The other libraries were either outdated or unmaintained, or had cumbersome or unidiomatic API design.

Although none of the reviewed libraries was chosen, a few observations were influential on the subsequent design process:

- *Templatable input and output ports.* CPN, TBB, RaftLib++, and Yapi use template parameters to indicate which data type is used by the input and output interfaces. In TBB, the parameters are passed to node declarations, e.g., `tbb::flow::function_node<int, int>`. In RaftLib++ the input/output interfaces contain template member functions, e.g. `output.addPort<int>(/*...*/)`. In Yapi and CPN, a separate interface class template is used, e.g., `Out<int>`. Such an approach seems certainly superior to casting a `void*` argument, and better than configuring data types at runtime.
- *Using strong types.* Some of the libraries seemed to be using code constructs specific to C in their C++ codebases. Most notably, some used generic pointers to `void` for "carrying" data between nodes. Such an approach provides no type safety, and might result in unexpected errors at runtime. Stronger typing can help prevent many errors at compile time and make software less ambiguous. Moreover, modern C++ provides avenues for polymorphic types and generic containers.
- *Abstracting the creation of queues from the user.* The libraries that did not require the explicit creation of queues seemed much friendlier. For example, in TBB the `tbb::flow::make_edge(node, node)` function is used, FastFlow uses consecutive calls to `pipeline.add_stage()`, and RaftLib++ provides a rather strange operator overloading (`map += producer >> consumer;`).
- *Function nodes.* Another aspect that distinguished TBB from the rest was the ability to quickly define nodes with a `function_node` class template. This feature makes it particularly useful for quick exploration and testing.
- Using pointers and references to objects and settings structures instead of string descriptions for configuring the *process network*. The libraries which rely on the latter required many more steps before the nodes and the network were usable.

3.2 Framework design

Figure 3.2 shows how the *process networks* concept could be applied to realise the covert channel model from fig. 2.2. The components of the channel model can be naturally expressed as *process network* nodes and connected with queues that can carry any data type, including heterogeneous or variable-size containers. The ellipses in the diagram indicate that other nodes could be introduced, as long as they conform to the input data type requirements of the dependent node.

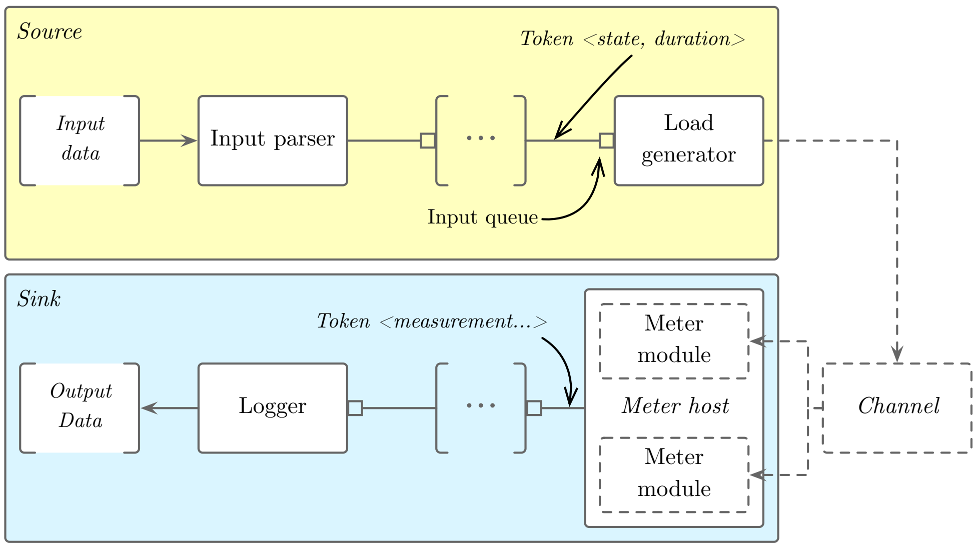

Figure 3.2: *Process Networks model applied to the covert channel*

The core framework of the new library aims to provide the building blocks necessary for using the *process network* model.

Nodes Initially, the framework will only provide support for single input and single output, which meets the needs of all existing covert channel applications. Three complementary classes of nodes are defined: consumers, producers, and processors. As the names suggest, they differ in the type of interfaces they provide, the latter combining both input and output interfaces. Similarly to GNU Radio and the frameworks listed in section 3.1, the nodes contain a single executable process.

Interfaces Since queues are generic data containers, encapsulating interfaces are required to enforce the formalism of the *process networks*. The framework will provide an abstraction of the underlying queues or communication channels, that ensures that only single-directional access is allowed.

Connectors In order to not burden the programmer with the creation of queues and bootstrapping the interfaces, the new framework will provide facilities to connect nodes together. The task of the connectors will be to verify the compatibility between nodes, create appropriate queues, and provide them to the nodes' interfaces.

Executors Executors will abstract the task of running the nodes' processes. These might in the future be used by bespoke schedulers, but initially will aim to execute the processes on threads that are scheduled by other entities, like the operating system.

In addition to the requirement imposed by *Kahn Process Networks*, to achieve execution with bounded memory the "scheduler should, if possible, execute the Kahn process network program so that only a bounded number of tokens ever accumulate on any of the communication channels" [50, p. 31]. As a result, practical implementations extend the classical semantics with blocking queue writes. However, this restriction may introduce deadlocks, called *artificial deadlocks*, which prevent progress and result from execution with queues of insufficient size. Therefore one of the important aspects of practical *process networks* implementations is deadlock detection and resolution. Providing such a mechanism is out of scope of this project; however, the reduced subset of the model almost completely obviates the need for it. Nevertheless, the library will provide a way of avoiding potential deadlocks by *extending the communication semantics*.

To allow for some deadlock avoidance, the interfaces in the framework will be extended with the ability to "time out". If the attempt to access the queue is not successful after a specified amount of time, it will be given up and indicated with a return status. If unsuccessful, there will be no side effects, i.e., the queue will not be modified. If used carefully, this mechanism will allow execution to progress without affecting the execution order (the order of queue reads and writes) and consequently its determinism.

3.3 Software design

From a software engineering perspective, the chosen design patterns and programming idioms need to allow for extendability, and provide an easy to use application programming interface (API). A good API "promotes modularization", "reduces code duplication", and makes it "easier to change the implementation" [43, p. 7]. The works that have shaped the framework's software design were *Modern C++ Design* by Alexandrescu [59], and the C++ Standard Template Library (STL).

To achieve reusability and extendability, the new framework aims to take the full advantage of the generic programming capabilities of C++. The language provides what is known as "templates", which allow creating classes and functions that can operate on different data types. For example, we may declare a function template:

```
template <typename L, typename R>
auto function(L arg_l, R arg_r);
```

Listing 3.1: *An example of a function template*

An entity with such a declaration does not have any hardcoded types. Instead, the function template is instantiated when needed (implicitly or explicitly), and the compiler generates the actual function [see 60, *temp.spec*]. For example a call to `function(static_cast<int>(1), static_cast<long>(2))` will instantiate a function implicitly with template parameters `L` and `R` being `int` and `long`. With modern C++ we can also place constraints on the template arguments. For example, the instantiation of the function template above can be restricted to arithmetic types with:

```
template <typename L, typename R,
          typename = std::enable_if_t<std::is_arithmetic<L>::value &&
                                      std::is_arithmetic<R>::value>>
auto function(L arg_l, R arg_r);
```

Listing 3.2: *An example of a conditionally enabled function template*

Templates are even more powerful when applied to classes and their member functions, and combined with other language facilities, like inheritance. The following list provides some of the programming idioms that guided the development of the framework.

Policy-based design This design technique makes use of templates to allow "assembling a class with complex behaviour out of many little classes, each of which takes care of only one behavioural or structural aspect" [59, p. 45]. The new framework will strive to make use of this idiom whenever some orthogonal functionality can be decomposed into smaller structures.

"Template template parameters" To facilitate using multiple template parameters, some of which also being template entities, the framework makes use of "template template parameters". These constructs also help with policy-based design [59, p. 76]. Examples can be found throughout the STL; for example, the `std::vector` is a class template with a template parameter for the value type, but also a parameter for an allocator, which itself is a class template. "Template template parameters" allow propagating types, making the API cleaner. In the framework, this idiom will be used to pass value types to containers, which then will be passed to interfaces operating on them. Since this idiom is quite difficult to describe, an application example is given in section 4.1.1 (for the code sample shown in listing 4.2).

RAII The behaviour known as "resource acquisition is initialisation" is used throughout the framework to ensure that access to a particular resource is held during an object's lifetime. Notably, the new framework will use reference-counted smart pointers for managing dynamically allocated queues. Thanks to that, there will be no risk of ending up with a "dangling pointer", since the shared queue will not be destroyed as long as there is any entity holding a reference to it.

SFINAE "Substitution failure is not an error" is a rule that applies to function templates. With so-called "type traits" and compile-time polymorphism it is possible to provide conditional overloading of function templates via `std::enable_if` (e.g., a single print function that has different overloads for different types), check for the existence of specific member functions, or to provide static checks of matching types. In the framework, it will be used to provide generic functionality while avoiding unnecessary abstraction through class hierarchy.

Meta-programming & variadic templates Templates can also be used to "generate" code at compile time, in a much type-safer manner than using preprocessor definitions. That also includes function and class templates that work with variable number of different types that need not share a common base class. This idiom can be particularly powerful when combined with inheritance in class templates, allowing the functionality of multiple smaller classes to be joined together.

Chapter 4

Implementation

Once the overall structure of the framework was conceived and the key software patterns were identified, the bulk of remaining work was the labour of implementing the required functionality.

The software has been arranged in descriptive namespaces:

covert::framework Defines the core application framework functionality, including the process network nodes (consumers, producers, and processors), input and output interfaces to communication queues/channels, concurrent single-producer, single-consumer queues, thread-safe state holder, node and pipeline connectors, and executors.

covert::utilities Includes a range of general-purpose and helper functions and classes, including time keeping facilities, thread attributes, synchronisation primitives, template meta-programming facilities, filesystem utilities, bit manipulation helpers, command line parsing, logging, type traits, input and output stream overloads, and workers.

covert::modules Includes modules used for *sink* meters, arranged by measured physical quantities (frequency, power, temperature).

covert::primitives Contains functions and classes that access low-level subsystems, such as model specific registers, time stamp counters, and platform-specific helpers.

covert::components Contains reusable complete process network components, such as load generators, loggers, and input schedule parsers.

Before delving deeper into the implementation of the framework, let us look at how an actual application is composed using the framework. The applications are written in quite a declarative style, as the example in listing 4.1 shows. The typical order is as follows:

(1) Declare type aliases for the sake of convenience, with the help of the `using` statement. Aliases are indispensable for class templates, variadic in particular. In the shown example, three meter modules are provided to the meter host component (1.1). Using a subtype contained in the new alias

meter_type, we then provide the right template parameter to the logger (1.2). Simple aliases shorten subsequent declarations and can improve clarity (1.3).
(2) Create configuration structure objects for the used components. Most component define a default settings structure, which encapsulates variables that are supposed to be modified externally.
(3) Create the command line parser object and add configurations of the individual components (3.1). Afterwards parse the command line arguments (3.2).
(4) Initialise global state handlers.
(5) Instantiate the used components using the configured settings structures.
(6) Connect the components into a functioning pipeline.
(7) Instantiate an executor and spawn component processes (7.1). *(Optional)* Explicitly wait for all executor threads to finish (7.2).

4.1 Core framework

4.1.1 Node structure

The most fundamental building blocks of the application framework are the process network nodes. Three classes of nodes are defined: *consumer*, *producer*, and *processor*. As the names suggest, the *consumer* node has the ability to only read data (consume tokens) from an input interface, the *producer* node provides a write-only interface, and the *processor* node combines the two. No bidirectional communication over a single interface is possible.

The nodes are realised using class templates with “template template parameters,”[3] using the technique described earlier in section 3.3. The declaration of such a class is shown in listing 4.2. The `Token` type is passed on as a template parameter to the `Container`, which then both are passed on to the `Reader` template parameter. With such construction, one only needs to write `IConsumer<int, queue, queue_reader>` instead of the more verbose `IConsumer<int, queue<int>, queue_reader<int, queue<int>>>`. Node classes derive from virtual, empty base classes (e.g. `TConsumer`), which are necessary to make type traits more usable with the class templates.

Each node class template contains an interface object, which can either be initialised in the constructor, or configured after instantiation using `set_input` and `set_output` functions. The end user will rarely, if ever, need to use the non-default constructor. Classes deriving publicly from the these base node templates can then access the inherited interface object thanks to the `protected` access specifier.

[3]“Template template parameter” is the terminology used in the C++ ISO standard.

```
int main(int argc, char** argv) {
  using meter_type = covert::components::meter_host<                        (1)
      std::chrono::milliseconds, covert::modules::thermal_msr,            (1.1)
      covert::modules::power_msr, covert::modules::frequency_sysfs>;
  using logger_type =                                                     (1.2)
      covert::components::Logger<typename meter_type::token_type>;
  using covert::utilities::CLI;                                           (1.3)
  using covert::utilities::Logging;

  meter_type::settings meter_settings;                                      (2)
  Logging::settings Logging_settings;

  CLI cli;                                                                  (3)
  cli.add_configurations(meter_type::configure(meter_settings),           (3.1)
                         Logging::configure(Logging_settings));

  if (!cli.parse(argc, argv)) {                                           (3.2)
    return 1;
  }

  covert::framework::init_global_state_handlers();                          (4)

  Logging logging(Logging_settings);                                        (5)
  logger_type logger;
  meter_type meter(meter_settings);

  covert::framework::Connector().pipeline(meter, logger);                   (6)

  covert::framework::ThreadExecutor exec;                                   (7)

  exec.spawn([&] { logger.process(); });                                  (7.1)
  exec.spawn([&] { meter.process(); });
  exec.wait();                                                            (7.2)

  return 0;
}
```

Listing 4.1: *A sample application*

```
template <typename Token, template <typename...> typename Container,
          template <typename, template <typename...> typename> typename Reader>
class IConsumer : public virtual Node, public virtual TConsumer {
 public:
  using consumer_type = IConsumer<Token, Container, Reader>;
  using interface_type = Reader<Token, Container>;

  IConsumer() = default;
  explicit IConsumer(typename interface_type::container_pointer input_queue)
      : in_(input_queue){};
  virtual ~IConsumer() = default;

  void set_input(typename interface_type::container_pointer input);
  typename interface_type::container_pointer get_input() const;

 protected:
  interface_type in_;
};
```

Listing 4.2: *Consumer node interface*

The class template for the *processor* node is slightly more involved, due to the multiple inheritance from both *consumer* and *producer* node classes. One important feature of *processor* classes is their ability to bridge potentially disparate domains; hypothetically speaking, the consumer side could be connected to a network interface, and the producer side to a regular queue.

A UML class diagram which grasps the relationships between the various class templates, regular, abstract and interface classes, is presented in fig. 4.1. Such an organisation of software components allows for easy extendability and reuse. Since their most often changed and crucial aspects are template parameters, it is trivial to declare nodes that deal with various token data types. If a new container or an interface is designed, to use it with the node framework one only needs to pass them as template parameters. For further convenience template aliases are provided for default containers and interfaces, such that the user only needs to supply the token types used by the node.

The classes above are generic and do not yet define any executable elements. To implement some functionality, a user can derive from the `IProcess` interface class that defines a process-oriented execution model. The interface contains a pure virtual member function that needs to be overridden by implementing classes. In addition to the `void process()` function the interface class adds the commonly used shared pointer to the global execution state. This is also the only place where an object, the `GLOBAL_STATE`, is declared with the `extern` specifier for external linkage.

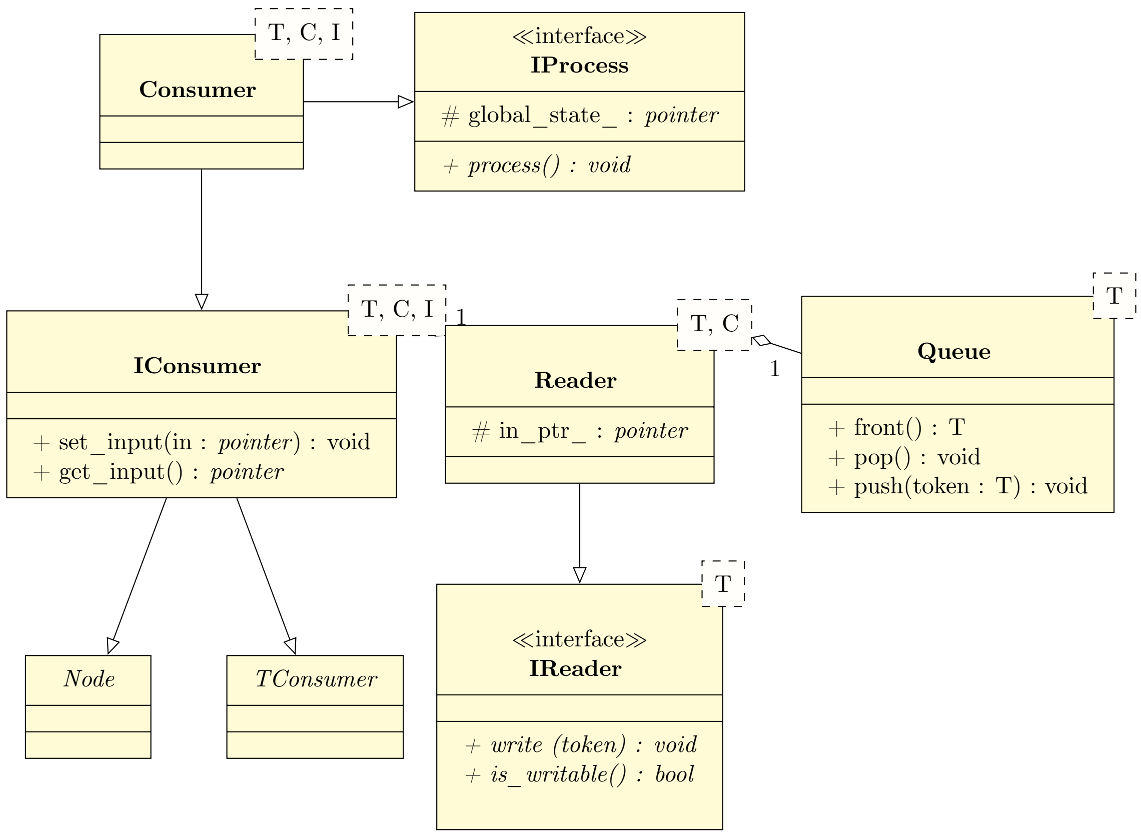

Figure 4.1: *UML class diagram showing relationships in core framework's nodes*

```
template <typename Token, template <typename...> class Container> class
QueueReader : public IReader<Token>, public Reader<Token, Container>;

template <typename Token, template <typename...> class Container> class
QueueWriter : public IWriter<Token>, public Writer<Token, Container>;
```

Listing 4.5: *Queue interfaces*

4.1.2 Interfaces

Communication between process network nodes happens through interface class templates. They provide the set of functionality required for satisfying the requirements of the process network model, that is *blocking reads*, and *non-blocking writes* (as long as queues are declared unbounded). The two pure virtual interface class templates shown below define the functions required for input (listing 4.3) and output interfaces (listing 4.4). The interfaces enforce the process network formalism and allow for different calling styles of the underlying containers (some use the combination of `front`, `pop` and `push` methods, while others define `enqueue` and `dequeue`). Similar layers of indirection are present in other process network implementations [53, p. 73, 61, p. 19].

```
template <typename Token>
struct IReader {
  virtual ~IReader() = default;
  virtual void read(Token &) = 0;
  virtual bool is_readable() = 0;
};
```

Listing 4.3: *Reader interface*

```
template <typename Token>
struct IWriter {
  virtual ~IWriter() = default;
  virtual void write(const Token &) = 0;
  virtual void write(Token &&) = 0;
  virtual bool is_writable() = 0;
};
```

Listing 4.4: *Writer interface*

Additional `Reader` and `Writer` base class templates allow reuse of constructors and provide common functionality of setting and getting the shared pointers to containers used as the 'transmission medium' between nodes. The concrete interfaces, the class templates `QueueReader` and `QueueWriter`, implement the reader and writer virtual interfaces for queue-like objects, and derive from the base classes above (listing 4.5).

The extended queue interfaces additionally implement alternative semantics described in section 3.2 and define member function templates such as `try_read_for(token, timeout)`, as shown in listing 4.6. Analogous functions are provided for the writer interface. The important distinction from the basic interface is the addition of a return type (`bool` instead of `void`), which indicates whether the operation was successful and allows for less conventional, defensive code constructs.

```
inline bool try_read(Token &token);
template <typename Rep, typename Period>
inline bool try_read_for(Token &token,
                         const std::chrono::duration<Rep, Period> &timeout);
template <typename Clock, typename Duration>
inline bool try_read_until(
    Token &token, const std::chrono::time_point<Clock, Duration> &time);
```

Listing 4.6: *Extended queue interface*

4.1.3 Queues

The underlying communication channel has to abide by the process network formalism. The queue provided in the C++ standard libraries does not meet the requirements. First of all, `std::queue` is non-blocking. An empty queue can be “popped”, and calls to the `front()` method can return data from uninitialised memory when the queue is empty. Moreover, `std::queue` is not thread-safe. Even with the queues having only two users, concurrent access has to be free of race hazards.

The framework provides a concurrent queue suitable for single producer-consumer scenarios that mirrors the interface of `std::queue`, and its more complex extension that additionally implements the extended semantics. The queues are provided as class templates, allowing the use of different token data types and synchronisation primitives. Moreover, the queues can have bounded capacity.

The implementation uses two thread synchronisation constructs: mutual exclusion locks, and condition variables. Their combination allows efficient locking and monitoring the status of certain boolean conditions. A lock protect access to private class variables in each of the public interface functions. Two condition variables are used for waiting and notifying on empty and full queues.

All read operations acquire the lock and wait until the queue is not empty, typically using the code construct listed below in listing 4.7. An analogous mechanism is provided for write operations, which will block on a full queue. Waiting using a condition variable is roughly equivalent to `while(!predicate) lock.lock();`.

```
std::unique_lock<mutex_type> lock(queue_mutex_);
queue_empty_cv_.wait(lock, [this] { return !empty_(); });
```

Listing 4.7: *Locking and waiting*

If the read or pop operation was called on an empty queue, the calling thread will efficiently wait until notified by another thread performing a write operation.

```
template <typename Rep, typename Period>
bool try_pop_for(reference value, const duration<Rep, Period> &timeout) {
  std::unique_lock<mutex_type> lock(queue_mutex_);
  if (queue_empty_cv_.wait_for(lock, timeout, [this] { return !empty_(); })) {

    // ...

    lock.unlock();
    notify_waiting_on_full_();
    return true;    // if successful
  } else {
    return false;  // if timed out
  }
}
```

Listing 4.8: *Accessing the queue with a timeout*

Notifications are provided by two internal functions `notify_waiting_on_empty_()` and `notify_waiting_on_full_()`.

The extended interface provides `try_pop` and `try_push` functions, which do not block if the queue is empty or full; if the preconditions are not satisfied, the queue will not be modified and the functions will return `false`.

The usefulness of the condition variables becomes apparent in functions that attempt to read or write only for a specified period of time. The listing 4.8 shows an excerpt from the source code, which defines a `try_pop_for` function. It's a function templates, which can be called with any duration object from the standard library, e.g. `try_pop_for(token, std::chrono::seconds{2})`.

4.1.4 State

State objects in the framework can be used to track local and global state. They contain a set of atomic boolean variables and functions to access and manipulate them, for example `start()` and `is_started()`. The objects are meant to be used through shared pointers, therefore the `State` class derives in a CRTP[4] fashion from `std::enable_shared_from_this<State>` and provides a member function to get a shared pointer to itself.

The only global variable with static storage duration used in the project is the pointer to a state object:

```
static State::state_pointer GLOBAL_STATE{std::make_shared<State>()};
```

Listing 4.9: *Global state object*

[4]Curiously Recurring Template Pattern, in which the class derives from a class template, using itself as a template argument.

Thanks to the static storage specifier the pointer can be used in C-style signal handlers. Currently, the framework defines the following global state signal handlers:

```
static void interrupt_handler(int) { GLOBAL_STATE->stop(); }
static void terminate_handler(int) { GLOBAL_STATE->terminate(); }
static void start_handler(int) { GLOBAL_STATE->start(); }
```

Listing 4.10: *Global state handlers*

4.1.5 Connectors

One of the limitations of some of the reviewed process network implementations was the need to manually create and connect queues to nodes. To remove this burden from the users, a method for connecting compatible nodes together is provided in the framework.

A function template `details::connect` takes two nodes as arguments, verifies that they have compatible interfaces and token types, constructs a queue and passes the shared pointer to neighbouring nodes. The verification uses custom type traits and happens at compile time thanks to `static_assert`.

Using a variadic function template, shown in listing 4.11, any number of nodes can be connected. There is a base function that takes only two arguments, and a recursive function that takes any number of arguments. Thanks to template pack expansion a call to `connect(Node1, Node2, Node3)` will produce a sequence of calls `connect(Node1, Node2); connect(Node2, Node3);`. Additionally, the framework provides a structure wrapper that allows setting the desired queue capacity,[5] and a `pipeline` function. The `pipeline` function uses type traits and a compile-time for-loop to additionally verify that the first passed node is a producer, and the last one is a consumer.

```
template <typename Left, typename Right>
void connect(Left &&left, Right &&right) {
  details::connect(std::forward<Left>(left), std::forward<Right>(right));
}

template <typename Left, typename Right, typename... Rest>
void connect(Left &&left, Right &&right, Rest &&... rest) {
  connect(std::forward<Left>(left), right);
  connect(std::forward<Right>(right), std::forward<Rest>(rest)...);
};
```

Listing 4.11: *Variadic function template for connecting nodes*

[5]With the `Connector` class one can connect nodes with a queue of size 10 by calling `Connector(10).connect(Node1, Node2, Node3);`.

4.1.6 Executors

Executors provide a uniform way to execute processes. The base class provides a function template `spawn`, shown in listing 4.12, which can be used to execute any callable object, optionally with arguments.

```
template <typename Callable, typename... Args>
void spawn(Callable &&callable, Args &&... args);
```

Listing 4.12: *Executor spawning interface*

The concrete executor classes that are provided in the framework at the moment are meant for executing callable objects on system and user-space threads. For example, the `ThreadExecutor` class will spawn each provided object in a separate system thread, and is well suited for executing process network nodes. It additionally provides a way of spawning an object on a specialised thread, with configurable pinning, scheduling policy and priority.

The other available executor facilitates the use of user-space threads, *fibers*, from the project **Boost/fiber**, and provides convenience functions for spawning *fibers* and adding *fiber* pool worker threads. Both executors also provide functions to query how many worker threads have been instantiated, and to wait for the completion of and join spawned system/user-space threads.

4.2 Meter modules

One of the novel aspects of the new framework is the addition of metering modules. In the legacy codebase, separate programs were written for every *sink* data source. If one chose to, for example, perform thermal and frequency measurements simultaneously, a new program would need to be written.

To facilitate the declaration and use of new metering facilities, the module class type has been introduced. The UML class diagram in fig. 4.2 shows the relationship between the meter host and the modules. A meter module has only a few requirements:

- It must have a settings structure, `struct settings`, which can provide defaults and can be empty;
- It must have a static member function `cli::group configure(settings &)`, which configures the settings class provided by reference, which can return an empty command line configuration group;
- It must have a constructor that takes the settings structure by reference;
- It must have a `measure()` function;

- It should provide a `vector<string> names_and_units()` function if the user wants to facilitate the process of adding headers to log files;
- The meter module should follow the RAII idiom, and should be perfectly usable after instantiation.

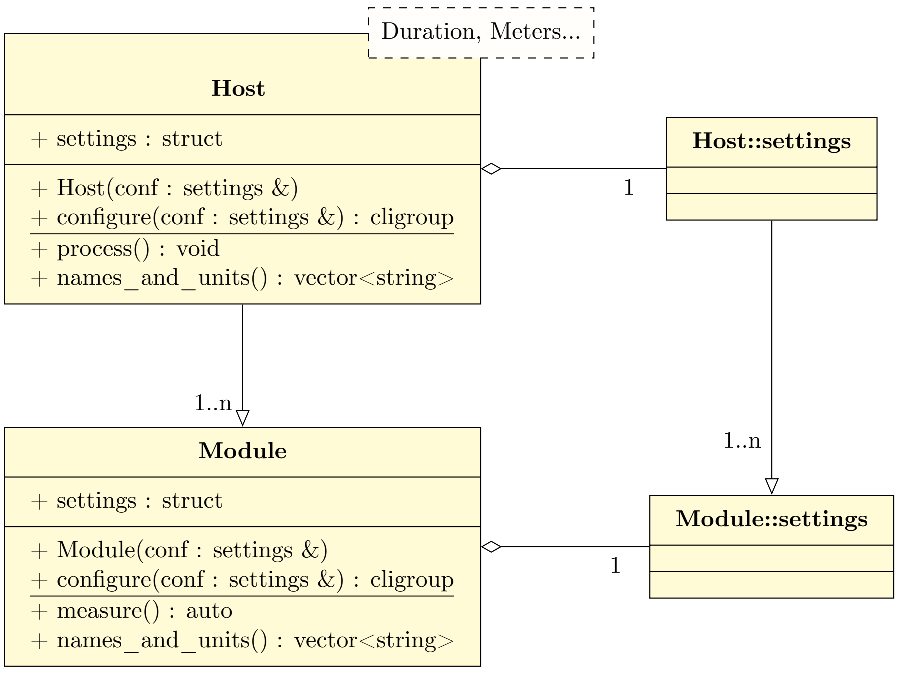
Duration, Meters...
Host
+ settings : struct
+ Host(conf : settings &)
+ configure(conf : settings &) : cligroup
+ process() : void
+ names_and_units() : vector<string>
Host::settings
1
1..n
1..n
Module
+ settings : struct
+ Module(conf : settings &)
+ configure(conf : settings &) : cligroup
+ measure() : auto
+ names_and_units() : vector<string>
Module::settings
1

Figure 4.2: *UML diagram showing relationship between meter host and modules*

There are no hard requirements imposed on the what kind of values a meter module can return via the `measure()` function. It is perfectly valid to have one meter return a `std::tuple<std::string, int, double>` and another return `std::vector<long>`. Although the meter function can return variable length arrays, it might be a better idea to keep the size constant after initialisation to obtain a well formed output.

At the moment the framework provides meter modules for reading temperature via MSRs and `sysfs` thermal zones, processor power via MSRs, and operating frequencies via `sysfs`. The use of meter modules is enabled by the meter host component, described on page 33. The modules can also be used independently of the host, for example in a reactive load generator. Unlike the meters used in the legacy framework, the new meter modules provide default constructors, which perform discovery of available logical cores or accessible pseudo-files.

4.3 Utilities

A wide range of utility functions and classes are provided in the library. These provide both essential functionality, like time keeping or synchronisation mechanisms, and auxiliary helpers, like string formatting and type traits. A partial list is provided below.

Timing Most of the legacy applications needed to perform some functionality with strict time constraints periodically or using an external schedule. The recurring pattern of sleeping and estimating time offsets has been extracted into a `TimeKeeper` class template. The class gives a straightforward interface, with member functions `begin()`, `sleep(chrono::duration)`, and `update_offset()` providing the bulk of the functionality. Additionally, the class has a `run_every` function, which can invoke a callable object with arguments periodically until some predicate is false. For example (listing 4.13), to perform and output some measurement every 10 milliseconds until the global state is terminated, one can run:

```
tk.run_every(std::chrono::milliseconds{10},
            [&]() { return !GLOBAL_STATE.is_terminated(); },
            [&]() { std::cout << meter.measure(); })
```

Listing 4.13: *Executing code blocks periodically*

The time keeper can use different clock sources, different sleep functions, and also provides mean and standard deviation of intervals and offset. The `<covert/utilities/timing.h>` header also provides a `std::chrono`-like interface to the POSIX `nanosleep` function.

Barrier synchronisation primitive The barrier implements a rendezvous point for a configurable thread count. Threads arrive at the barrier and are allowed to pass it only once the specified number threads have reached it. The threads wait on the barrier efficiently, i.e. there is no busy waiting, and threads are descheduled. To allow for greater usability and portability, the older barrier primitive from the POSIX thread library has been replaced with a solution inspired by Boost libraries and C++ standard proposals [62], and uses just the C++ standard library. The advantage of the new barrier is that it can be reused multiple times and can be used on the Android system with API support below level 24.[6]

The new barriers are implemented with a combination of locks and condition variables, both of which are template parameters, allowing for the

[6]The functions `pthread_barrier_init` and `pthread_barrier_wait` are marked `__INTRODUCED_IN(24)` in the `<pthread.h>` header file in the Android's system root. API level 24 is not yet publicly available.

use of different primitives. The barrier is initialised with a desired thread count. As threads arrive at the barrier, they increment an internal current thread count. If the count is less then the desired one, the entering thread waits on a condition variable. If the entering thread is the last to enter the barrier (incremented current count is equal to the desired count), it uses the condition variable to notify all waiting threads.

Command line parsing The command line interfaces in the new framework rely heavily on the very idiomatic header-only library `clipp` by Müller [63]. One of the most helpful features of that library is the automatic generation of help messages, allowing for good extendability of applications and reducing the programmer's burden. For example, to provide an application interface that takes a list of processor cores, a configuration group can be quickly created with `option("-c", "--cores") & integers("core", conf.cores)`, which will write the values into a vector of integers `conf.cores`. Such configuration functions are provided for most meter modules and components in the library.

A separate class `CLI` builds upon the parsing facilities provided in the `clipp` library, and allows easily adding individual components configurations to a master configuration, adding description and example sections to the printed help message,[7] and takes care of parsing the command line arguments and notifying about parsing errors.

Filesystem utilities Since many of the application components access various pseudo-files provided by `procfs`, `devfs` and `sysfs`, the library provides functions for listing directories and for searching directories using regular expressions. Both recursive and non-recursive methods are given. For example, to search for all `thermal_zone` temperature access pseudo-files, the user needs only a single function invocation, `grep_directory_r("/sys/devices/virtual/thermal", ".*zone\\d+/temp$")`, which will return a vector of paths to the files (e.g., "…/thermal_zone0/temp").

Meta-programming helpers and type traits The library provides several compile-time helpers, which are used in other parts of the framework. These include a compile-time `for`-loop, which can also be used as a replacement for code generation preprocessor directives, and functions to apply functions to heterogeneous `std::tuple` containers. For example, to create 256 invocations of a callable object with signature `void(size_t)`, one can use `const_for<0, 256>([](auto I) { fun(I); })`. The template will be instantiated at compile time and expand to a sequence of calls `fun(0); fun(1); ... ; fun(255);`.

[7]Additional sections are automatically wrapped to 80 columns.

Another functionality that does not have any direct application, but is used throughout the library are custom type traits. Type traits, which provide information about types and evaluate predicates at compile time, are heavily used in the C++ standard library. In the case of this work, they are used to check if a type is iterable or const-iterable, if a type is a tuple, or if a node has an input or output interface. These can later be used in `static_assert` statements, or to conditionally enable certain templates.

Formatting and logging support To facilitate the process of formatting readings for logging, the library provides overrides for formatting any iterable or tuple-like objects to `basic_ostream` objects. An example of a signature of a function that overloads the << operator is show below (listing 4.14).

```
template <typename T, typename CharT, typename CharTraitsT>
typename std::enable_if<covert::utilities::is_iterable<T>::value &&
                            !covert::utilities::is_const_iterable<T>::value,
                        std::basic_ostream<CharT, CharTraitsT> &>::type &
operator<<(std::basic_ostream<CharT, CharTraitsT> &ostream, const T &range);
```

Listing 4.14: *Output stream operator overload*

What this achieves is that all types `T`, which return true for `is_iterable<T>::value`, will be allowed to use this overloaded function. In practical terms, the user no longer has to write loops to output vectors. For example, this becomes a valid statement with the overload above: `std::cout << std::vector<int>{1, 2, 3, 4};`. Possibly the greatest utility comes from an overload for tuples, since iterating over tuples is not possible. The overload uses the `const_for` loop above to access tuple elements, allowing for printing of heterogeneous containers. All individual elements are comma separated in the output.

Similar overloads are provided for reading tuples and `chrono::duration` objects from `basic_istream` objects. This functionality is used in the *schedule reader* described later (page 33).

Logging Since logging is used universally across the whole framework and in all applications, it has been extracted into a separate class, which has a uniform configuration interface. the `Logging` class is responsible for creating log files, setting log levels (e.g., critical, debug), and makes sure that the log files are readable. The functionality relies on the `spdlog` library [64], which provides a modern and highly configurable interface, and is also compatible with Android logging facilities. The logging library is also thread-safe and very fast: 4,328,228 messages per second can be logged to a file in a single-threaded mode on an Intel i7-4770 processor [64]. For convenience, most components and modules try using loggers defined in the global logger registry, both for application and debug logging.

Thread parameters The library also features a `ThreadTraits` class which provides a portable way of setting the affinity and scheduling of threads. Most notably, a solution has been found for setting affinity on the Android platform, which was not possible in the legacy codebase. The Android-compatible way of pinning threads relies on the a syscall to `SYS_gettid` and using the POSIX `sched_setaffinity` function from the `<sched.h>` header.

Workers Several worker classes are available in the framework. A worker can wrap some callable object in a loop that checks the global execution state. Most notably, a worker using policy-based design principles is provided, which can take different synchronisation and threading policy/mixin classes, allowing easy extendability. For example, `Worker<BarrierSynchronisation, SpecialisedThreads>` declares a worker that will be pinned to a CPU and will invoke the callable object between two barriers.

In addition to generic utilities, the library also provides more platform-specific utilities, arranged in the separate *covert::primitives* namespace. At the moment, these mostly include functionality specific to Intel-based platforms.

Model specific register access The framework provides a class for reading and writing *model specific registers* (MSR), which provides some improvements over the methods used in legacy code. In line with the overall aim of making most of the classes conform to the RAII (resource acquisition is initialisation) idiom, the class initialises access to MSRs upon instantiation. A number of checks are performed to make sure that the arguments supplied in the constructor are sensible and that all registers can be accessed. Moreover, in addition to `read` and `write` functions, the class provides methods `read_first`, `read_any`, and `read_all`, the latter returning a vector of readings. These functions, which have their write access counterparts, are mostly meant for users who use the default constructor of the class.

Time stamp counter clock Quite an interesting addition to the framework is a clock that explicitly uses the time stamp counter (TSC) on Intel processors as the time source. The class provides the same interface like the clocks found in the header of the C++ standard libraries. Thanks to that all the facilities in the library (operations on and between time point and duration objects), there is no longer any need for timing functions used in the legacy code. For example, in the legacy codebase, to get the execution time in microseconds of some code block one would need to declare many variables and use multiple transformations, as shown in listing 4.15. With any clock implementing the standard interface the required operations are greatly reduced, as shown in listing 4.16.

Using the TSC as the clock source can be quite beneficial, but there is always the need to estimate the frequency of the monotonic counter. In

```
uint64_t tsc_freq_mhz =
    gettscfreq() / 1000000;

timing t_begin, t_now;
initTiming(&t_begin);
getTime(&t_begin);

// ...

getTime(&t_now);
uint64_t elapsed =
    timing2us(timingDiff(t_begin, t_now),
              tsc_freq_mhz);
```

Listing 4.15: *Legacy timing code*

```
auto begin = tsc_clock::now();

// ...

auto elapsed =
    duration_cast<microseconds>(
        begin - tsc_clock::now());
```

Listing 4.16: *New timing code*

the legacy code the estimation had to be performed internally or by an external executable and supplied as a command line argument. To maintain a clean and usable interface, in `tsc_clock` the estimation happens upon the first invocation of the `tsc_clock::now()` function. Every subsequent invocation can use the estimate immediately, because the relevant function-local variables have have static storage duration and are initialised the first time the control flow passes their declarations.

4.4 Components

Load generator The new framework provides a new multi-threaded load generator. Similarly to the legacy `loadgen_mt` (page 5) the generator can be configured to run on specific processor cores, and can use different thread scheduling policies. In terms of functionality, the new loadgen interprets differently the value indicating which cores are to become active and execute the work loop. In the legacy code, cores in the interval from 0 to the specified value became active; for example, if the loadgen was configured to run on cores 1, 3, 5 and 7, providing the value 3 activated cores 1, 3, and 5, and providing 0 activated no cores. In the new framework the value is interpreted as a bitset, allowing activation of any combination of cores. For example, with the same configuration providing the value 5 (101_2) would activate cores 1 and 5, and providing 12 (1100_2) would activate 5 and 7. Validation is also performed to make sure that provided values are sensible.

The load generator inherits from `IProcess` and the basic consumer class. The token type used is a `std::tuple<unsigned int, std::chrono::microseconds>`, the latter subtype indicating how long a particular state is to be maintained. The node uses the new worker template using policies for barrier synchronisation and specialised threads. Access to the bitset specifier is protected in a lock-free fashion using a `std::atomic_flag`.

Schedule reader The new schedule reader is a producer class template that parses values line-by-line either from a file, or from the standard input (in contrast to standard input only in the legacy framework). For ease of use and reusability, the schedule reader forms a token from each line of input, as long as proper overloads were provided. At the moment any tuple type can be formed into a valid token. Thanks to such design, the schedule reader can be adapted to different components on the receiving end, with only minimal involvement of the user:

```
using loadgen = components::loadgen_mt;
using reader = components::schedule_reader<typename loadgen::token_type>;
```

Listing 4.17: *Using type aliases with the schedule reader*

As this example demonstrates, the developer does not even need to explicitly declare the token type, and can just use the internal types of other components.

Meter host The host allows combining individual meter modules into a readily usable process network component. First, the `MeterHost` is class template, where meter modules are provided as variadic template parameters. The `MeterHost` then inherits publicly from each module, thanks to template pack expansion, as shown in listing 4.18. It's token type is also determined automatically from the inherited modules, using an alias template `meter_token_type`.

```
template <typename Duration, typename... Meters>
class meter_host
    : public Meters...,
      public framework::IProcess,
      public framework::Producer<meter_token_type<Duration, Meters...>>
```

Listing 4.18: *Meter host*

Secondly, all module configuration structures are combined and appended with host-specific settings, also using multiple inheritance and template pack expansion. The host has its own settings class, `struct settings : Meters::settings...;`, and own `configure` function, which incorporates module-specific equivalents. The host's settings class is then passed to modules' constructors; each module only accesses its relevant part of the structure.

The host also combines all meter module measurement functions, and the results of their reported variables' names and units. Thanks to that the meter host's process is very compact:

```
auto until = [this]() { return !global_state_->is_stopped(); };
auto action = [this]() { out_.write(measure()); };
timer_.run_every(conf_.period, until, action);
```

Listing 4.19: *Meter host's process*

Logger The logger node is a very simple, but universal addition to the framework. It is a class template that takes a token type as a template parameter. Thanks to the provided output stream overloads, it will format the token as a string and write it to the application log, separating individual values with commas.

Function nodes The new library also provides nodes that take a callable object as an argument to the constructor. These can be conveniently used for simpler operations and testing, and wrap the invocations of the callable in loop that monitors the global state. Three class templates are provided: `FunctionConsumer`, `FunctionProducer`, and `FunctionProcessor`, which take callable objects with signatures `void(token)`, `token(void)`, and `token(token)` respectively. Unlike C-like function pointers, the callable types can be more complex capturing and/or mutable lambdas. For example, to produce a sequence of 10 tokens the user of the library can write:

```
using token_type = std::tuple<int, std::string>;
int counter{0};

FunctionProducer<token_type> node([&]() mutable -> token_type {
  if (++counter > 10) GLOBAL_STATE->stop();
  return token_type{counter, std::string{"Token #"} + counter};
});

node.process();
```

Listing 4.20: *Declaring function nodes*

Adapter nodes The last type of components in the library allows bridging domains using different concurrency primitives: system and user-space threads. These nodes simply forward tokens, in a way that does not cause race issues.

4.5 Build process and testing

Significant changes were made to the way that the the library and individual applications are built. A more robust solution had to be found to make it easier to use the library, especially considering that it has been decoupled from actual applications. The most notable change is the departure from a manual Makefile-oriented workflow, to another build or build-generation tool. Some of the criteria for the new system included readability, support for cross-compilation, the Android platform, and testing, reproducibility of builds, ease of instrumentation with analysis tools. The tools that were considered were the Python-based SCons and Waf, Meson, Bazel, Buck, and CMake. CMake, Bazel and Meson were given particular attention because they allow creating toolchain descriptions, useful when compiling for different targets from different host machines. Although Bazel and Meson seem to have a more consistent and friendlier syntax, CMake was chosen as the build system for the project. The primary motivations were more robust support for C and C++ projects, longer history, and official support in the Android build system. Moreover, CMake does not build the software itself, but rather generates other build systems' files, including Unix Makefiles, which can be used on different operating systems and in Integrated Development Environments. Compared with GNU Make, CMake provides better convenience, superior correctness and easier scalability [46, p. 146, p. 262].

The library is structured into four directories: *include/covert*, *src*, *vendor* and *test*. The *include/covert* directory contains folders with header files, which reflect the namespace organisation described earlier. To use or add a piece of library code, a developer can include the files in a meaningful way, without caring about the relative path, e.g. `<covert/utilities/barrier.h>`, because the include directories are exported with the library target. The *src* folder contains all source files used for building object code. *Vendor* contains the external third-party dependencies of the library. The need for the *vendor* folder unfortunately arises from the difficulty of importing external CMake-enabled code. An alternative that clones and checks out correct tags of external git repositories during the configuration step was also developed, but was deemed less reliable.[8] The dependencies do not have to be committed into the repository; they are set up as git submodules, and a call to `git submodule init` will pull the repositories and point to specific commits. This ensures that only stable features of vendor libraries are used. A `.gitmodules` file is created in the root of the git repository, containing entries like:

```
[submodule "covert/vendor/spdlog"]
    path = covert/vendor/spdlog
    url = https://github.com/gabime/spdlog.git
```

[8]CMake interaction is based on two steps: configure, and build. The external repositories were imported during the build step, therefore inelegant workarounds were required to import their build targets.

And a call to `git submodule status` shows the specific commits and tags of the imported projects:

```
 d2a0196c4dda55d6a55118b248da441ba1f1281b covert/vendor/clipp (v1.1.0-9-gd2a0196)
 cd1d7474374c8cebed5c609564cdaa2f5184be25 covert/vendor/doctest (1.2.9)
+135ab5cf71ed731fc9fa0653051e7d4884a3652f covert/vendor/fmt (4.1.0)
+560df2878ad308b27873b3cc5e810635d69cfad6 covert/vendor/spdlog (v0.17.0)
```

The new framework aims to follow the 'gitflow' workflow, which defines a structure for the repository and software development and release cycle.[9] Develop, release, feature, and hot-fix branches allow better use and organisation of the development efforts. Git flow is also directly supported by the versioning system used at ETH. Thanks to that and tooling support, extending the framework and using it to build applications is easier and more predictable.

The library build targets are governed by the top-level `CMakeLists.txt` file. It defines the project `covert-library` and the main build target, `covert`, which a static library `libcovert.a`. The target is configured with a range of compile features and options, depending on the build configuration type: Release, or Debug. The third-party libraries, most of which include are CMake projects, are included in the top-level configuration. Thanks to that, all dependant targets will inherit the compiler and build settings from the main project. This avoids the situation in which an imported library, possibly from system paths, has been compiled using a different compiler, standard library or ABI. Having an application linked to different standard libraries is strongly discouraged.

Moreover, there are custom targets to auto-format code to achieve uniform style, and to enable static analysis checks using `clang-tidy`. When enabled, the static analyser will be invoked during each build, and a `tidy-all` target will be created, which lints all library code. In the Debug configuration custom targets are also provided with LLVM sanitisers enabled; for example, to compile the target `covert` with the memory sanitiser, one can use an auto-generated target `covert-san-memory`, or `covert-san-thread` for the thread sanitiser. Moreover, a Developer can use the provided CMake function `target_enable_sanitiser(target sanitiser)` to enable those for any target. To improve the library user's experience, some helpful messages are printed during configuration. For example, the linked libraries and compile flags are printed out in the console.

To use the library, a developer can use the new framework by including the library repository in their sources, and importing the library target via the `add_subdirectory` CMake function. The submodule mechanism described above can be used to import a specific version of the library to produce applications.

[9]For a more detailed description and a tutorial, please refer to ETH's Gitlab help pages [65].

The developer can create separate configurations which do not ‘pollute’ the codebase, by running:

```
cmake -DCMAKE_TOOLCHAIN_FILE=path/to/toolchain.cmake \
      -DCMAKE_BUILD_TYPE=Release -B build/Release -H.
cmake -DCMAKE_TOOLCHAIN_FILE=path/to/toolchain.cmake \
      -DCMAKE_BUILD_TYPE=Debug -B build/Debug -H.
```

Listing 4.21: *Configuring the compilation environment*

All necessary build files and all build artefacts are contained in the directories specified with the `-B` flag. Then, to build separate binaries for a target `covert-target`, the user can navigate to those folders and use the `make` program, or from the project’s root run:

```
cmake --build build/Release --target covert-target
cmake --build build/Debug --target covert-target
```

Listing 4.22: *Compiling particular targets*

An important feature of the new build process is the addition of toolchain files, which describe the essentials required for compilation. For example, `x86_64-linux-clang-libcxx.cmake` could define a Clang compiler and use *libc++* instead of *stdlibc++* as the standard library. The `android.toolchain.cmake` from the Android’s Native Development Kit can also be used as a toolchain. Most of the executables produced using the new framework use the LLVM’s Clang compiler. It seems superior to GCC in terms of ease of cross-compilation [66, 67]. [10] To make the builds reproducible, the new library uses the Nix package manager to isolate the build process from the system. Recently the Nix package manager has also been recognised in the scientific community as a tool to facilitate the reproducibility of experimental software [68].

For testing purposes the modern and fast testing framework *doctest* is used in this project [69]. The build file also provides an executable target `covert-test`, which creates a test runner and combines all individual test suites and test cases from the *test* directory. The executable has a rich calling interface, which allows listing available tests and running specific cases. The creation of unit tests is still an ongoing process.

[10]The Android project also relies on LLVM’s Clang by default.

Chapter 5

Evaluation

Evaluating software quality is not an exact science. Even the most popular international standard, ISO/IEC 9126 [70], provides a rather equivocal quality model and only hints at the requirements of metrics used for assessment. Moreover, none of the existing quality models, compared in [71, p. 24], specifies the evaluation methodologies. In the subsequent evaluation, whenever possible, a comparison with the legacy framework is made using a sufficiently accurate metric. However, in many cases the quality characteristics are either vague or refer to the somewhat subjective "quality in use". A more formal assessment is out of scope of this project, and would have to be performed by a more authoritative entity.

In addition to the characteristics listed in the ISO standard, this section will also address the review comments from section 2.2, and some of the attributes mentioned in *API Design for C++* [43], and *Refactoring for Software Design Smells* [39].

5.1 Showcase

A number of concrete applications were created using the new framework. They all share the same structure as the one shown in listing 4.1, and include a single source application, and all possible meter configurations.[11] An experimental data set was produced for the new loadgen, which was pinned to cores 1, 3, 5, and 7. Although it should not be too difficult to adjust the legacy execution framework to the new application interface and log format, due to some external issues a custom runner script had to be written (shown in appendix A.2). As such, the presented examples cannot be threated as formal validation of the external quality requirements. Nevertheless, they do show that the developed applications produce the desired outputs, and the correctness of their execution could have been traced with the help of the log files.

[11]In the future, a meter factory class may be provided to set up meter modules at runtime.

Two examples of logged data are visualised in figs. 5.1 and 5.2. The former presents an excerpt produced with a meter that combines thermal and power information accessed via model specific registers, and the latter shows a combination of all available meter modules in a single log output. The red and blue dashed vertical lines are produced from the load generator's application log, and indicate the onset and end of high utilisation states, respectively. The presented graphs have been post-processed with a 5-sample moving average filter, to improve the clarity of the output.[12] As the graphs show, the change in measured states coincides with the changes in the input state trace. Moreover, fig. 5.2 shows how the meter host (section 4.4) could be used for a more exploratory analysis, where individual modules can be cross-referenced.

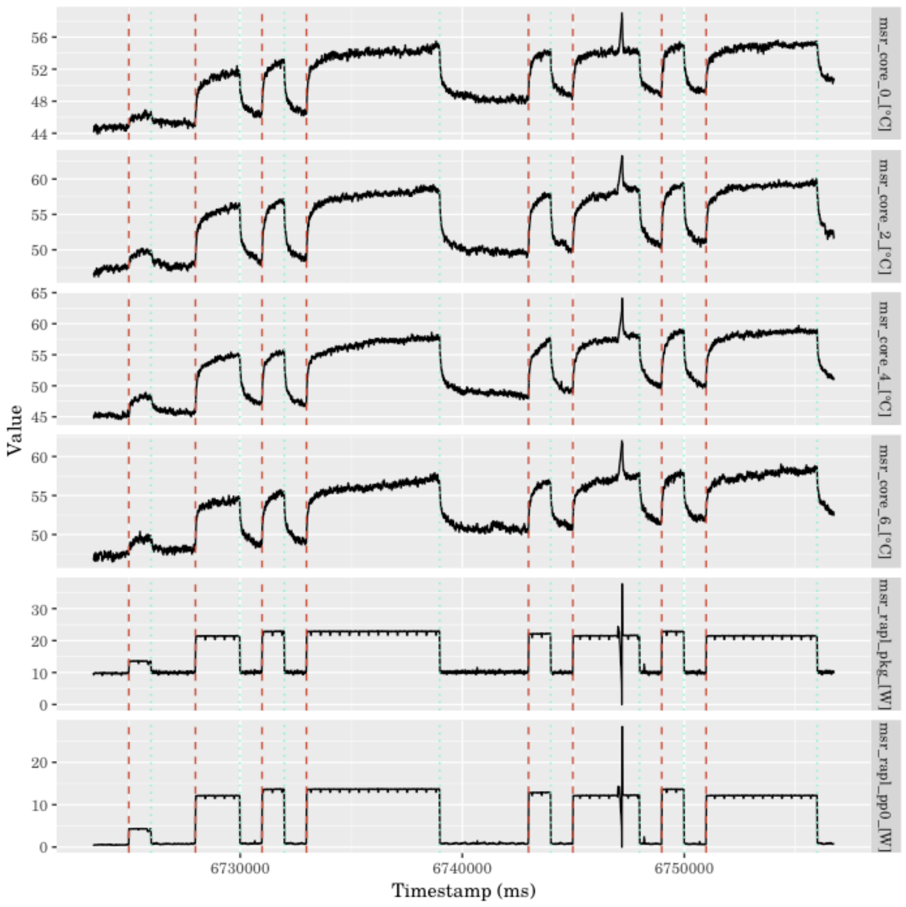

Figure 5.1: *Showcase of a typical experimental run*

[12]In particular, the averaging was performed to alleviate the issue with MSR access as described by Thoroddsen [28].

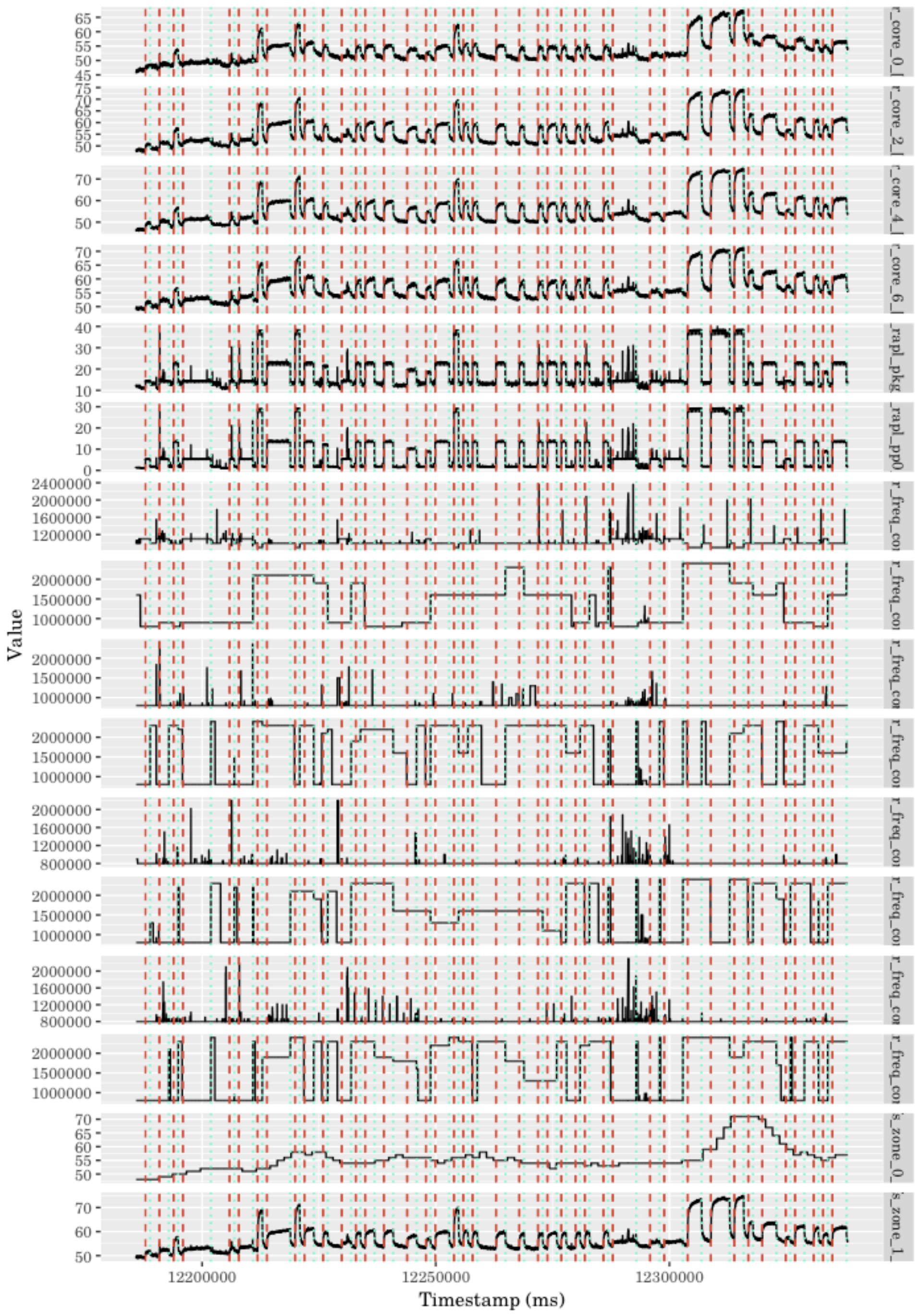

Figure 5.2: *Showcase of a meter using all meter modules*

From top to bottom, the displayed variables are: core temperatures for cores 0, 2, 4, and 6; power in RAPL domains PKG and PP0; scaling frequencies of cores 0 to 7; thermal information from `sysfs` *thermal zones 0 and 1.*

5.2 API design & (re-)usability

A considerable attention has been directed towards creating a usable application programming interface.

The new framework has quite extensive code documentation, which benefits the ISO standard's characteristics of "understandability" and "learnability". There are now 2709 comments per 4169 lines of code – a significant improvement over related work (chapter 1) and the legacy codebase (section 2.2). While the majority of them are Doxygen-formatted function and class headers, there are also many comments describing step-by-step functionality and explaining the use of certain code constructs. The repository could still benefit from additional examples and wiki-page entries, which are likely going to be added in the future.

In terms of "operability", defined as "the capability of the software product to enable the user to operate and control it" [70], the new library largely facilitates the creation of command line interfaces. Examples of invocations of complete *sink* and *source* applications are given in listing A.9.

In terms of API design, the classes in the new library make a reasonably good attempt at providing an abstraction of the problems they address. For example, the `MSR` class "abstracts away" low-level register access, and meter modules enclose all necessary functionality to measure a single aspect of the shared machine state. All classes have good degree of encapsulation, only exposing the neccessary methods and variables, either by class visibility, or namespace separation.[13] In many cases the coupling between classes has been made fairly relaxed, commonly using "aggregation" and "composition" type relationships, with another layer of indirection offered by template parameters.

Some more attention might need to be directed towards evaluating the library for unncessary or missing class abstraction and hierarchy. In particular, quite often class hierarchy is introduced only to facilitate compile-time checks, with the help of completely empty classes. In other cases, the interplay between node interfaces and underlying containers results in some duplication of functionality, which may nevertheless be indispensable for enforcing certain formal requirements.

5.3 Reliability

Suryanarayana *et al.* [39] define reliability as "the extent to which the design fragment supports the correct realization of the functionality and helps guard against the introduction of runtime problems." The new framework aims to

[13]Lower-level functionality not intended for direct use and/or instable, is commonly contained in a `details` sub-namespaces.

provide some improvement over the legacy codebase, especially with respect to avoiding misuse and improving run-time safety.

While the data races found in the legacy applications did not affect the correctness of program execution (reads and writes were still separated by synchronisation barriers), the new framework pays more attention to thread safety and consistently protects access to all shared variables with mutexes or atomic flags. Although a definite guarantees cannot be given due to limited code coverage, the new library seems to be free of data races and memory leaks, thanks to the dynamic analysis of applications and testing code with sanitiser runtimes.

The ISO standard additionally specifies the sub-characteristics of fault tolerance, and recoverability. With those in mind, the new framework introduces fairly widespread range checks and validation of preconditions and arguments, and provides meaningful runtime errors in the case when settings or input data are malformed, or resource access was not successful. Many errors are completely precluded thanks to strong typing and static assertions. Those errors that could arise at runtime can be encapsulated in the try-catch blocks to provide fault recovery.

The notable improvement of the new library is the overhauled command line interface, which prevent incorrect input being passed to the application, thanks to well defined ordering of parameters, and some preliminary type checking. Listings A.7 and A.8 provide samples of the command line interfaces of loadgen and meter applications, respectively.

To further protect the framework from potential misuse, in many cases eumeration classes with descriptive names are used instead of boolean flags. This not only informs the user about what parameter the flag controls, but also helps when successive flags are required.

Another improvement to the reliability of the library and the self-containedness of software components is the almost complete removal of global variables. In the legacy codebase, the `extern` keyword has been used 72 times in *sink*, and 32 times in *source* applications. Each specific application required from 2 to 5 variables with static lifetime and external linkage. In the new framework, the `extern` keyword appears only once, and in a piece of code which the user does not interact with.

Yet another step that the new library performs to ensure safety and correctness is static analysis during compilation. Some of the checks performed by clang-tidy include those specified by C++ core quidelines, as well as ones targeting obsolete or under-performing code. Mostly all discovered issues were fixed. The ones that are still reported are related to variable instantiation order in constructors, and can be considered false positives.

```
using T = int; // or tuple<int, std::chrono::microseconds>

producer<T> p(token_count);
consumer<T> c(token_count);
components::function_processor<T, T> f([](T in) -> T { return in; });
auto f1 = f, f2 = f, f3 = f, /* ... */, f10 = f;

framework::Connector().pipeline(p, f1, f2, /* ... */, f10, c);
framework::ThreadExecutor exec;
exec.spawn(p, f1, f2, /* ... */, f10, c);
```

Listing 5.1: *Evaluation code with forwarding nodes*

```
using T = int; // or tuple<int, std::chrono::microseconds>

producer<T> p(token_count);
consumer<T> c(token_count);

framework::Connector().pipeline(p, c);
framework::ThreadExecutor exec;
exec.spawn_with_properties(0, SchedulingPolicy::RoundRobin, 99, p);
exec.spawn_with_properties(2, SchedulingPolicy::RoundRobin, 99, c);
```

Listing 5.2: *Evaluation code without forwarding nodes*

5.4 Efficiency

Maintaining accurate timing is an important aspect of both the legacy and the new frameworks. To ensure adequate time behaviour, certain parts of the software were further analysed.

First of all, the overhead of the *process network* was roughly estimated using a continuous forwarding test. Two networks were assembled, using a consumer-producer pair of nodes. The producer was provided a command line interface to determine a finite number of randomly created tokens, all of which were passed on through the networks. The first network contained 10 intermediate forwarding nodes, while the second had a direct connection between the consumer and the producer. The abbreviated code samples in listings 5.1 and 5.2 present the instantiation of the two networks. As we can see, the second network is spawned on specialised threads.

In both cases, all nodes were spawned simultaneously. The purpose of this evaluation was to determine the average time required by a token of specific type and size to travel through the network. Time measurement began as soon as the first token reached the consumer, and finished when the specified number of tokens was received. Two token types were evaluated: a 4 byte `int`, and a 16 byte `std::tuple<int, std::chrono::microseconds>`. Although the latter is composed of an `int` and `unsigned long` (8 bytes), its size is 16 bytes due to alignment to 8

Token type	No forwarding nodes		10 forwarding nodes	
	No optim.	High optim.	No optim.	High optim.
Integer	0.67µs	0.27µs	1.87µs	0.79µs
Tuple	0.69µs	0.31µs	2.11µs	0.95µs

Table 5.1: *Average time required to push a token through a process network*

byte boundaries (as reported by `alignof()`). The results of this informal study, using 10 million tokens, are shown in table 5.1. They indicate that the chosen model of computation does not overburden the applications.[14] The overhead evaluation program was also instrumented with call graph and cache analysers (Valgrind). No worrisome heat points were identified, and the majority of the overhead seems to result from locking on queue accesses, and due to token construction/allocation/destruction in the case of the evaluation program.

The timing primitives were also separately evaluated in terms of their ability to perform a function periodically. Two clock source–sleep function pairs were chosen, using `std::chrono::steady_clock` and the new `tsc_clock`, which were bundled with sleep functions `std::this_thread::sleep_for()` and the new `nanosleep()`, respectively. The time keeper was then evaluated 10'000 times each using 256 iterations of a floating-point work loop as the action passed to the `run_every` function (see listing 4.19). Table 5.2 shows the means and standard deviations of intervals and calculated offsets for the both timers, with and without round robin scheduling. Both methods provide good timeliness. Depending on the executed callable and overall system load the performance may vary. Nevertheless, the meter used to produce fig. 5.1 reported an average interval within 0.01% of the desired value, with a standard deviation of 26µs.

The repeatability and accuracy of the new clock source using the time stamp counter was also evaluated. As discussed on page 31, the TSC clock requires frequency estimation. In the case of the new clock, the calculation is rather repeatable, with standard deviation typically below 20 kHz, and the mean being close to that reported by the Linux kernel (within 5‰).

5.5 Maintainability & portability

One of the issues of the legacy codebase was lack of separation of concern. In the new framework, a lot of the functionality has been encapsulated into classes with well defined and constrained responsibilities. Thanks to the more modular

[14]The time required for the creation of the token is included in this overhead calculation.

Duration	`steady_clock` and `this_thread::sleep_for`		`tsc_clock` and `nanosleep`	
	w/ scheduling	w/o scheduling	w/ scheduling	w/o scheduling
	Interval			
100µs	$\mu = 99$µs $\sigma = 0$µs	$\mu = 99$µs $\sigma = 0$µs	$\mu = 99$µs $\sigma = 0$µs	$\mu = 99$µs $\sigma = 1$µs
1ms	$\mu = 1000$µs $\sigma = 9$µs	$\mu = 1000$µs $\sigma = 4$µs	$\mu = 1000$µs $\sigma = 6$µs	$\mu = 1000$µs $\sigma = 3$µs
10ms	$\mu = 10000$µs $\sigma = 4$µs	$\mu = 10000$µs $\sigma = 2$µs	$\mu = 10000$µs $\sigma = 5$µs	$\mu = 10000$µs $\sigma = 4$µs
	Offset			
100µs	$\mu = 5$µs $\sigma = 0$µs	$\mu = 55$µs $\sigma = 0$µs	$\mu = 5$µs $\sigma = 0$µs	$\mu = 55$µs $\sigma = 0$µs
1ms	$\mu = 75$µs $\sigma = 7$µs	$\mu = 130$µs $\sigma = 4$µs	$\mu = 75$µs $\sigma = 3$µs	$\mu = 129$µs $\sigma = 4$µs
10ms	$\mu = 78$µs $\sigma = 3$µs	$\mu = 131$µs $\sigma = 2$µs	$\mu = 76$µs $\sigma = 3$µs	$\mu = 129$µs $\sigma = 3$µs

Table 5.2: *Comparison of timing accuracy for `steady_clock` and `tsc_clock`*

structure, individual asoects of the library are more testable, and are not bound to any particular covert channel application. Because a lot of the implementation details are hidden away from the library user, changes and improvements can be made, potentially without affecting library client's code.

Portability is primarily achieved by relying on the Standard Template Library, and is also addressed by changes to the build process. Platform-specific functions and modules are protected with standard pre-processor directives, and the support for most of the generic functionality has been verified across Unix platforms. Since the library uses quite modern C++ facilities, its portability might be limited by the available compiler and STL support for a target platform. All required features are incorporated into the build configuration requirements. Some minor portability improvements over the legacy codebase have also been given, for example allowing the thread scheduling to be set on Android platforms.

Reddy [43] considers code duplication to be "one of the cardinal sins of software engineering". Regardless of whether that is true, the new framework considerably reduces code duplication compared to the legacy framework. Performing the same step as in section 2.2 revealed that there are 276 duplicate lines in 3011 significant lines of code. Importantly, most of the duplicated blocks in the new framework are below 10 lines; the legacy codebase had much larger blocks, indicative of copy-pasted code. Centralising and compartmentalising the software behaviour makes it more testable and requires only a single point of change for fixes and updates.

Chapter 6

Future work

As a consequence of the new library framework a range of new ideas can be transformed into reusable modules. Moreover, incremental improvements and extensions are also possible thanks to the isolation and tighter scope of framework components. However, due to its infancy there are still multiple issues, both minor and fundamental, that can or need to be resolved. The following sections describe the possible future work classified into functional enhancements, framework extensions and improvements, and software engineering refinements.

6.1 Functionality & core framework

Load generation Some effort is still required to port the remaining load generators to the new framework. Since the work is largely repetitive, it has eventually been given a lower priority. The remaining components would need to provide the functionality of the frequency-oriented programs: `loadgen_bc` and `freq-cc`.

The tokens used by the load generator and the associated schedule reader could also be extended with the ability to use vector-valued arguments. For example, if a duty cycling generator was used, processor cores in separate thermal and frequency domains could independently produce specific utilisation levels, extending the available state space for the encoding of information.

An idea that might be worth exploring is the combination of multiple load generators, most likely those that use orthogonal channels. An encoding scheme combining multiple side-channels could achieve higher capacity.

New *sink/source* pairs To compare and contrast different covert channels reported in literature, the well studied ones and those that have publicly available codebases could be ported to our application library framework. The porting itself could be worthwhile, since a lot of the prior work could benefit from improved documentation and more object-oriented APIs. The potential candidates include the Flush+Reload [7], Flush-Flush [6], and ARM-based [10] cache side/covert channels. Moreover, the various sensors and actuators accessible through the native Android library could be incorporated into the framework.

Exploration Since a large part of the research work involves informal exploration of the side-effects of execution of programs, it might be helpful to provide a more direct, perceptible indication of acquired meter measurements. A pipeline component or a script could be provided to visualise the accumulating readings, using graphical or terminal-based output. Moreover, exploratory visualisation could be provided for the Android platform, possibly taking advantage of existing example code which plots sensor data using OpenGL primitives.

Other functional extensions include:

- Having the meter host estimate and report the load it imposes on the system. While spatial separation should limit the extent to which the metering functions "contaminate" the system state, it might be helpful to have a roughly quantify the load created by metering, which is not negligible, especially in the case of high sampling frequencies.
- A meter module that uses the Linux kernel's power capping framework [72], which provides access to Intel's Running Average Power Limit technology. Since read accesses do not require administrator privileges or loading kernel modules, such a meter would offer an alternative to the power meter using model specific registers.

Some of the possible extensions to the *core framework* are:

New paradigms Implementing the *task-based model of execution* would allow for a broader variety of programming styles and allow for greater sharing of resources among asynchronous components, which could be executed by a thread pool and avoid expensive thread context switches. This goal has originally been attempted, but was unsuccessful due to conceptual and programming difficulties, mostly related to the need for an orchestration mechanism which would monitor task statuses and push tasks to the thread pool's task queue. Implementing a thread pool for a generic one-shot callable object is quite straightforward, but less so for code blocks that need to be executed repeatedly. Using user-space threads was supposed to be the solution, but proved exceedingly difficult to incorporate from compilation perspective.

MIMO The process network nodes could be extended with *multiple input/output interfaces*. These are easy to provide for interfaces that share a type (which can itself be heterogeneous, using `std::variant` and `std::any` template classes from the STL), and a working prototype has already been made. However, even a simple MIMO extension increases the complexity of the support framework and makes the process of connecting component more cumbersome.

Executors The executors in the current framework are quite basic, and provide only a thin layer of abstraction over lower-level system or user-space threads. Further possibilities exist, including loop executors and thread pool executors [see 73].

Communication between components At the moment the queues used for communication are concurrent, but there is likely room for improvement in terms of performance. Additional focus could be given to *zero-copy mechanisms* in order to verify that the implemented move operations (or copy elision/return value optimisation) actually prevent copies being created. It also might be the case that there are leaner synchronisation mechanisms than the currently used combination of locks and condition variables. Moreover, using a more hand-crafted storage method (e.g., using a circular buffer) rather than wrapping a queue class from the STL could deliver better performance, or provide bulk/stride access to queue elements.

Implementing a control flow in addition to data flow might also allow for better collaboration between components; for example, in the GNU Radio project the ability to exchange control messages between nodes allows the use of push/pull message passing semantics (instead of the push-only model of classical process networks).

Deadlocks Currently deadlocks may only arise from improper ordering of functional code and state changes inside the process network nodes. In the case of a pipeline arrangement of nodes there is little chance for deadlocks, both global and local (due to insufficient queue capacities). If the framework was to be extended with multiple inputs and outputs, a proper deadlock detection and resolution mechanism would need to be provided. An observer could monitor and resize the queues, but the process nodes might also need to be instrumented with an execution state and thread-safe access to it. Successful mechanisms have been developed by Allen *et al.* [74] and Geilen and Basten [51]. At the moment, the extended semantics of `try_{read,write}_{for,until}` allow for handling of local deadlocks without any impact to token ordering, and may even prove sufficient for more complex use cases.

Configuration The reliance on command line arguments for configuring programs is not always ideal. Unless the command is written down, there might be no way of determining which parameters have been configured. One possible improvement to the framework would be to allow declaring the configuration of components and meter modules via file in JSON, YAML, or TOML format, all of which are often used for configuration files. Parts of the associative arrays of the configuration file could be interpreted by components, for example using self identifiers as keys. With such an approach it would be possible to document and reproduce particular execution parameters. For example, a configuration of a meter application could be provided as `./application --config conf.yaml` and look like that in listing 6.1.

```
App:
  Pinning: 1
  Logging:
    Application: /tmp/app.log.csv
    Debug: /tmp/dbg.log.txt
  Period: 10e-03
  Scheduling: RoundRobin
Meter:
  thermal_msr:
    Cores: [1, 3, 5, 7]
    Package: true
  power_msr:
    Domains: [pp0, pkg]
  frequency_sysfs:
    Cores: [1, 3, 5, 7]
```

Listing 6.1: *A hypothetical configuration file*

Other minor framework improvements include:

- *Extended read/write interface.* The interface to queues/channels that allows the "try" semantics should ideally exist as an interface class, such that a user can extend the framework by deriving from it and implementing the member functions. However, the functions are template member functions, which cannot be made virtual.[15] Using a different mechanism than inheritance might be necessary.
- *Documentation.* The Doxygen-generated documentation should be provided alongside source code. The majority, if not all code comments already use Doxygen keywords.
- *Duplicated MSR access objects.* Each meter that needs access to model specific registers has its own instance of the `MSR` class, resulting in duplication of functionality and increased count of file descriptors opened by the process. It might be a good idea to make the `MSR` a singleton class, and allow users to limit which registers they want to access.
- Providing a way of interfacing Java code with the process network to improve interoperability.
- Allowing the meter modules to share configuration.

[15]"Member template functions cannot be declared virtual. Current compiler technology expects to be able to determine the size of a class's virtual function table when the class is parsed."[75, p. 242]

- Improving the state management semantics, and holding the state in an enumeration instead of atomic boolean variables.
- Adding a *meter host* that executes each meter module using a separate worker with barrier synchronisation. The only challenging part about this extension is passing of data between threads. Using the combination of futures and promises would be a valid approach, the difficulty being that the meter host is a variadic class template. The most likely implementation would probably use a tuple using `std::tie` and the meta-programming utilities to transport data to the main thread.

6.2 Software engineering

- *Dimensions.* The calculations should not only be type safe, but also meaningful. It is possible to provide physical quantities to raw values and perform dimensional analysis to make sure that a certain calculation is sensible. This can be achieved without any overhead to the actual operations.
- It might be a good idea to do away with runtime polymorphism where it is not necessary. At the moment polymorphic access through base class pointers is rarely used. Runtime polymorphism could be exchanged entirely to compile-time, with the help of the already used static checks and type traits.
- Systematise the use of copy/move/assignment constructors and operators.
- Add more setter and getter methods to allow more expressive configuration of modules beyond the point of construction (most constructors already provide sensible defaults).
- Try to achieve successful builds with the Boost dependency on Android.
- Extend the barrier primitive with optional timeouts or access to the state object, which would possibly allow for recovery mechanisms in case of deadlocks or program termination.
- *Build process.* I think it might be beneficial to set up continuous integration on stable and development branches of the program, with cross-compilation for all target platforms. ETH's Gitlab already provides this functionality, and also supports containerised environments.
- The used test framework allows for test-driven development, where tests can be written inside the production/library code. It might be beneficial from a usability point of view, since the burden of creating separate test files, adding the necessary includes and incorporating them in the build system would no longer be present.

- *Packaging software.* A more efficient and reproducible way of packaging the applications would be nice to have. The idea would be to mimic the way that Android bundles all the required shared libraries together with the application in the APK file. There are several, possibly mutually compatible, solutions to this problem.

  First of all, with the support of the build system one could attempt to statically link all necessary libraries. However, this might prove challenging, since the support for static linkage of C and C++ standard libraries, compiler support libraries, and application binary interfaces is somewhat varied.

  An alternative to static linkage, available on most Unix platforms, is the use of the run-time search path, RPATH, hard-coded into the executable, which can be used to instruct the dynamic linker to use bundled shared libraries instead of those provided by the system. CMake has good support for controlling the RPATH for each build target [76].

  Alternatively, each target platform could be provided with a Nix profile, such that each application is executed in a known environment, that contains all the necessary run-time dependencies.

  To support the options above and to achieve both reproducible builds and experiment runs, we could use the `arx` utility [77] to package the necessary sources, build instructions, and execution parameters into a single self-extracting and self-executing script, which could be easily distributed to target platforms.

Chapter 7

Conclusions

The semester project resulted in a complete rewrite of the legacy codebase. Rather than just factoring out common functionality from the legacy codebase, the project has focused much more on creating a completely new approach to building application, somewhat departing from original assignment goals. However, the restructuring, with the conceptual underpinning of the *process networks* model, enabled greater extendablity and reuse of the library. The core framework, which provides the building blocks necessary for creating and using *process network* nodes, has been backed up with a rather substantial array of utilities and ready-to-use components. Due to the fact that the project cycle has been altered, insufficient attention has been directed towards a rigorous testing of the application built on top of the new framework. Nevertheless, a preliminary evaluation of the new library suggests that it offers good performance and matches the functionality of the legacy framework, whilst significantly improving the usability.

The count of the margin bullets across the report provides some indication of what the development of the library has focused on: extendability (5●), functionality (17●), portability (4●), quality (10●), reproducibility (1●), reusability (13●), usability (22●). The foremost quality has usually been provided at a very structural level, hence does not appear so often.

Appendix A

Software excerpts

A.1 Process network code examples

```
CPN::Kernel kernel(CPN::KernelAttr("Kernel Name"));

CPN::NodeAttr attr("Node Name", "Node Type");
attr.SetParam("queueSize", 100);

kernel.CreateNode(attr);
kernel.CreateExternalReader("Output");

CPN::QueueAttr qattr(100 * sizeof(long), 100 * sizeof(long));
qattr.SetWriter("Node Name", "Output");
qattr.SetExternalReader("Output");

kernel.CreateQueue(qattr);

CPN::IQueue<long> in = kernel.GetExternalIQueue("Output");

std::vector<long> results;

while (long value; in.Dequeue(&value, 1)) {
  results.push_back(value);
}

in.Release();
kernel.DestroyExternalEndpoint("Output");
kernel.WaitForNode("Node Name");
```

Listing A.1: *Computational Process Networks*

```
tbb::flow::graph graph;

tbb::flow::function_node<int, int>
    mirror_node(graph, 3, [](const int v) -> int { return v; });

tbb::flow::function_node<int, int>
    increment_node(graph, 1, [](const int v) -> int { return v + 1; });

tbb::flow::function_node<int, void> print_node(graph, 1, [](const int v) {
  std::cout << v << std : endl;
});

tbb::flow::make_edge(mirror_node, increment_node);
tbb::flow::make_edge(increment_node, print_node);

for (int i = 1; i <= 10; ++i) {
  mirror_node.try_put(i);
}

graph.wait_for_all();
```

Listing A.2: *Threading Building Blocks*

```
class Producer : public raft::kernel {
private:
  int i = 0;

public:
  producer() : raft::kernel() { output.addPort<int>("out"); }

  virtual raft::kstatus run() {
    if (i < 10) {
      output["out"].push(i++);
      return (raft::proceed);
    } else {
      return (raft::stop);
    }
  }
}

class Consumer : public raft::kernel {
public:
  producer() : raft::kernel() { input.addPort<int>("in"); }

  virtual raft::kstatus run() {
    std::cout << input["in"].peek<int>() << std::endl;
    input["in"].recycle();
    return (raft::proceed);
  }
}

int main() {
  Producer producer_node;
  Consumer consumer_node;

  raft::map map;

  map += producer_node >> consumer_node;

  map.exe();

  return 0;
}
```

Listing A.3: *RaftLib++*

```
class Node : public ff::ff_node {
private:
  int filter;

public:
  Node() : filter(0);
  void *svc(void *task) {
    unsigned int *t = (unsigned int *)task;

    if (filter == 0) {
      filter = *t;
      return GO_ON;
    } else {
      if (*t % filter == 0) {
        return GO_ON;
      } else {
        return task;
      }
    }
  }
}

int main(int argc, char* argv[]) {
  ff::ff_pipeline pipeline;

  pipeline.add_stage(new Generate());
  pipeline.add_stage(new Node());
  pipeline.add_stage(new Printer());

  pipeline.run_and_wait_end();
  pipeline.ffStats(std::cerr);
  return 0;
}
```

Listing A.4: *FastFlow*

```
class Producer : public Process {
public:
  Producer(const Id &n, Out<int> &o);
  const char *type() const;
  void main();

private:
  OutPort<int> out;
};

class PC : public ProcessNetwork {
public:
  PC(const Id &n);
  const char *type() const;

private:
  Fifo<int> fifo;

  Producer prod;
  Consumer cons0;
  Consumer cons1;
};

PC::PC(const Id &n)
    : ProcessNetwork(n), fifo(id("fifo")), prod(id("prod"), fifo),
      cons0(id("cons0"), fifo), cons1(id("cons1"), fifo){};

const char *PC::type() const { return "PC"; };

int main() {
  RTE rte; // create yapi run-time environment

  ofstream f("./Results/pc.out"); // redirect standard output
  rte.setOutStream(f);
  ofstream g("./Results/pc.err"); // redirect standard error
  rte.setErrorStream(g);

  PC pc(id("pc")); // create toplevel process network

  rte.start(pc); // start the process network and wait for processes to finish

  return 0;
}
```

Listing A.5: *Yapi*

A.2 Runner script

Listing A.6: *Runner script for executing a covert channel experiment*

```python
#!/usr/bin/python3

import argparse
import logging
import os
import re
import shlex
import shutil
import signal
import subprocess
import time

from pathlib import Path

# cli
config = argparse.ArgumentParser()
config.add_argument('--build', dest='build_dir',
                    nargs=1,
                    required=1,
                    help='directory containing build artefacts')
config.add_argument('--experiment', dest='exp_dir',
                    nargs=1,
                    required=1,
                    help='directory containing experiment files')
config.add_argument('--loglevel', nargs=1, default='info', help='logging
↪   verbosity')
config.add_argument('--meter', nargs=1, default='app_msr_meter', help='meter app
↪   name')
config.add_argument('--period', default=1, type=int, help='meter period in
↪   milliseconds')
args = config.parse_args()

if isinstance(args.loglevel, (list,)):
    loglevel = args.loglevel[0]
else:
    loglevel = args.loglevel

# directories
build_dir = Path(args.build_dir[0])
exp_dir = Path(args.exp_dir[0])

runner_log = exp_dir / 'runner_log.txt'
os.remove(str(runner_log.resolve()))

logging.basicConfig(level=getattr(logging, loglevel.upper(), None),
                    format='%(asctime)s >> %(levelname)s >> %(message)s',
                    filename=str(runner_log))

match = re.search('Debug', str(build_dir))
if match:
    loadgen_verbosity = 'trace'
else:
    loadgen_verbosity = 'debug'

for path in [build_dir, exp_dir]:
    if not path.exists():
        logging.critical('directory "{}" does not exist!'.format(path))
        raise FileNotFoundError('directory "{}" does not exist!'.format(path))

# executables
loadgen_path = (build_dir / 'app_loadgen').resolve()
meter_path = (build_dir / args.meter[0]).resolve()

logging.info('using meter executable: {}'.format(meter_path.name))

for path in [loadgen_path, meter_path]:
    if not path.exists():
        logging.critical('executable "{}" does not exist!'.format(path))
        raise FileNotFoundError('executable "{}" does not exist!'.format(path))

# loop through experiment run directories
for index, path in enumerate(sorted(exp_dir.glob('run_*'))):

    logging.info('Run {}: executing with run directory "{}"'.format(index,
     ↪  path.name))
    schedule_file = path / 'all_input.sched'

    if not schedule_file.exists():
        logging.critical('schedule file "{}" does not
         ↪  exist!'.format(schedule_file.name))
        raise FileNotFoundError('schedule file "{}" does not
         ↪  exist!'.format(schedule_file.name))

    debug_dir = path / 'Debug'
    output_dir = path / 'Output'

    # create log directories
    for path in [debug_dir, output_dir]:
        if not path.exists():
            path.mkdir()

    # cli arguments strings
    loadgen_args = '--verbosity {verbosity} -l {app_log} -dl {dbg_log} '
                   '--file {file} --cores {cores} --scheduling roundrobin'
    meter_args = '--verbosity debug -l {app_log} -dl {dbg_log}'
                 '--pin {pin} --period {period} --scheduling roundrobin'

    # loadgen process
    loadgen_app_log = (output_dir.resolve() / 'src.log.csv')
    loadgen_dbg_log = (debug_dir.resolve() / 'src.log.txt')
    loadgen_args = loadgen_args.format(file=schedule_file, cores='1 3 5 7',
        verbosity=loadgen_verbosity,
        app_log=str(loadgen_app_log),
        dbg_log=str(loadgen_dbg_log))
    loadgen_args = shlex.split(loadgen_args)
    loadgen = subprocess.Popen([str(loadgen_path)] + loadgen_args,
     ↪  stderr=subprocess.STDOUT)
    logging.debug('loadgen invocation: {}'.format(loadgen.args))

    # meter process
    meter_app_log = (output_dir.resolve() / 'snk.log.csv')
    meter_dbg_log = (debug_dir.resolve() / 'snk.log.txt')
    meter_args = meter_args.format(pin=0, period=args.period,
        app_log=str(meter_app_log),
        dbg_log=str(meter_dbg_log))
    meter_args = shlex.split(meter_args)
    meter = subprocess.Popen([str(meter_path)] + meter_args,
     ↪  stderr=subprocess.STDOUT)
    logging.debug('meter invocation: {}'.format(meter.args))

    logging.debug('loadgen pid: {}, meter pid: {}'.format(loadgen.pid,
     ↪  meter.pid))

    # start processes
    logging.info('started...')

    time.sleep(1)
    meter.send_signal(signal.SIGUSR1)
    time.sleep(0.1)
    loadgen.send_signal(signal.SIGUSR1)

    # wait for loadgen to finish
    loadgen.wait()
    logging.info('loadgen exited with return code
     ↪  {}'.format(loadgen.returncode))

    # stop meter
    meter.send_signal(signal.SIGINT)

    meter.wait()
    logging.info('meter exited with return code {}'.format(meter.returncode))

    logging.info('finished!')
    for app in [meter, loadgen]:
        app.kill()

    for line in meter_dbg_log.open():
        for match in [re.search('(timing offset.*)', line),
                      re.search('(timing interval.*)', line)]:
            if match:
                logging.info(match.group(0))

shutil.copy(__file__, str(exp_dir / __file__))
```

```
archive_name = '_'.join([exp_dir.name, build_dir.name,
↪  time.strftime("%Y%m%d-%H%M%S")])
shutil.make_archive(archive_name, 'bztar', str(exp_dir))
```

A.3 Command-line interface

```
DESCRIPTION
        This multithreaded can be used to generate synthetic load on the
        specified number of cores.
SYNOPSIS
        ./app_loadgen [--loglevel [off|critical|err|warn|info|debug|trace]]
                      [-l <file>] [-dl <file>]  [-f <path>] [-c <core>...]
                      [--scheduling [other|fifo|roundrobin]]
        ./app_loadgen -h
OPTIONS
        --loglevel, --verbosity [off|critical|err|warn|info|debug|trace]
                    configure debug log level, default: [info]
        -l, --logfile <file>
                    application log file, default: [none]
        -dl, --debug-logfile <file>
                    debug log file, default: [none]
        -f, --file <path>
                    read schedule from specified file, default: [false]
        -c, --cores <core>
                    The cores to which the workers will be pinned
        --scheduling [other|fifo|roundrobin]
                    Set the scheduling policy of self & workers, default: [other]
        -f, --fifo  Set FIFO scheduling policy of self & workers
        -h, --help  print help message
EXAMPLE
        To run the meter on cores 1 and 3, run: ./msr_meter --cores 1 3. All
        valid options are shown in the synopsis above, using standard man page
        syntax.
```

Listing A.7: *The command line interface of the load generator*

```
OPTIONS
        --loglevel, --verbosity [off|critical|err|warn|info|debug|trace]
                    configure debug log level, default: [info]
        -l, --logfile <file>
                    application log file, default: [none]
        -dl, --debug-logfile <file>
                    debug log file, default: [none]
        -t, --period <period>
                    set sampling period in ms, default: [10ms]
        --pin <cpu> pin meter to given cpu, default: [last core]
        --scheduling
                    Set the scheduling policy of self & workers, default: [other]
        -f, --fifo  Set FIFO scheduling policy of self & workers
    Configuration for the MSR thermal meter
        --thermal_msr
                flag for configuring this meter
        -c, --cores <core>
                read temperature of these cores, default: [0..thread/core..n]
        -p, --package
                enable reading package-level temperature, default: [false]
    Configuration for the MSR power meter
        --power_msr
                flag for configuring this meter
        -d, --domains ([pp0] [pp1] [pkg] [sys])
                read power from these RAPL domains, default: [pp0, pkg]
```

Listing A.8: *An excerpt of a meter's command line interface*

```
# Invoking a loadgen
./app_loadgen --loglevel trace -l ./log.csv -dl ./debug.txt       \
              --file sched.txt --cores 1 3 -scheduling roundrobin

# Invoking a complex meter
./app_all_meter --loglevel debug -l ./app.csv \
                -t 2 --pin 7                  \
                --thermal_msr -c 1 3 -p       \
                --power_msr -d pp0 pp1 pkg    \
                --freq_sysfs -c 1 3 5 7
```

Listing A.9: *Examples of valid program invocations*

Appendix B

Declaration of originality

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

You also want to explore other security leaks?
Building an easily extendable application library for security leak research.

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
KLOPOTT	BRUNO

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZÜRICH, 2018-08-02	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.

Bibliography

[1] B. H. Bratton, *The Stack*, English, ser. On Software and Sovereignty. MIT Press, Feb. 2016, ISBN: 026202957X.

[2] P. Kocher, J. Jaffe, and B. Jun, “Differential Power Analysis”, in *Advances in Cryptology — CRYPTO’ 99*, M. Wiener, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 388–397, ISBN: 978-3-540-48405-9. DOI: 10.1007/3-540-48405-1. [Online]. Available: http://link.springer.com/10.1007/3-540-48405-1.

[3] P. C. Kocher, “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”, in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, London, UK: Springer-Verlag, 1996, pp. 104–113, ISBN: 3-540-61512-1. [Online]. Available: http://dl.acm.org/citation.cfm?id=646761.706156.

[4] J. Loughry and D. A. Umphress, “Information Leakage from Optical Emanations”, *ACM Transactions on Information and System Security*, vol. 5, no. 3, pp. 262–289, Jan. 2002. DOI: 10.1145/545186.545189.

[5] Z. Zhou, W. Zhang, Z. Yang, and N. Yu, “Exfiltration of Data from Air-gapped Networks via Unmodulated LED Status Indicators”, pp. 1–12, Nov. 2017. arXiv: 1711.03235. [Online]. Available: https://arxiv.org/abs/1711.03235.

[6] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, “Flush+Flush - A Fast and Stealthy Cache Attack.”, *DIMVA*, 2016. [Online]. Available: http://dblp.org/rec/conf/dimva/GrussMWM16.

[7] Y. Yarom and K. Falkner, “FLUSH+RELOAD - A High Resolution, Low Noise, L3 Cache Side-Channel Attack.”, *USENIX Security Symposium*, 2014. [Online]. Available: http://dblp.org/rec/conf/uss/YaromF14.

[8] C. Maurice, C. Neumann, O. Heen, and A. Francillon, “C5 - Cross-Cores Cache Covert Channel.”, *DIMVA*, vol. 9148, no. Chapter 3, pp. 46–64, 2015. DOI: 10.1007/978-3-319-20550-2_3. [Online]. Available: http://link.springer.com/10.1007/978-3-319-20550-2_3.

[9] G. Irazoqui, T. Eisenbarth, and B. Sunar, “Cross Processor Cache Attacks.”, *AsiaCCS*, pp. 353–364, 2016. DOI: 10.1145/2897845.2897867.

[10] M. Lipp, “Cache Attacks on ARM”, Master’s thesis, Graz University of Technology, Oct. 2016.

[11] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "TruSpy - Cache Side-Channel Information Leakage from the Secure World on ARM Devices.", *IACR Cryptology ePrint Archive*, 2016. [Online]. Available: http://dblp.org/rec/journals/iacr/ZhangSSLH16.

[12] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre Attacks - Leaking Enclave Secrets via Speculative Execution.", *CoRR*, vol. cs.CR, 2018. [Online]. Available: http://arxiv.org/abs/1802.09085v2.

[13] A. Al-Haiqi, M. Ismail, and R. Nordin, "A New Sensors-Based Covert Channel on Android", English, *The Scientific World Journal*, vol. 2014, pp. 1–14, 2014. DOI: 10.1155/2014/969628. [Online]. Available: http://www.hindawi.com/journals/tswj/2014/969628/.

[14] E. Novak, Y. Tang, Z. Hao, Q. Li, and Y. Zhang, "Physical media covert channels on smart mobile devices", in *the 2015 ACM International Joint Conference*, New York, New York, USA: ACM Press, 2015, pp. 367–378, ISBN: 9781450335744. DOI: 10.1145/2750858.2804253.

[15] H. Ritzdorf, "Analyzing Covert Channels on Mobile Devices", English, Master's thesis, ETH Zürich, Department of Computer Science, 2012. DOI: 10.3929/ethz-a-007305126. [Online]. Available: http://hdl.handle.net/20.500.11850/153456.

[16] J. Brouchier, T. Kean, C. Marsh, and D. Naccache, "Temperature Attacks", *IEEE Security & Privacy Magazine*, vol. 7, no. 2, pp. 79–82, Mar. 2009. DOI: 10.1109/MSP.2009.54. [Online]. Available: http://ieeexplore.ieee.org/document/4812164/.

[17] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA - Exploiting DRAM Addressing for Cross-CPU Attacks.", *USENIX Security Symposium*, 2016. [Online]. Available: http://dblp.org/rec/conf/uss/PesslGMSM16.

[18] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon - Cache Attacks on Mobile Devices.", *USENIX Security Symposium*, 2016. [Online]. Available: http://dblp.org/rec/conf/uss/LippGSMM16.

[19] M. Schwarz and M. Weber, "CJAG: Cache-based Jamming Agreement", Tech. Rep., Mar. 2017.

[20] Lipp, Moritz, "IAIK/armageddon", 2017. [Online]. Available: https://github.com/IAIK/armageddon.

[21] T. Hornby, *defuse/flush-reload-attacks*, 2016. [Online]. Available: https://github.com/defuse/flush-reload-attacks.

[22] ——, "Side-Channel Attacks on Everyday Applications", in *Black Hat USA*, Jul. 2016, pp. 1–11. [Online]. Available: https://github.com/defuse/flush-reload-attacks.

[23] C. Hunger, M. Kazdagli, A. S. Rawat, A. G. Dimakis, S. Vishwanath, and M. Tiwari, “Understanding contention-based channels and using them for defense.”, *HPCA*, pp. 639–650, 2015. DOI: 10.1109/HPCA.2015.7056069. [Online]. Available: http://ieeexplore.ieee.org/document/7056069/.

[24] C. Hunger, *casenh/covert-channels*, 2015. [Online]. Available: https://github.com/casenh/covert-channels.

[25] M. Selber, “UnCovert3: Covert Channel Attacks on Commercial Multicore Systems”, Master’s thesis, Zürich, Apr. 2017.

[26] ——, “UnCovert: Operating Frequency, a Security Leak?”, ETH Zürich, Zürich, Tech. Rep., Feb. 2016.

[27] P. Wild, “UnCovert: Evaluating thermal covert channels on Android systems”, ETH Zürich, Zürich, Tech. Rep., Aug. 2016.

[28] O. J. Thoroddsen, “UnCovert4: The Power Covert Channel”, ETH Zürich, Zürich, Tech. Rep., Jun. 2017.

[29] D. B. Bartolini, P. Miedl, and L. Thiele, “On the capacity of thermal covert channels in multicores.”, *EuroSys*, pp. 1–16, 2016. DOI: 10.1145/2901318.2901322.

[30] P. Miedl, “Frequency Scaling as a Security Threat on Multicore Systems”, Mar. 2018.

[31] P. Miedl and L. Thiele, “The Security Risks of Power Measurements in Multicores”, English, in *33rd ACM/SIGAPP Symposium On Applied Computing (SAC 2018)*, Pau, France: ETH Zurich, Apr. 2018, pp. 1585–1592. DOI: 10.3929/ethz-b-000255436. [Online]. Available: https://www.research-collection.ethz.ch/handle/20.500.11850/255436.

[32] IEEE and The Open Group, *usleep*, 2004. [Online]. Available: http://pubs.opengroup.org/onlinepubs/009695399/functions/usleep.html.

[33] R. Redelmeier, “cpuburn”, 2011. [Online]. Available: http://manpages.ubuntu.com/manpages/xenial/en/man1/cpuburn.1.html.

[34] D. Brodowski, *Linux CPUFreq User Guide*, 4.x, Apr. 2017. [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/user-guide.txt.

[35] P. Mochel and M. Murphy, *sysfs*, 4.x, Sep. 2011. [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt.

[36] T. Bowden, B. Bauer, J. Nerin, S. Feng, and S. Seibold, *The /proc Filesystem*, Oct. 1999. [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/proc.txt.

[37] IEEE and The Open Group, *getopt, optarg, opterr, optind, optopt*, 2018.

[38] ——, *time.h*, 2018. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/.

[39] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells*, English, ser. Managing Technical Debt. Morgan Kaufmann, Nov. 2014, ISBN: 0128016469.

[40] "cgag/loc", 2018. [Online]. Available: https://github.com/cgag/loc.

[41] S. Harris, "Simian", 2018. [Online]. Available: https://www.harukizaemon.com/simian/.

[42] C. J. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software", English, *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, Jul. 2008. DOI: 10.1007/s10664-008-9076-6. [Online]. Available: http://link.springer.com/10.1007/s10664-008-9076-6.

[43] M. Reddy, *API Design for C++*, English. Elsevier, Mar. 2011, ISBN: 9780123850041.

[44] LLVM Authors. (2018). ThreadSanitizer, [Online]. Available: https://clang.llvm.org/docs/ThreadSanitizer.html.

[45] P. Miller, "Recursive Make Considered Harmful", *AUUGN Journal of AUUG Inc.*, vol. 19, no. 1, pp. 1–14, 1997. [Online]. Available: http://lcgapp.cern.ch/project/architecture/.

[46] P. Smith, *Software Build Systems*, ser. Principles and Experience. Addison-Wesley, 2011.

[47] D. Marjamaki, "Cppcheck", 2018. [Online]. Available: http://cppcheck.sourceforge.net.

[48] LLVM Authors, "Clang-tidy", 2017. [Online]. Available: http://clang.llvm.org/extra/clang-tidy/.

[49] *GNU Radio Scheduler Details*, Sep. 2013. [Online]. Available: http://www.trondeau.com/blog/2013/9/15/explaining-the-gnu-radio-scheduler.html.

[50] T. M. Parks, "Bounded Scheduling of Process Networks", PhD thesis, Dec. 1995. [Online]. Available: http://ptolemy.eecs.berkeley.edu/papers/parksThesis.

[51] M. Geilen and T. Basten, "Requirements on the Execution of Kahn Process Networks.", *ESOP*, vol. 2618, no. Chapter 22, pp. 319–334, 2003. DOI: 10.1007/3-540-36575-3_22. [Online]. Available: http://link.springer.com/10.1007/3-540-36575-3_22.

[52] G. E. Allen, "Computational process networks", English, PhD thesis, University of Texas at Austin, Austin, May 2011. [Online]. Available: https://repositories.lib.utexas.edu/handle/2152/ETD-UT-2011-05-2987.

[53] Z. Vrba, "Implementation and performance aspects of Kahn process networks", PhD thesis, Jul. 2009. [Online]. Available: https://dblp.org/rec/phd/basesearch/Vrba09.

[54] J. C. Beard, P. Li, and R. D. Chamberlain, “RaftLib”, in *the Sixth International Workshop*, New York, New York, USA: ACM Press, 2015, pp. 96–105, ISBN: 9781450334044. DOI: 10.1145/2712386.2712400.

[55] M. Torquati, *Parallel Programming Using FastFlow*, September 2015, University of Pisa, Sep. 2015.

[56] H. Van Der Linden, “Scheduling distributed Kahn process networks in Yapi”, PhD thesis, Technische Universiteit Eindhoven, 2003.

[57] A. Kukanov, V. Polin, and M. J. Voss, “Flow Graphs, Speculative Locks, and Task Arenas in Intel® Threading Building Blocks”, Intel Corporation, Tech. Rep., Jun. 2014.

[58] A. Podobas, M. Brorsson, and K.-F. Faxén, “A Comparison of some recent Task-based Parallel Programming Models”, English, in *3rd Workshop on Programmability Issues for Multi-Core Computers*, 2010. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:ri:diva-23671.

[59] A. Alexandrescu, *Modern C++ Design*, ser. Generic Programming and Design Patterns Applied. Addison Wesley, Feb. 2001, ISBN: 0-201-70431-5. [Online]. Available: http://www.worldcat.org/title/c-in-depth/oclc/316330731.

[60] C++ Technical Committee, *ISO/IEC 14882:2017*, 2017. [Online]. Available: http://www.eel.is/c++draft/.

[61] M. Goel, “Process Networks in Ptolemy II”, PhD thesis, Berkeley, CA, Dec. 1998.

[62] A. Mackintosh, *C++ Latches and Barriers*, ISO/IEC JTC1 SC22 WG21, May 2018.

[63] A. Müller, “clipp”, 2018. [Online]. Available: https://github.com/muellan/clipp.

[64] G. Melman, “spdlog”, 2018. [Online]. Available: https://github.com/gabime/spdlog.

[65] Gitlab Authors. (Jul. 2018). Gitflow Workflow, [Online]. Available: https://gitlab.ethz.ch/help/workflow/gitlab_flow.md.

[66] J. Roelofs, “Which targets does Clang support?”, in *EuroLLVM 2014*, Mentor Graphics, Apr. 2014, pp. 1–15. [Online]. Available: https://llvm.org/devmtg/2014-04/.

[67] P. Smith, “How to cross compile with LLVM based tools”, in *FOSDEM’18*, Linaro, Feb. 2018, pp. 1–28. [Online]. Available: https://fosdem.org/2018/schedule/event/crosscompile/.

[68] B. Bzeznik, O. Henriot, V. Reis, O. Richard, and L. Tavard, “Nix as HPC package management system”, in *HUST 2017*, Nov. 2017, pp. 1–24.

[69] V. Kirilov, *onqtam/doctest*, 2018. [Online]. Available: https://github.com/onqtam/doctest.

[70] *ISO/IEC 9126-1*, 2000.

[71] D. Gade, "The Evaluation of Software Quality", Master's thesis, University of Nebraska-Lincoln, May 2013. [Online]. Available: http://digitalcommons.unl.edu/imsediss/38/.

[72] *Power Capping Framework*, Apr. 2017. [Online]. Available: https://www.kernel.org/doc/Documentation/power/powercap/powercap.txt.

[73] C. Mysen, *Executors and schedulers*, ISO/IEC JTC1 SC22 WG21, Apr. 2015.

[74] G. E. Allen, P. E. Zucknick, and B. L. Evans, "A Distributed Deadlock Detection and Resolution Algorithm for Process Networks", in *2007 IEEE International Conference on Acoustics, Speech, and Signal Processing*, IEEE, 2007, pp. II–33–II–36, ISBN: 1-4244-0727-3. DOI: 10.1109/ICASSP.2007.366165. [Online]. Available: http://ieeexplore.ieee.org/document/4217338/.

[75] B. Eckel and C. D. Allison, *Thinking in C++*, English, ser. Practical Programming. Prentice Hall, Dec. 2003, vol. 2, ISBN: 0-13-035313-2. [Online]. Available: http://www.cs.ust.hk/~dekai/library/ECKEL_Bruce/.

[76] "Rpath handling · Wiki · CMake / Community", May 2018. [Online]. Available: https://gitlab.kitware.com/cmake/community/wikis/doc/cmake/RPATH-handling.

[77] J. Dusek, "arx", 2018. [Online]. Available: https://github.com/solidsnack/arx.