Runtime Performance Evaluation of a Non-Preemptive Cooperative Multitasking Framework Through Tracing

Bachelor's thesis

for the degree

Bachelor of Science

of the course Informationstechnik

at the Baden-Wuerttemberg Cooperative State University Mannheim

by

Lea Jungmann

Submission on August 28, 2023

Processing Period:	06.06.2023 – 29.08.2023
Student id, course:	7400528, TINF20IT1
Apprenticing company:	DLR e.V.
Company's supervisor:	Jan Sommer
University's reviewer:	Alexander Dück

Declaration

I hereby assure you that I have written my bachelor's thesis on the

Subject

Runtime Performance Evaluation of a Non-Preemptive Cooperative Multitasking Framework Through Tracing

independently and that I have not used any other sources and aids than those indicated.

I also assure you that the electronic version submitted is the same as the printed version.*

* if both versions are required.

Braunschweig, August 28, 2023

Abstract

Software systems for space applications can grow quite complex, making the development of them a time-consuming process. To aid in the development of space application, the Tasking Framework was created to establish a modeling system tailored to the data processing needs of space applications. The Tasking Framework is a cooperative, non-pre-emptive framework and execution platform capable of multi-tasking. The complexity of space applications makes debugging them the traditional way difficult. Tracing is a way to record the behaviour of a system during runtime. These recordings, called traces, can then be analysed to find bugs and extrapolate future system behaviour. This work seeks to implement tracing within Tasking Framework in a way so that it can be used for debugging and further analysis of systems developed with Tasking Framework, while working off of pre-existing structures. Furthermore, the traces are then graphically represented to facilitate debugging. Additionally, this work presents different ways to use traces to predict and draw conclusions about future system behaviour and how they can be applied to the generated traces, such as arrival curves, minimal distance functions and execution time analyses.

Kurzfassung

Softwaresysteme für Raumfahrtanwendung können recht komplex werden, was ihre Entwicklung zu einem zeitkonsumierenden Prozess macht. Um bei der Entwicklung von Software für Raumfahrtanwendungen zu helfen, wurde das Tasking Framework geschaffen, das ein auf die Datenverarbeitungsbedürfnisse angepasstes Modellierungssystem mit sich bringt. Das Tasking Framework ist ein kooperatives, nichtunterbrechendes Framework zur Softwareentwicklung und Ausführungsplattform, die multitaskingfähig ist. Es wird vor allem in Raumfahrtanwendungen eingesetzt. Diese Systeme werden sehr schnell sehr komplex, was traditionelles Debugging schwierig macht. Tracing ist eine Methode, um ein System während seiner Laufzeit aufzuzeichnen. Die Aufzeichnungen, die Traces genannt werden, können im Nachhinein auf Fehler hin analysiert werden und dafür eingesetzt werden, zukünftiges Systemverhalten zu extrapolieren. Diese Arbeit strebt an, Tracing im Tasking Framework so zu implementieren, dass es sowohl für Debugging als auch für tiefergehende Analysen von Systemen, die mit Tasking Framework entwickelt worden sind, eingesetzt werden kann und auf bereits bestehenden Strukturen aufbaut. Dazu werden die erzeugten Traces graphisch dargestellt, um das Debugging zu erleichtern. Zusätzlich stellt sie verschiedene Arten vor, auf die man Tracing benutzen kann, um Aussagen über zukünftiges Systemverhalten zu treffen, wie zum Beispiel Ereigniskurven, Minimalabstandsfunktionen und Laufzeitanalysen.

Contents

List of Figures

1 Introduction

The German Aerospace Center, Deutsches Zentrum für Luft- und Raumfahrt (DLR), is the German, government-backed, research centre for aerospace. [1] It contains over fifty institutes in thirty locations across Germany conducting research in the sectors of aviation, aerospace, energy and mobility as well as security and digitalisation. One of these institutes is the Institute for Software Technology (SC), with its main research topics being Artificial Intelligence, Visualisation, Software for embedded systems, High-Performance and Quantum Computing and software development for distributed and intelligent systems. [19]

Within the department for Software for Space Systems and Interactive Visualisation of SC, the Onboard Software Systems (OSS) group is developing software systems for space applications. Software in space applications can, for example, be used for the control of subsystems, that start and conduct experiments or record data. Another application of software in space is the collection and processing of data of sensor or experiment data before it is used by internal systems or sent back to ground.

2 Motivation

Often times, subsystems for spacecraft use embedded systems for data processing. These systems can grow very complex, with high requirements for example regarding reliability and performance. Furthermore, the demand for software in space applications is rising and software needs to be completed on set dates, which leads to the question on how to develop software for space applications fast but ensure its quality not declining due to rushed development. As an answer to this, the OSS group of the Institute of Software Technology, German Aerospace Center (DLR), developed the Tasking Framework [15] with the purpose to have a framework to build space applications. This saves time that can then be invested in testing and code analysis. Tasking Framework is capable of multi-threading which can make the process of understanding what thread executes when and why quite a difficult task. This step is, however, necessary for debugging. Since Tasking Framework is graph-based, using graphical methods to represent the execution of a Tasking Framework application and search those graphs for errors can facilitate the debugging process [6]. However, the basis of these graphs is information about the runtime. A way to acquire this information is tracing. Regular debugging using breakpoints, that is starting and stopping the execution of a program to peek into its internal state, could violate real-time requirements and would thusly not actually test the system under the required conditions or skew the observed performance. Tracing circumvents this issue by recording the runtime with as little overhead as possible, leaving all analysis for after the recording has been finished. Tracing can also be used during integration tests, to monitor the performance of both the entire system and individual subcomponents. This approach would enable locating possible errors within subsystems or components with just as much information as is needed, as traces from unaffected subsystems can be easily filtered out.

The goal of this work is to heavily expand on the tracer for Tasking Framework that was already implemented in a previous work [21] and be able to display generated traces in a graphical manner using already existing software infrastructures. Additionally, further analysis on these traces should be implemented and tested on several use cases for Tasking Framework.

3 Background

This chapter touches on the theoretical and technical fundamentals that this work will build upon. It introduces the Tasking Framework, and the theory behind it, as well as tracing and the tracing format that is used in this work. This chapter further elaborates on the explanations given in [21].

3.1 Petri Nets

Many systems are modelled using state machines, or a variation of them such as state charts [18] as seen in Figure 3.1, that depict a system as being in a certain state, changing or staying in the state depending on the state itself, depending on the state machine some external input and sometimes additional constraints. Small variations in input may cause additional states, as most state machines do not possess variables but rather express them in additional states. This can lead very quickly to a state space explosion [4], since the amount of states needed can grow exponentially. Because of this, state machines and related state-based modelling systems are often

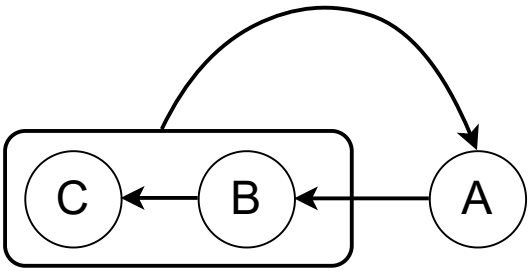

Figure 3.1: A hierarchical state chart with three states.

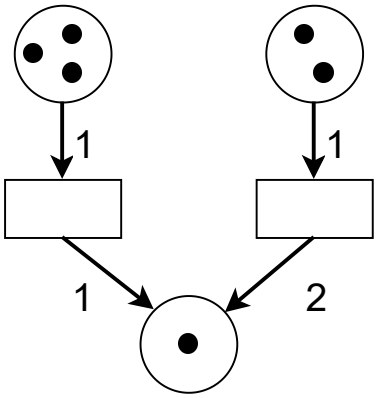

Figure 3.2: A place/transition Petri net with three places and two transitions.

unsuited for data-driven applications, identifying the need of data-driven applications for a different modelling system.

Petri nets can be used to model data-driven systems. Contrary to state machines, Petri nets are data-flow oriented. Instead of modelling the state of a system with individual states to switch into depending on input, like state machines do, Petri nets model how data flows through a system, being processed and transformed along the way.

Petri nets are directed graphs with two different kinds of nodes, places and transitions [27], as seen in Figure 3.2. They are connected by arcs that show the direction of the connection. Two nodes of the same kind are not allowed to follow after another, meaning a place must always follow after a transition and vice versa but a transition can never follow a transition or a place after another a place. Transitions represent the execution of a certain action or set of actions, while places represent data storage [12].

The data of a system is modelled with tokens. They move through the Petri net as the data would. Tokens are stored in places and are consumed and produced by transitions. Transitions consume a specified number of tokens to fire, meaning to execute, and may produce a specified number of outgoing tokens [12]. Figure 3.3 shows a Petri net before and after a transition has fired. Note that the firing transition consumes one token and produces two tokens. The current state of a

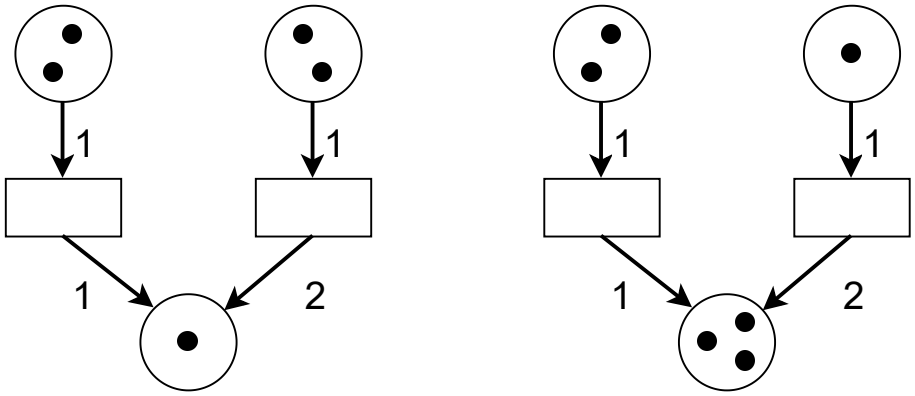

Figure 3.3: A place/transition Petri net before and after the right transition fires.

Petri net is shown by its configuration of tokens, called a marking. How many tokens are consumed and produced by a transition is specified along the arcs that are leading to and from the transition. Concurrency is only allowed in Petri nets if the firing transitions do not affect each other directly, either through different locations within the net or a sufficient amount of tokens [12]. Other kinds of Petri net may forbid concurrency completely or partly. This version of Petri nets is called a place/transition Petri net [28]. However, the model of the Petri net has been expanded upon several times to suit a range of modelling needs. A common variant of the Petri net is the Coloured Petri net (CPN) [11], that expands on the tokens in a Petri net. Instead of one kind of token, CPNs use different kinds of token to represent different data. This can be data in a different format or data type but also data that differs in meaning, such as radiation measurements and altitude. These different tokens are differently *coloured*, most often represented by different coloured tokens but also different token shapes or, in textual representation, by flags that denote their type [11].

3.2 Tasking Framework

The Tasking Framework is a non-preemptive, multitasking, cooperative framework and execution platform, mainly used in the development of space applications [17]. It has been in development by the OSS group of the Institute of Software Technology,

German Aerospace Center, since 2008. It is implemented in C++11 and while it supports different platforms such as Linux, RTEMS and FreeRTOS, it can also be run on bare metal. A main motivation behind the development of Tasking Framework were timing issues that arose during the development of the Attitude and Orbit Control System (AOCS) of the Bispectral Infra-Red Detection (BIRD) satellite due to limited computing power and sensor imposed timing requirements.

Tasking Framework has been used in several projects, including Autonomous Terrain-based Optical Navigation (ATON), Euglena Combined Organic food Production In Space (Eu:CROPIS), Scalable On-Board Computing for Space Avionics (ScOSA) and Matter-Wave Interferometry in Weightlessness (MAIUS).

The Tasking Framework itself is split into two parts: the API and the execution platform. The API consists of the classes that are used to model the application such as tasks, channels, inputs, groups, and events. The execution platform deals with the scheduling of tasks and the different scheduling policies.

3.2.1 API

In Tasking Framework, applications are modelled as a graph of tasks, channels and inputs, as shown in Figure 3.4. Note that the sensors in this figure represent data sources but are not actual part of the way Tasking Framework models application. This task and channel model is modelled after Petri nets, with tasks analogous to transitions and channels likened to places. As with Petri nets, channels and tasks are connected through inputs. Channels can be thought of as data storage while tasks are processing units that take their input from and push their output to channels. Any data that is generated by a task that is not pushed onto a channel is lost immediately after the execution of the task. Once data is pushed on a channel, the inputs that connect tasks to the channel are notified of the new data on the channel. This may lead to the activation of the connected tasks. In addition to channels and tasks, there are also events. Events are used to either periodically trigger a task or to trigger the task after a time-out. Events are a specialised form of channels, meaning they can be directly connected to tasks via inputs, without having to push on a separate

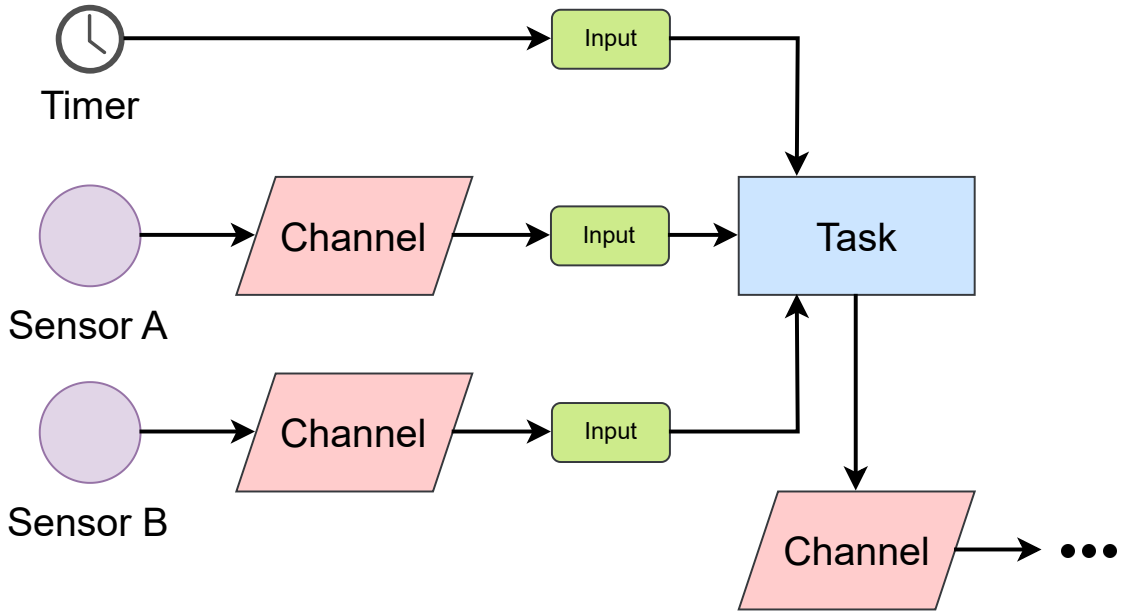

Figure 3.4: A sketch of how the elements of Tasking Framework interact.

channel. Rather, they cause a push on themselves either periodically or after the time-out is up.

The point in time at which a task is activated, that means marked as ready to be executed, depends on the activation model used for this task. Tasking Framework supports different activation models: the OR-semantic, the AND-semantic and custom activation models. The OR-semantic states that at least one input of the task has to have new data for the task to be activated while the AND-semantic requires all inputs to have new data. Custom activation models can be realized in Tasking Framework by setting an input as final, meaning that if this particular input gets new data, the task will be activated, without regard to any other input. If an application needs more complex call semantics, there are two more constructs in Tasking Framework that can be used to regulate the activations of tasks. One is the *Group*. A group is a subset of tasks that, if one task of the subset executes, will wait until all tasks of the subset are executed before the first task can be executed again. The other construct that can control the activation and subsequent execution of tasks is the *Barrier*. Inputs activate after a set amount of pushes to their connected channels. If the exact number of pushes that will activate an input is not known at

compile time, a barrier can be used to count up pushes and notify connected inputs once a certain number of pushes has been reached. Unlike inputs, this number of tolerated pushes can be changed during runtime. The barrier will be instantiated with a minimum number that can be adjusted during runtime if needed.

When a task is activated, it will be queued for execution. Each task and channel possess a unique identifying number. This number is derived from the first four characters of the task's or channel's name, if the object in question got assigned one.

While Tasking Framework can be used solely as a software developing library, it is also possible to use Tasking Framework as an execution platform to execute the application, even without an operating system.

3.2.2 Execution platform

The following describes how Tasking Framework behaves, either on top or in place of an operating system. If Tasking Framework is used on top of an operating system, Tasking Framework has no influence over the internal scheduling of the operating system. The following elaborations only focus on how Tasking Framework handles the execution of tasks internally.

With the activation of a task, it is queued to be executed. Depending on the scheduler, this queuing can look different. Tasking Framework supports three scheduling algorithms, First-In First-Out (FIFO), Last-In First-Out (LIFO) and a priority-based scheduling policy. It is to note that Tasking Framework works non-preemptively, meaning that once a task has started executing, it can not be interrupted. Rather, it will execute until it has finished its execution and then free up an execution slot. Tasking Framework is capable of multi-threading and uses the following approach to solve it:

Every application developed with Tasking Framework needs to possess at least one scheduler object. This scheduler object has several things associated to it, including a collection of tasks, a scheduling policy and a number of so-called executors, which

are Tasking Framework's response to threads. The scheduler will assign tasks to idle executors, ensuring that no executor stays idle while there are tasks in the queue.

The implementation of the execution model is platform-dependent. One of the implemented threading models is that of POSIX, supporting Linux operating systems. Another uses the threading model of C++11 and OUTPOST-core, an execution platform for embedded systems in spacecraft that is developed by DLR, supporting Real Time Operating Systems (RTOS) such as RTEMS and FreeRTOS. The execution model consists of four classes, that work together to ensure the desired scheduling of Tasking Framework. They are the *schedulerExecutionModel()*, implementing the scheduler object and managing all created executor threads, the *clockExecutionModel()*, that is responsible for handling the time for timing events, the *Mutex()*, that implements the mutex, and the *Signaler()*, that implements the conditional variables. If Tasking Framework is to work on bare metal, the execution model, consisting of these four classes, needs to be implemented on the employed hardware.

3.3 Tracing

This section introduces the concept of tracing, how it is distinguished from similar techniques and which tools are used for tracing in this work.

3.3.1 Tracing

Tracing is a form of dynamic software analysis. Other than static software analysis techniques, that include manual review [13] or static analysis tools like checkstyle [20, 30], dynamic software analysis requires the software to be run [5], often even multiple times [35] to study the behaviour of the software. While dynamic software analysis has the advantage of being precise in its observations and their range [5], one must be aware of its pitfalls [5] when employing dynamic software analysis. For one, dynamic software analysis cannot prove the absence of errors, just like software tests, as they cannot cover every possible runtime condition of the program. Therefore, the developer has to choose certain scenarios that are to be observed. Additionally,

programs can behave differently because they are observed, a behaviour called the *probe effect* [25]. Placing hooks or other instrumentation cause an overhead, meaning they add to the execution time of the software. If not used strategically, the hooks' overhead can also affect the scalability of the dynamic software analysis method [5].

Tracing records the behaviour of a system during its execution by placing hooks, called tracepoints, in the code [5]. At its core, tracing produces a trace file that can be read and analysed after it is produced [9]. Trace files are finite and contain a chronological list of all recorded events that occurred during the traced time interval. Even if a continuously running system is traced, only a finite time interval of its execution is traced. Tracing uses tracepoints, small code sections that, when passed during the execution of a program, record the exact time that the tracepoint has been reached at and additional information called payloads. These payloads contain information about the state of the program at the time the program reaches the tracepoint such as values stored in variables or trace messages. Since employing tracing in a program produces a certain amount of overhead, it is of vital to keep this overhead as small as possible as to not interfere with the actual execution itself. It is, however, not possible to avoid this overhead, since no overhead would mean no tracing which would in turn mean no observation of the program.

3.3.2 Tracing versus Logging

Literature is conflicted about the definition of tracing and how it differs from logging. Sometimes the two words are understood to be synonymous [26] or have one of the two be a variety of the other [2, 32], while others view them as completely distinct [3]. This is partly because some tools and techniques are used by both processes and the boundaries between the two are fluid. In this work adopts the distinction made in [32], which details the difference between the two with two metrics: purpose and event volume. While both tracing and logging deal with recording events during the runtime of a program or system, logging serves the purpose of locating bugs with a small event volume, that is the total amount of recorded events, so that redundant information is kept to a minimum. For example, a system administrator could use

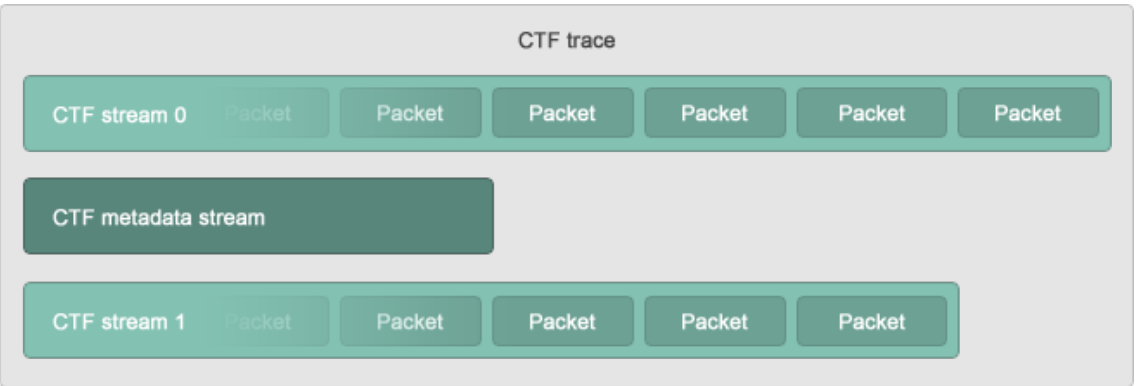

Figure 3.5: The structure of a CTF trace with two streams [7].

logging to overview the processes within a system and narrow down the source of a bug, such as a system call, without having to search through swaths of irrelevant information. Tracing on the other hand deals with much larger event volumes, and has the purpose of accurately depicting the inner workings of every process in a system. Thus, tracing deals with more events on a much smaller scale, with a low level of abstraction.

3.3.3 Common Trace Format

The Common Trace Format (CTF) [7] is a flexible and lightweight binary format used to write traces. CTF traces are split into two parts: the metadata file and binary streams. Figure 3.5 illustrates the structure of a CTF trace and its components. The metadata file is a text file that contains information about how exactly the binary streams are encoded. This includes definitions of data types, headers, packets and events. The metadata file is written in a C-like description language called Trace Stream Description Language (TSDL). TSDL allows for both integer and floating point numerical data types, as well as strings and labelled enumerations and arrays. All these data types can be used within structures to make up more complex data types, or in variants, that dynamically select out of different types. The data in a CTF trace is organized in streams. A CTF trace uses at least two streams in two separate files: a metadata stream and one binary stream. All binary streams could

be written into one single file but can also be separated into one file per binary stream. Each binary stream is made up out of one or more packets. The packets then contain the events, which are recording instances of a tracepoint. An event is made up out of a header and a payload. The payload can contain data in all specified data types and is technically nothing other than a structure itself.

3.3.4 Babeltrace

Babeltrace 2, successor to Babeltrace 1 and Babeltrace in the following, is a command line tool for opening, converting and processing traces, especially in the CTF format [29]. It is capable of reading and printing traces, trimming them, that means only showing the specified time interval, and filtering events. The simplest usage of its Command Line Interface (CLI) is to provide the path to a folder containing a CTF trace as argument. This lists all events within the trace in chronological order in the terminal. It is also possible to additionally provide a time range or start or end times to trim the trace, since most traces contain a sizeable amount of events. Furthermore, Babeltrace is the reference implementation for CTF [7] and is also equipped with bindings for both C and Python. This opens up the opportunity to read and process trace data saved in CTF within C and Python programs.

3.3.5 TraceCompass

TraceCompass is an Eclipse Rich Client Platform (RCP) tool to read, visualise and analyse traces. It is specialised to read CTF traces produced by the Linux Trace Toolkit: next generation (LTTng) tracer [32] that can be used on Linux distributions to trace both user and kernel space processes in correlated traces. LTTng uses one binary stream per CPU [7]. For traces recorded with LTTng, TraceCompass provides a variety of charts, showing the CPU usage, interrupt request analyses, general statistics about the trace, execution graphs for the operating system, flame graphs, disk activity and many more. These charts allow for inspecting, measuring and analysing the opened trace. For example, it is possible to measure the distance (time

interval) between events or to get statistics only for a selected part of the trace instead of its entirety. CTF traces that are not generated by LTTng only get a small selection of generic charts, that is a chronologically ordered list of all events and information about absolute and relative frequencies of all occurring events.

Like all Eclipse RCPs, TraceCompass can be modularly expanded with the help of plug-ins to add more functionality. Plug-ins for TraceCompass include plug-ins for additional analyses, scripting, global filters and support for additional trace types by different tracers and profilers.

4 Implementation of the Tracer

This chapter describes how a tracer is implemented in Tasking Framework so that the flow of applications will be recorded during runtime.

4.1 Previous work

In a previous work [21], a rudimentary tracer was created for Tasking Framework that replicated a single Linux kernel tracepoint to produce a CTF trace. This tracer used the *sched_switch* event to track which task was executing at which point during the execution. The goal of [21] was to implement displayable traces into Tasking Framework. CTF traces already prove to have infrastructure such as TraceCompass and Babeltrace that facilitate visualising and processing them. While there are already tools that allow for recording CTF traces for a system such as LTTng, which traces Linux systems, Tasking Framework is able to run on other platforms next to Linux, such as some RTOSs and bare metal, necessitating a tracing mechanism that can be used either platform independently or with only very few changes depending on the platform. The first tracer was reverse engineered by analysing a Linux kernel trace recorded using LTTng and its metadata file to figure out the precise format of the data within the binary streams. Once the format was apparent, a *Tracer()* class within Tasking Framework was created to record the trace during execution and produce the trace file. The tracer records the *sched_switch* event when the scheduler assigns the task to an executor to start executing the task, meaning right before the execution of a task. The tracer creates one binary file called *stream_0* that in combination with the metadata file makes up a complete CTF trace. This trace can

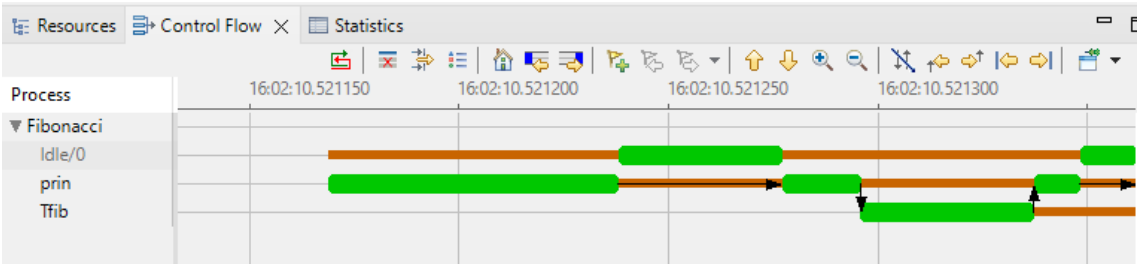

Figure 4.1: A Control Flow graph generated for a trace recorded by the first version of the tracer for Tasking Framework.

then be opened in TraceCompass, where the *Control Flow* chart is generated, as seen in Figure 4.1. It depicts the order in which the tasks are executed but does not contain any information about channels or task activation. This tracer only works on Linux systems because C++11 does not have internal functionality to open and write to a file platform independently. The tracer itself is implemented as a singleton with wrapper functions providing access, meaning that during runtime, even if multiple threads are tracing events, only one tracer object can be created. This prevents conflicting write processes.

4.2 Expansion of the Tracer

This section deals with the implementation and expansion of the tracer in Tasking Framework. While described one after the other, the development of the tracepoints and their graphical representation is intertwined and dependent on each other, with the pitfalls in graphical representation leading to furthering the development of the tracepoints.

Tracing the Tasking Framework is reliant on the instrumentation of its code, i.e. the hooks placed in the source code to record a Tasking Framework application. The Tasking Framework is large enough that it would cause too much overhead to record every single action that is executed during the runtime. Thus, one must identify a configuration of points within Tasking Framework that give an accurate picture of

the inner happenings of the framework. Preferably, with as few points as possible as to avoid causing too much overhead. These tracepoints are:

1. A push on a channel. The push on a channel happens whenever new data is made available to the channel. In turn, all connected inputs are notified, which inform the tasks they are connected to that new data is available on the channel.
2. Activation of a task. The activation of a task signals that a task is ready to be executed and has been queued by the scheduler to wait for the next free execution slot.
3. Task starts & stops executing. This gives insight into how long the task had to wait before executing as well as how long it executed for.

The first attempt to expand on the tracer was to use the same method of repurposing tracepoints from the Linux kernel to add additional information to the Control Flow chart produced in TraceCompass for the traces produced by the first tracer. Unfortunately, none of the selected events from the Linux kernel were picked up on in TraceCompass in any chart save for the *Statistics* chart. The *Statistics* chart is one of the default charts displayed if TraceCompass cannot interpret the trace beyond reading it. Hence, the idea of repurposing events from the Linux Kernel was not further followed and custom events were defined for Tasking Framework, as follows:

```
1  event {
2    name = "activated_task";
3    id = 1029;
4    stream_id = 0;
5    fields := struct {
6      utf8_t _task[16];
7      int32_t _tid;
8    };
9  };
```

Each event in a trace needs to have both a name, an ID and needs to be able to be sorted to a stream, in case multiple streams possess different with the same ID. Custom designing also gives greater control over the amount of overhead produced by the tracer because custom events can use only the exact amount of data and do

not have to fill fields with empty data, as was the case for the *sched_switch* event. Custom events pose the question of how exactly their payloads are supposed to look like. Keeping the payload small is imperative to keeping the overhead small. The four custom events are: push on a channel, activation of a task, start of a task executing and stop of a task executing. Each of these events needs at least the identification number of the particular task or channel to match the events to their corresponding tasks or channels. While this is enough to complete necessary calculations and calculate corresponding graphs, these graphs are not particularly readable for humans. The number associated to each task is not a speaking name and would require the developer or user to look the numbers up in Tasking Framework in a time-inefficient manner. To prevent this, the payload of each custom event includes not only the identification number of a task but also its four character name that is used for display purposes and for user interaction.

4.3 View traces in TraceCompass

While developing and debugging, it can be useful to have the trace of a program not only in textual form, as the Babeltrace CLI provides, but also as a graphical representation. This can help to identify points during the execution where the program does not behave as expected and help to locate bugs [6]. It also aids in the comprehension of large traces, as larger patterns can emerge when zoomed out.

4.3.1 EASE scripting

Eclipse Advanced Scripting Environment (EASE) is a scripting framework to extend Eclipse IDEs and RCPs with custom scripts [31]. It supports JavaScript, Python and Groovy as scripting languages [33]. EASE makes it possible to access internal structures of the IDE or RCP through an API and either add new functionality or build upon existing ones without having to rebuild the application.

In TraceCompass, the EASE plugin can be used to script new graphs to analyse traces. There are two different types of graph supported by TraceCompass that can

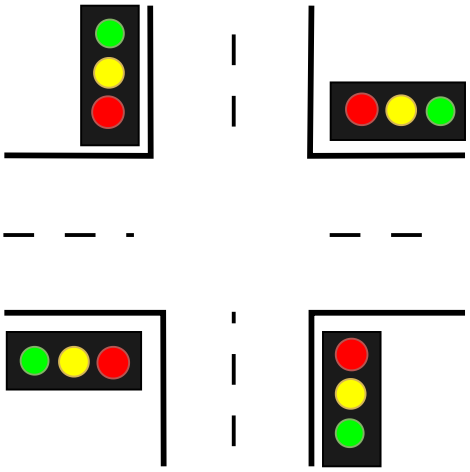

Figure 4.2: Sketch of a four-way intersection with traffic lights

then in turn be used via the interface that EASE provides, namely the *XY-Graph* and the *TimeGraph*. The data is provided to these charts in the form of a *state system*. The state system is a backend representation of trace data and is organized into *elements*, that may be grouped hierarchically. The elements of a state system assume *states* in certain time intervals to display the history contained in the trace, meaning that each individual element has its own history displayed separately next to the others.

To illustrate the state system, consider the following every-day-world example where the different states and their cycle are already known:

Given is an intersection of streets, one going north to south, the other east to west. This intersection is governed by traffic lights, as seen in Figure 4.2. The traffic lights on each street (north/south and east/west) are synched, so that traffic coming from both north and south or east and west has the right of way at the same time. To model this situation, one has to model two interdependent traffic lights: the first for the north to south street and the second for the one going east to west. If this system was traced and its trace put into TraceCompass' state system backend, the individual light bulbs of the traffic lights may be grouped under their respective traffic light and assume either the state *on* or *off*. When given as input into a TimeGraph, the resulting chart could look like what is depicted in Figure 4.3. The north/south traffic

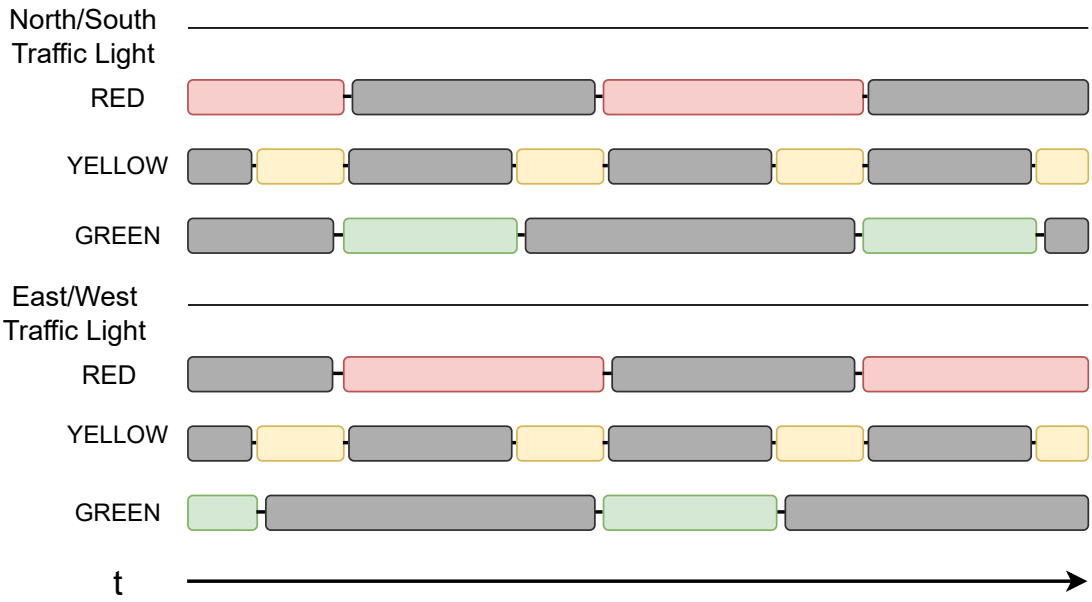

Figure 4.3: A TimeGraph for the state system created by the traffic light example. Coloured sections mean that the particular light is *on*, grey section signal the state *off*.

light only turns on its green light bulb when the east/west traffic light has its red light bulb burning and vice versa. For clarity reasons, this illustrated TimeGraph depicts the *on* states colour coded according to light colour, while *off* states are depicted in grey.

This kind of chart is especially useful when searching for bugs or looking at the behaviour of a specific element.

4.3.2 Custom Chart for Tasking Framework

Since TraceCompass is built and specialised on Linux kernel and user space traces, it does not include many charts for custom traces. In fact, a completely custom CTF trace imported into TraceCompass will get two charts generated by TraceCompass, the Statistics chart, as seen in Figure 4.4, that shows the absolute and relative frequencies of the events in the trace, and a list of all events and their payloads contained in the trace, that is by default chronologically ordered, also shown in

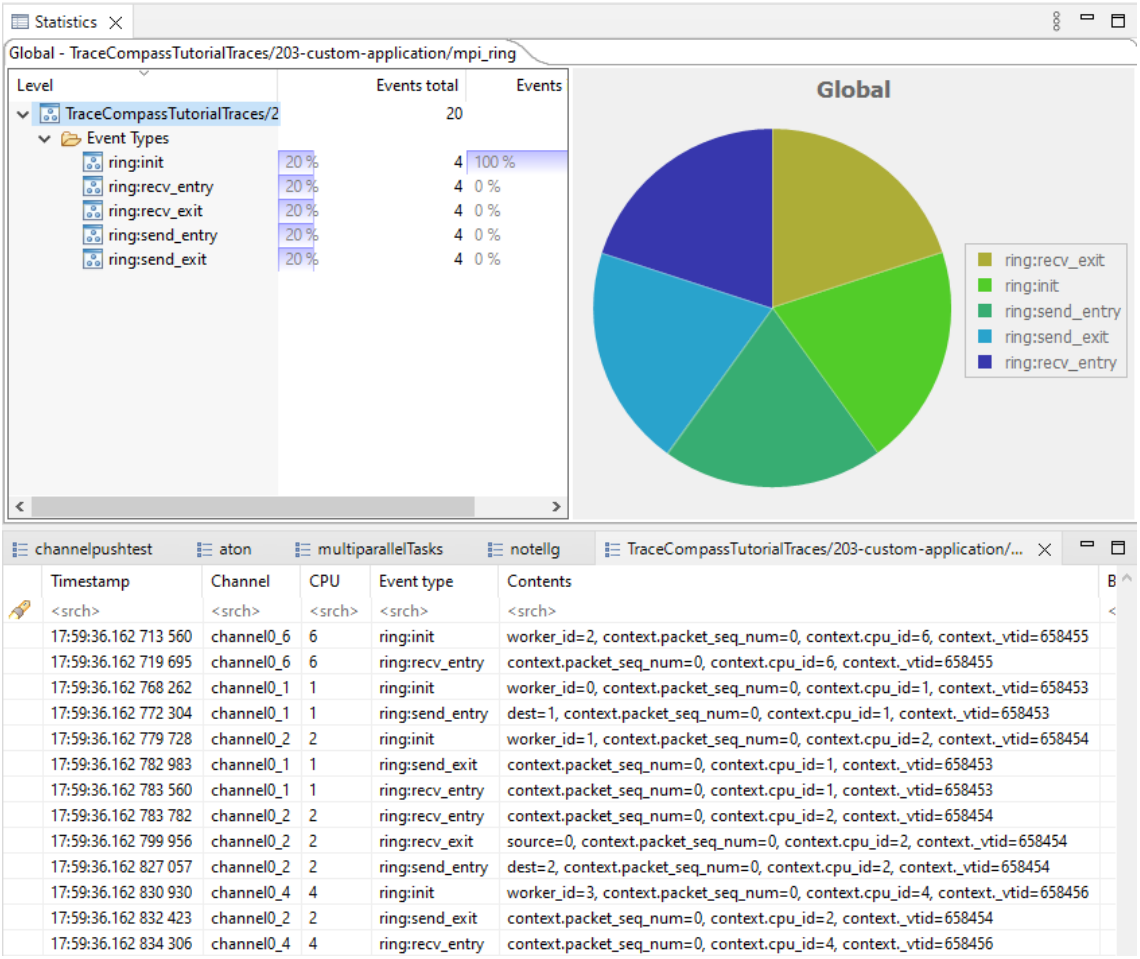

Figure 4.4: The two default charts of TraceCompass: Statistics (above) and a list of events (below), pictured here with example data.

Figure 4.4. While manageable for very small traces, these two charts are not very helpful when used with traces that contain more than a handful of events.

The kernel tracepoints other than the *sched_switch* tracepoint were not recognized by TraceCompass as anything it could analyse and turn into a graph. Thus, the idea of using them was not explored further. This cleared the way for a custom graph representing processes that are specific to Tasking Framework.

Making a custom graph is possible using a Python script and the EASE scripting module integrated into TraceCompass. Python was chosen over JavaScript for its

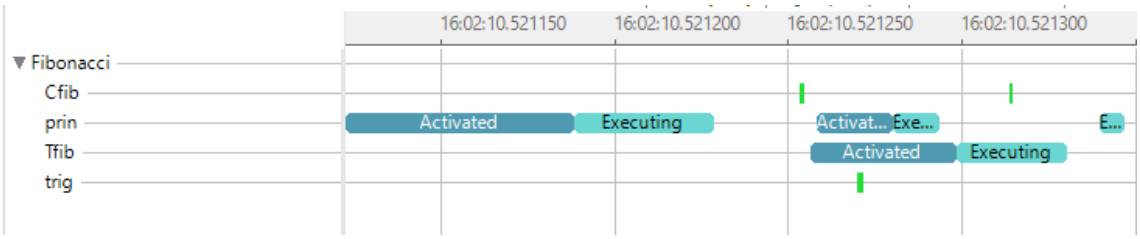

Figure 4.5: A custom TimeGraph created for Tasking Framework applications, depicting the states Activated (dark blue), Executing (light blue) and Push (green).

readability and object-oriented type system. The script takes the currently opened trace as input and iterates over the events. Each event whose name can be matched to one of the custom events is used to extract its *quark*, which in TraceCompass stands for a unique identifier for an object. In this case, the quark is generated from the event name, meaning the events are sorted by task or channel name. The event is then added to the state system using the quark and the timestamp to sort it to the right position. Once all eligible events are added to the state system, it is used to create a *TimeGraph*. TimeGraph lists all states, in this case tasks and channels, on the left side of the diagram while using a timeline as x-axis, as can be seen in Figure 4.5. This custom TimeGraph will be referred to as *Tasking Graph*. This means, each task or channel gets its own individual timeline that displays how the object changes state during the runtime. This means the individual push or activation and execution patterns can be observed per channel or task.

5 Analysis of produced traces

The traces produced by Tasking Framework applications need software infrastructure, like TraceCompass, see Section (3.3.5), to read and analyse them in order for them to be useful to the developer, as they are binary files that are difficult to impossible to read by humans. This chapter elaborates on building and modifying the necessary tools to turn the traces into readable output.

5.1 Analyses of task behaviour

While tracing is useful for debugging and postmortem analysis, it is not the full extent of its uses. When tracing the activations and executions of a task, one can also analyse the trace to study possibly emerging patterns in the task behaviour and use them to predict system behaviour. For systems that run, ideally, in perpetuity or for very long stretches of time, tracing can only offer a snapshot of the system behaviour. However, graphical analysis is not the only analysis that can be applied to a trace. The recorded events of a Tasking Framework trace allow for the extraction of the following information: execution time of every instance of a particular task, activation times and wait times, as well as push behaviour. This array of information has many different uses, two of which will be outlined in the following: arrival curves and distance functions.

5.1.1 Arrival Curves

Contrary to the name, an arrival curve is a function that can be applied to a trace or any other timeline of events. More specifically, the minimal and maximal arrival curves $\eta^-(\Delta t)$ and $\eta^+(\Delta t)$, also called lower and upper arrival curves, are defined as functions on $R^+ \to N^+$, so that for any half-open time interval $[t, t + \Delta t)$ they return respectively either the minimal or maximal number of events that can occur within the interval [16, 22]. Arrival curves are *non-decreasing* [23], with maximal arrival curves being *sub-additive*, meaning that the following is always true for a maximal arrival curve:

$$\forall \Delta_{t1}, \Delta_{t2} \in R^+ : \eta^+(\Delta_{t1} + \Delta_{t2}) \leq \eta^+(\Delta_{t1}) + \eta^+(\Delta_{t2}) \tag{5.1}$$

For an arbitrarily small value $\epsilon > 0$, $\eta^+(\epsilon) \geq 1$ due to events being singular points in time, therefore an interval, however small, can always fit at least one event. If an arrival curve is limited to one event or task τ, it is called arrival curve of τ. This work will refer to the maximal arrival curve $\eta^+(\Delta t)$ as arrival curve, unless explicitly stated otherwise.

5.1.2 Distance Functions

Distance functions are the pseudo-inverse of arrival curves, with the minimum distance function being the pseudo-inverse of the maximal arrival curve and the maximum distance function the pseudo-inverse of minimal arrival curve [16]. Like arrival curves, they can be applied to traces and similar timelines of events. The minimum (*maximum*) distance function $\delta^-(n)$ (respectively $\delta^+(n)$) is defined on $N^+ \to R^+$ and returns the smallest (*largest*) time interval Δt that contains at least (*at most*) n events. Minimum distance functions are non-decreasing and super-additive [16], meaning that every minimum distance function fulfils the following:

$$\forall n, n' \in N^+ : \delta^-(n) + \delta^-(n') \leq \delta^-(n + n') \tag{5.2}$$

Since an event is a singular point in time, the interval containing it can be chosen arbitrarily small as $\epsilon > 0$, leading to $\delta^-(1) = \epsilon$, with $\epsilon > 0$,

Distance functions can be used to derive arrival curves like so [8]:

$$\begin{aligned}
\Delta t = 0 \quad &: \eta^+(\Delta t) = 0 \\
&\quad \eta^-(\Delta t) = 0 \\
\Delta t > 0 \quad &: \eta^+(\Delta t) = \max_{n \geq 1, n \in N} \{n | \delta^-(n) < \Delta t\} \\
&\quad \eta^-(\Delta t) = \min_{n \geq 0, n \in N} \{n | \delta^+(n+2) > \Delta t\}
\end{aligned} \tag{5.3}$$

Like arrival curves, if the distance function is limited to one particular event or task τ, it is called distance function of τ. Unless otherwise specified, this work refers to the minimum distance function as distance function.

5.1.3 Extrapolating trace data

Exploiting the subadditive and super-additive properties of arrival curves and distance functions, it is possible to extrapolate data for the behaviour of a trace under the following assumption: the $\delta^-(2)$, that was observed within the trace, is also the global minimum. With this assumption in mind, distance functions can be extrapolated as follows:

$$\delta^-(n+1) = \delta^-(n) + \delta^-(2). \tag{5.4}$$

while arrival curves can be extrapolated like so:

$$\eta^+(\Delta_t + \Delta_{t2}) = \eta^+(\Delta_t) + \eta^+(\Delta_{t2}) \tag{5.5}$$

When using this extrapolation, it has to be noted that this is a pessimistic extrapolation. Meaning, the extrapolated data has to be treated as lower or upper bound for the distance function and arrival curve respectively, meaning in case of a distance function that the extrapolated values represent the smallest possible value with no

indication of an upper limit. Meanwhile, for the arrival curve the extrapolated data represents an upper bound with no indication for a lower bound.

5.1.4 Application of Maximal Arrival Curve and Minimum Distance Function

The Babeltrace bindings for Python make it possible to iterate over the events of a trace and extract their payloads as well as their timestamps. With this data easily accessible, a command line tool was created that reads in a CTF trace generated by a Tasking Framework application and returns, depending on additional arguments, information about the trace. With no additional argument, the tool returns the number of task activations, the length of the trace in nanoseconds, as well as the shortest, longest and average execution time of a task in the trace. The addition of the name of a specific task returns the same information but specific to this task, minus the overall length of the trace. Next to information about execution time, the tool can also compute arrival curves, both minimal and maximal, and distance functions, both minimum and maximum as well, for either all tasks or a specific task. The arrival curves require a time interval in nanoseconds as input, while distance functions need an integer n to compute the distance. If the given input value exceeds the bounds of the trace, e.g. the argument for a distance function is greater than the number of activations in the trace, the remaining values up to the input value are extrapolated as described in Section 5.1.3. Every time a value is extrapolated, a warning is generated and printed out to the user, so that the user is aware of the data being extrapolated. In addition to the textual output of the tool, it is also possible to plot a graph of the requested data. In the plotted graphs, extrapolated data is marked in red, contrary to the blue of the trace data, as can be seen in Figure 5.1 that depicts a distance function generated for a Tasking Framework application and shows the linear nature of the extrapolation as the distance between each point and its predecessor is the minimum distance between two points. Figure 5.2 shows the arrival curve of the same trace although with less extrapolated data.

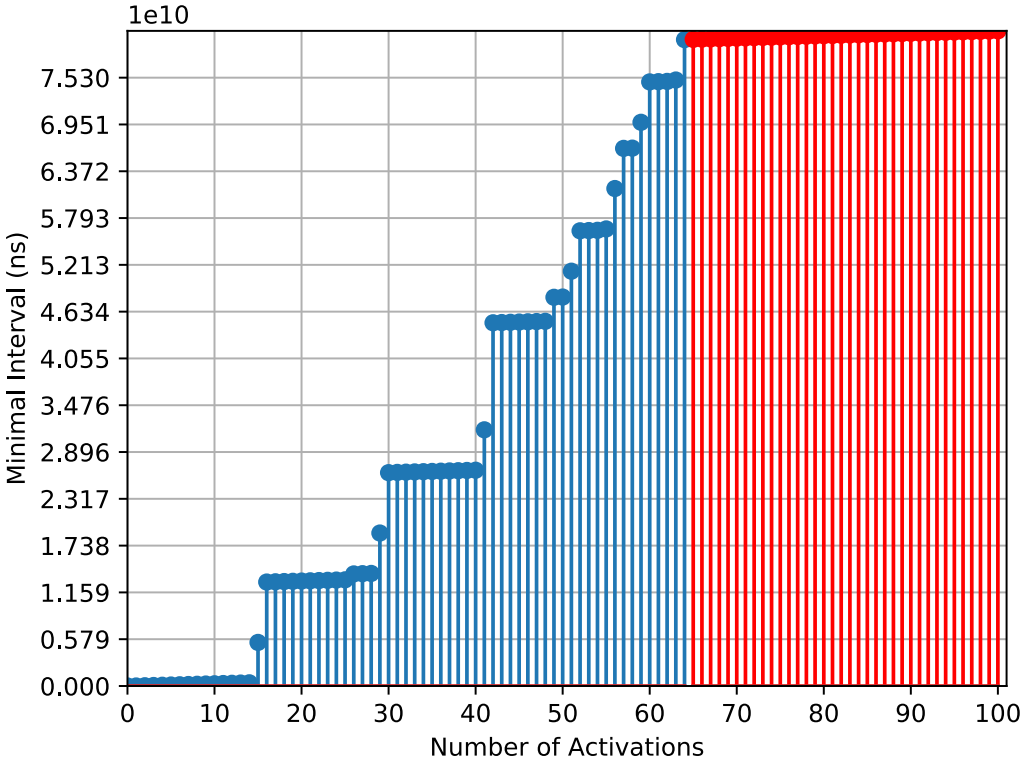

Figure 5.1: A graph of a distance function with extrapolated data shown in red.

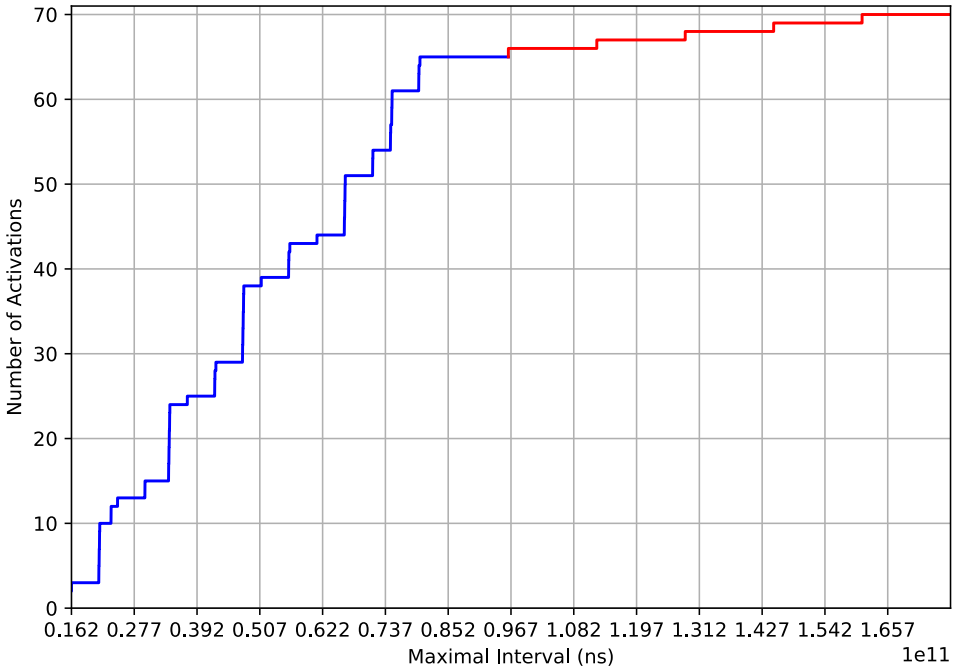

Figure 5.2: A graph of a arrival curve with extrapolated data shown in red.

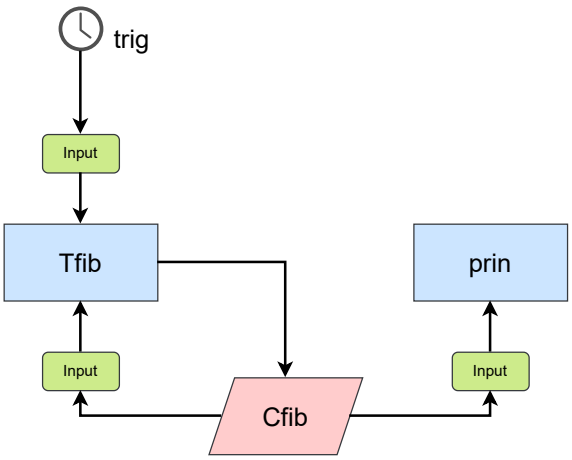

Figure 5.3: Model of the Fibonacci example.

5.2 Use cases

This section shows how the tools implemented for this work can be used on Tasking Framework applications and what information can be gleamed from them. Both of the following examples have been implemented in Tasking Framework and produce traces when executed.

The first use case is a simple example to test functionality that computes and prints Fibonacci numbers until it detects an overflow in the variable that stores the numbers. It consists of two tasks, named *Tfib* and *prin*, a channel that both are connected to called *Cfib* and a periodic event to trigger the computation of the numbers. Figure 5.3 shows the structure of the example. Both tasks are part of a group so that once a new number is computed it is ensured it gets printed before the new number is computed. Once it is executed and the trace produced, the trace is imported into TraceCompass. In TraceCompass, at first, the only chart generated in addition to the two standard ones is the Control Flow graph. To get the Tasking graph, the script responsible for it must be executed first. The resulting Tasking Graph, when zoomed out, can be seen in Figure 5.4. As already seen, the Tasking Graph in TraceCompass uses dark blue to denote the state *Activated*, light blue for *Executing* and green for a push on a channel. If a state is too small for TraceCompass to properly visualise at the current zoom level, its presence is hinted at with a small black dot. Peculiar

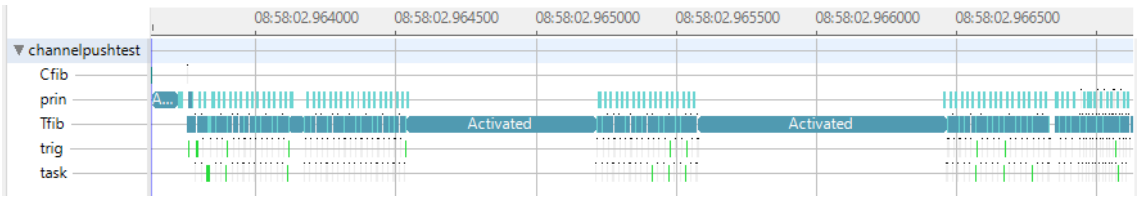

Figure 5.4: The Tasking Graph generated by TraceCompass for the trace of the Fibonacci example.

about this particular Tasking graph are the two long stretches of *Activated* of the task *Tfib*. Since this application was executed on a Linux machine, it is likely that these long periods of activation are due to the operating system interfering with the Tasking Framework application. The same gaps can be observed in both the arrival curve, Figure 5.6, as one wide step in the left half of the graph, and the distance function, Figure 5.5, as two jumps disrupting the otherwise approximately linear growth. The jumps in the distance function are a bit more intuitive to comprehend than the long step in the arrival curve but both have the same cause.

The second use case is a replica of the ATON [34] Optical Navigation subsystem. While the tasks themselves only contain placeholder code, the timing of the triggers, structure and scheduling method of the system are identical. The structure of the subsystem can be seen in Figure 5.7. For readability reasons, the inputs (green rectangles with rounded edges) are not labelled in the figure. The subsystem uses two cameras and a sensor measuring inertia to determine its position and write it to the flight controller and into a log. Since this structural example does not compute data, the inertia sensor is abstracted to a second timer.

The subsystem consists of seven tasks, five channels and two different triggers. The two camera tasks are triggered every 1000ms, while the navigation task is triggered every 100ms. The camera tasks both put their data on a channel that is connected to an image processing task. The output channel of the image processing tasks is connected to the navigation task. The navigation task pushes on the *OutPos* channel that is connected to two terminal tasks. One represents the logger of ATON while the other represents the flight controller. Figure 5.8 shows the Tasking Graph of the trace of an execution of the subsystem. In this Tasking Graph, the interaction

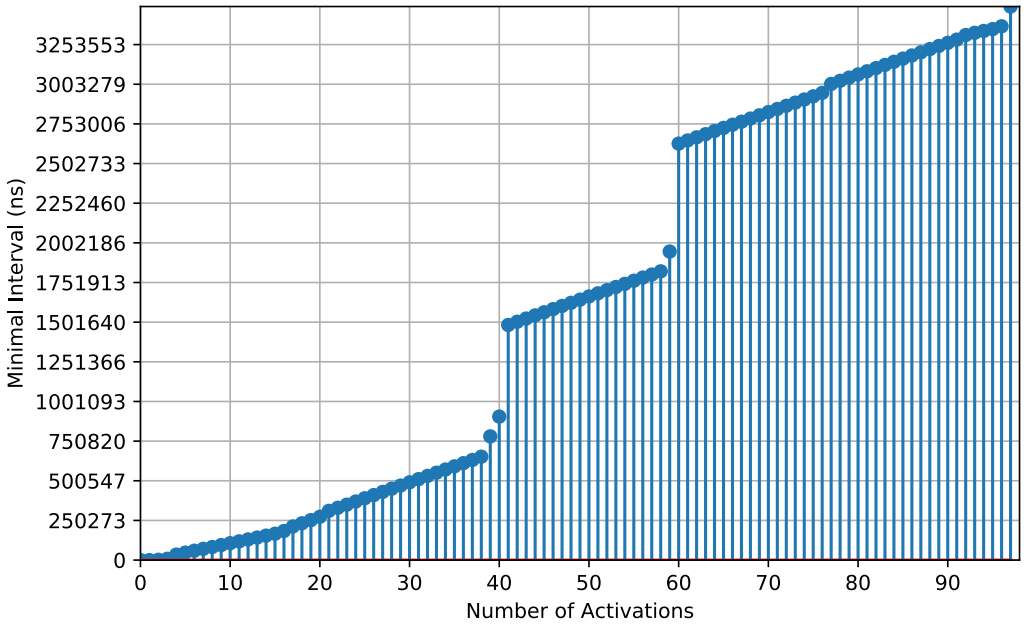

Figure 5.5: The distance function computed from the trace of the Fibonacci example.

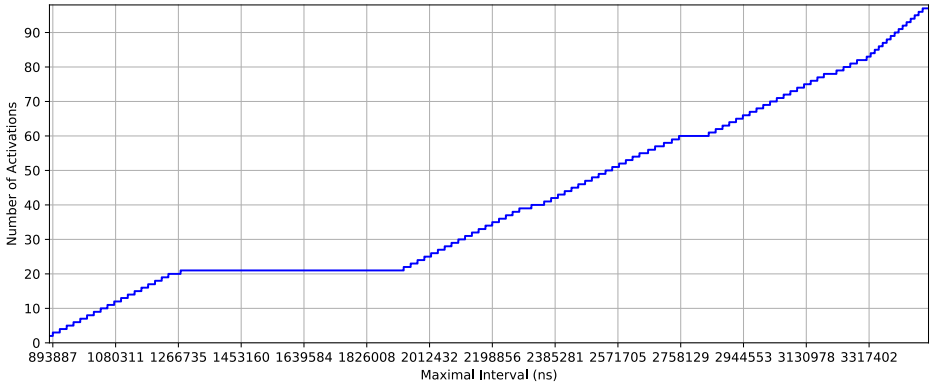

Figure 5.6: The arrival curve computed from the trace of the Fibonacci example.

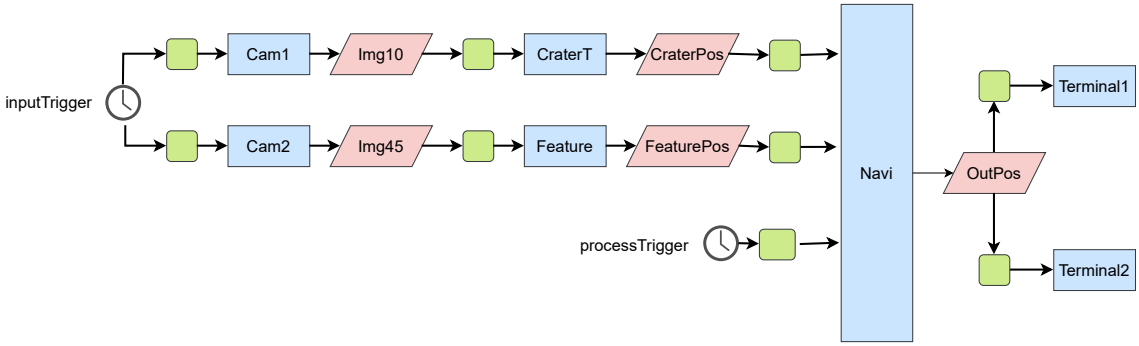

Figure 5.7: The structure of the ATON Optical Navigation subsystem modelled in Tasking Framework.

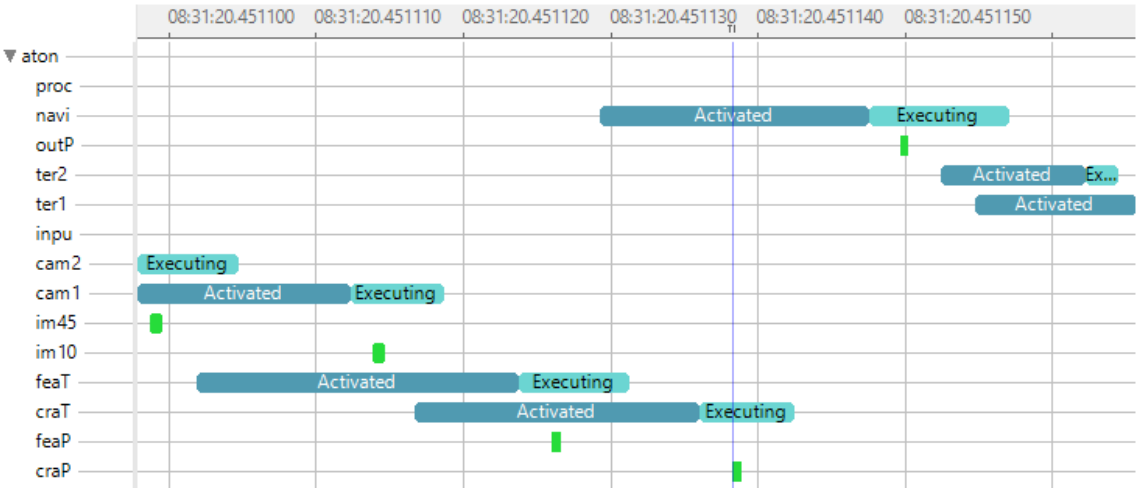

Figure 5.8: Part of the Tasking Graph for the ATON use case.

between tasks and channels is clearly visible. During execution, a task pushes data to a channel, leading to the activation of another task. An exception to this is the last push visible in the figure, the push on *OutP*, which activates both terminal tasks because both are solely connected to it (see Figure 5.7).

5.3 Profiling the traced Tasking Framework

As mentioned in Section 3.3, tracing produces an overhead, as the instrumentation of the code adds to the runtime of the system. In this section, the overhead caused

by the tracer is investigated using different profilers. There are other methods of measuring overhead, such as comparing overall execution times of two identical applications, with the only difference that one is traced and the other is not. However, profiling gives far more insight on how much of the execution time is spent where in the application, making it possible to make more nuanced observations.

5.3.1 Gprof

Gprof [14] is a profiler build into the GNU compiler that uses code instrumentation during compile time to profile its code. To get information about the execution of a program, gprof employs two main strategies: execution counts and execution timing. For the execution counts, gprof counts how many times an individual function is called during runtime. For the execution timing, rather than measuring the elapsed time from entering to exiting a function, gprof samples the execution counter to find out what function is currently executing. The sampling rate is, however, dependent on the system, gprof is running on. The sampling method was chosen for gprof because it is easier than measuring elapsed time on systems that employ time-slicing, i.e. pre-emptive scheduling. It is to note that all execution times that gprof infers this way are statistical and not exact times. Gprof can only profile one process at a time, meaning that all system calls that are made during execution are not counted into the time of the function that calls them. Gprof collects data on the following execution times: the total amount of time consumed by a function, the average time spent in one call of the function and the average time spent in one call of the function including all subroutines called on by the function.

Gprof can present its results in two ways: the flat profile and the static call graph. The flat profile is a table that lists all functions contained in the program along with the following information:

1. the relative time spend in the function in relation to the entire runtime in seconds,
2. the total amount of seconds spent in the function during the entire seconds which the list is sorted by,

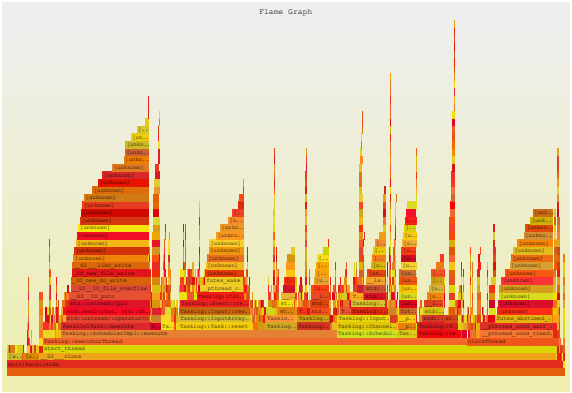

Figure 5.9: A flamegraph.

3. the running sum of time spent in all functions up until the listed function,
4. the number of calls made during the execution,
5. the average time spent in this function alone in milliseconds and
6. the average time spent in this function and all functions called from this function also in milliseconds.

This flat profile data representation will be used in this work to evaluate the profiling results. The other representation is the static call graph, that depicts the call tree, a representation of which function was called by which function, and how much time was spent in the different branches of the tree. A similar depiction of function call behaviour is the basis of the flamegraph, which is described in the following section.

5.3.2 Flamegraph

The flamegraph is a graphical representation of the call tree of a program. Functions are depicted as rectangles with their width corresponding to the amount of time spent in them or their children. Functions that are called by a particular function are represented by their rectangle being stacked upon the rectangle of the function that called it. The result of this representation are towers of function calls that dwindle in width the higher they grow, that look like flames, like in Figure 5.9. Hence, the name flamegraph. It is to note, that flamegraphs do not depict their data in relation

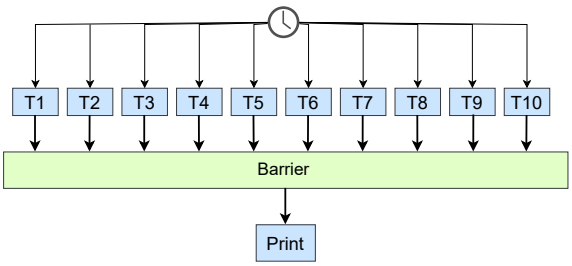

Figure 5.10: A model of the use case used to test the overhead of the tracer.

to time, meaning the position of a specific tower on the x-axis does not reflect on the temporal behaviour of the function.

In this work, flamegraphs are generated using a tool called *Flamegraph*, written in the Rust programming language. The tools executes the program and profiles it using *perf* [24], another profiling tool. The binary output produced by perf is then used to generate the flamegraphs. The specific subcommand of perf used for profiling is *perf record*, which samples the program, by default with a frequency of 1000Hz, and produces a binary output file. This output file can then be turned into a flamegraph and saved in different image file formats. Flamegraphs saved in the Scalable Vector Graphics format (file ending in .svg) can be interactively viewed in a browser, meaning that the name of each function, the number of times it was sampled and the percentage relative to the total runtime are displayed below the graph, even if the name can not fit on the rectangle.

5.3.3 Profiling a traced application

For the sake of testing the tracer, a use case is used, with the intention to write many events in a short timeframe. A model of this use case can be seen in Figure 5.10. This use case triggers ten tasks with a frequency of 1000Hz, that push on a barrier. The barrier activates once it reaches ten pushes and activates a printing task. The program terminates once the user enters a line of text.

An initial look at the flamegraph, see Figure 5.11, shows that more than 50% of the time spent in function calls belonging to the tracer, in the following referred to as

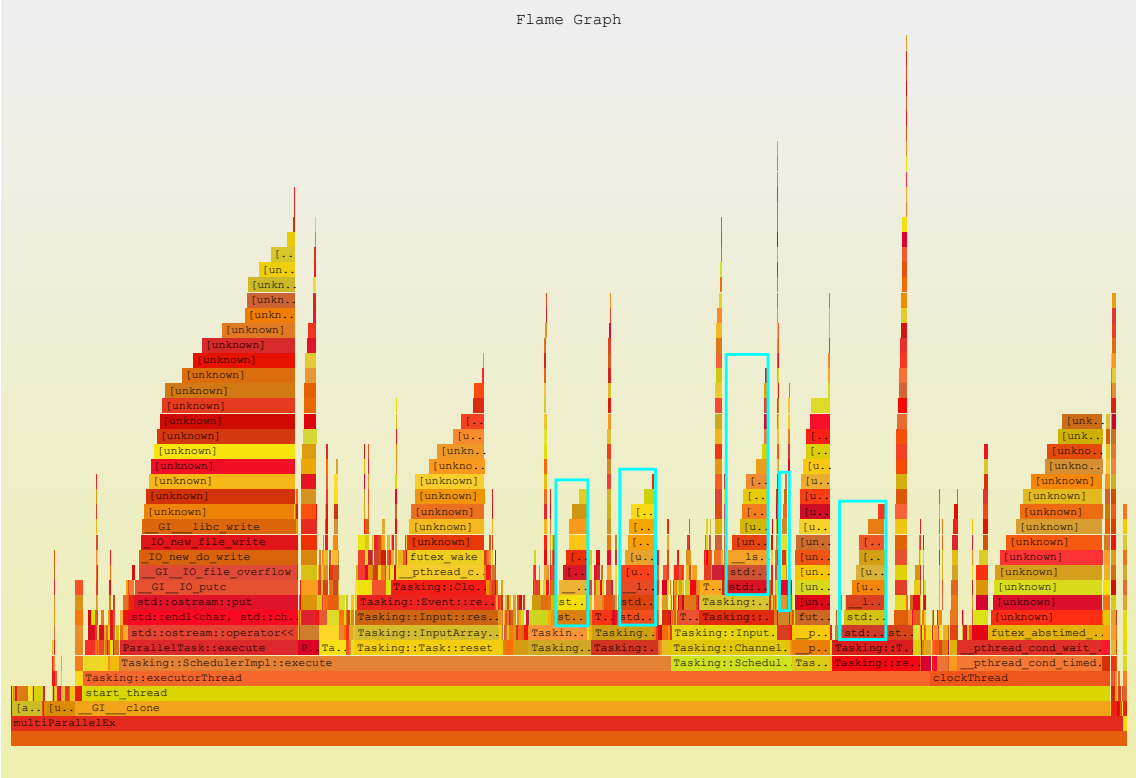

Figure 5.11: Initial flamegraph of the use case, with function calls of tellg marked in light blue.

tracing functions, are taken up by calls of a function called *tellg()*, which was used to determine the size of each event. The tracing functions make up around 26% of the runtime, meaning *tellg()* takes up more than 13% of the total runtime. To reduce the amount of overhead, a different way of determining the size of events was implemented to avoid the use of *tellg()*. This new implementation does not result in a loss of information for the tracer. The resulting flamegraph of the reworked tracer indicated an overhead of around 15% of the runtime. To investigate this further, the same application was also traced with gprof. See in the following excerpts of the output of gprof, that pertains to tracing functions:

```
1    %       self              self      total
2   time    seconds     calls  ms/call  ms/call  name
3   16.67      0.01    114826     0.09     0.09  Tasking::Tracer::writeEventHeader
4   16.67      0.01     25612     0.39     0.55  Tasking::recordTaskStopExecution
5    0.00      0.00     77376     0.00     0.09  Tasking::Tracer::recordTaskUtility
```

```
 6    0.00      0.00     28286      0.00      0.09  Tasking::Tracer::recordSwitchEvent
 7    0.00      0.00     28281      0.00      0.16  Tasking::recordSwitchEvent
 8    0.00      0.00     26037      0.00      0.13  Tasking::Tracer::
          recordTaskStartExecution
 9    0.00      0.00     26012      0.00      0.16  Tasking::recordTaskStartExecution
10    0.00      0.00     25682      0.00      0.13  Tasking::Tracer::
          recordTaskActivation
11    0.00      0.00     25669      0.00      0.16  Tasking::recordTaskActivation
12    0.00      0.00     25639      0.00      0.13  Tasking::Tracer::
          recordTaskStopExecution
13    0.00      0.00      4578      0.00      0.17  Tasking::Tracer::recordChannelPush
14    0.00      0.00      4577      0.00      0.17  Tasking::recordChannelPush
15    0.00      0.00         1      0.00      0.00  Tasking::Tracer::endTrace
16    0.00      0.00         1      0.00      0.00  Tasking::Tracer::initTrace
```

Listing 5.1: Excerpts of the gprof output for the use case. Note, that the values of *self ms/call* and *total ms/call* are averages for individual function calls, while *self seconds* depicts the total time spent in the function. The argument list of the functions has been left out for readability purposes.

It has to be kept in mind, that all profiling results shown here are computed from sampled data and that gprof and perf use different sampling rates, so results from both may differ. The function *Tasking::Tracer::writeEventHeader* (Listing 5.1, line 3) is called upon by every tracing function that records events (meaning all except the initTrace() and endTrace() functions), so the probability of this function being sampled is higher than for any of the other tracing functions. Additionally, the flat profile in gprof does not take into account subsequent function calls in all of its categories. All functions whose names do not start with *Tasking::Tracer* are wrapper functions that call their respective counterpart, meaning the time spent within them, excluding internal function calls should be minimal but the wrapper function *recordTaskStopExecution* is in second place in terms of execution time in this listing. To reduce a possibly occuring sampling error, the same application was again profiled with gprof but let run six times as long:

```
 1    %      self              self     total
 2   time    seconds    calls  ms/call  ms/call  name
 3   16.14     0.05  1198554     0.04     0.05  Tasking::Tracer::recordTaskUtility
 4    6.45     0.02   437587     0.05     0.05  Tasking::Tracer::recordSwitchEvent
 5    3.23     0.01  1781468     0.01     0.01  Tasking::Tracer::writeEventHeader
```

```
 6    3.23      0.01    401813      0.02      0.08  Tasking::Tracer::
          recordTaskStopExecution
 7    3.23      0.01    399588      0.03      0.08  Tasking::recordTaskActivation
 8    3.23      0.01     72555      0.14      0.22  Tasking::recordChannelPush
 9    0.00      0.00    437669      0.00      0.06  Tasking::recordSwitchEvent
10    0.00      0.00    401379      0.00      0.08  Tasking::recordTaskStopExecution
11    0.00      0.00    399673      0.00      0.05  Tasking::Tracer::
          recordTaskActivation
12    0.00      0.00    394721      0.00      0.05  Tasking::Tracer::
          recordTaskStartExecution
13    0.00      0.00    392702      0.00      0.06  Tasking::recordTaskStartExecution
14    0.00      0.00     72561      0.00      0.08  Tasking::Tracer::recordChannelPush
15    0.00      0.00         1      0.00      0.00  Tasking::Tracer::endTrace
16    0.00      0.00         1      0.00      0.00  Tasking::Tracer::initTrace
```

Listing 5.2: Excerpts of the gprof output with a larger sample size

As can be seen in Listing 5.2, except for the function pair responsible for recording an activation (lines 7 and 11), all wrapper functions are listed below their counterparts, meaning they were sampled less often. With more sample data, the granularity has also gone down, as can be seen in Listing 5.2, lines 4-8 as compared to Listing 5.1, lines 5-16, where functions with bigger average values for the whole function call (column *total ms/call*) did not receive values in the column *% time*.

Now, the 15-20% of overhead given by the profilers describe the offline overhead, meaning the total overhead, of the tracing functions. Far more relevant is the overhead that affects the execution time of a task and thus, the timing behaviour of the application. The overhead of the tracing functions that affect the timing behaviour is caused by three functions, namely the ones that record pushes on a channel and the start and stop of a task executing. Their overhead sums up to 5.8%-6.5%, depending on the profiler. Note, that the execution times of the tracing function are mostly fixed, since they always write the same amount of data. That means, if a task executes longer, the percentage of the overhead goes down accordingly. The use case uses dummy tasks with no significant computation times. Thus, the overhead shown here is likely to be an upper boundary on the possible overhead.

6 Conclusion

6.1 Summary

In this work, a tracer was built within Tasking Framework upon an already existing tracing mechanism. Key points of Tasking Framework that needed tracing were identified to realize the implementation of the tracer. The tracing of these key points was realized using custom trace events, since the approach of repurposing existing Linux kernel tracepoints failed. To display the traces generated by Tasking Framework applications, a custom chart was implemented for TraceCompass. Furthermore, a Python script was developed to analyse traces of Tasking Framework applications further. These analyses include information about the execution times of tasks, arrival curves and minimum distance functions, that are available over a command line interface and can, in case of the arrival curves and distance functions, also be plotted. These analyses can all be applied on the entirety of the trace or on a specific task. Next to the analyses of the traces, a safe extrapolation mechanism was developed out of the definition of the arrival curves and distance functions.

All implemented features were then tested on use cases. Another use case was used in conjunction with different profiling applications to determine the overhead of tracer, find expensive function calls and try to reduce the overhead. The offline overhead of the tracer is currently at around 15-20% of the total runtime, with the overhead for a task executing being around 5-6%.

6.2 Outlook

Future work on this tracer could work on reducing the overhead of the tracer. An idea for reducing the overhead is that, instead of writing every event directly into a file, all events are buffered and upon completion of the trace are then written into a file, since writing to a file is quite expensive in terms of computing time. Even though CTF is a binary format and thus very compact and memory efficient, this could lead to memory overflow in the Random Access Memory (RAM) before the trace is written, causing data to be lost, especially when tracing for longer periods of time or applications with a high density of events. A compromise on the current, more costly, approach to write everything directly to file and the faster, though less safe, approach of buffering until the tracing is completed, could be to buffer a certain amount of events and periodically write them to file as needed, preventing to many events being held in the RAM.

A second approach to reducing the overhead in the timing behaviour would be to not alter the behaviour of the tracer, but rather to subtract the minimum overhead of the tracing functions from the time spent in the recorded events after the trace has finished recording. This is possible, since the tracing functions have a mostly fixed execution time, as the amount of data recorded with a tracing function always stays the same. Variations on this execution time can be accredited to an underlying operating system. Hence, the minimum overhead can be safely subtracted from execution and activation times.

Another point to work on, would be to find complete activation chains with tracing. Meaning, to find the exact chain of cause and effect from a push to the input notification, over task activation all the way to the task executing and pushing on a channel. To complete this step, an additional event would be needed, namely the notification of an input, which would have to include the name of the channel that the push is originating from. This is currently not possible in Tasking Framework and would require additional changes to Tasking Framework to realize. If this information was available, however, it could also be graphically included into the Tasking Graph as arrows leading from the activating push to the *Activated* state of the task.

Another application of the tracer would be part of a feedback loop with the Tasking Modelling Language [10], that can generate Tasking Framework code. The tracer could then be used to refactor the generated code.

Acronyms

AOCS Attitude and Orbit Control System

ATON Autonomous Terrain-based Optical Navigation

BIRD Bispectral Infra-Red Detection

CLI Command Line Interface

CPN Coloured Petri net

CTF Common Trace Format

DLR Deutsches Zentrum für Luft- und Raumfahrt

EASE Eclipse Advanced Scripting Environment

Eu:CROPIS Euglena Combined Organic food Production In Space

FIFO First-In First-Out

LIFO Last-In First-Out

LTTng Linux Trace Toolkit: next generation

MAIUS Matter-Wave Interferometry in Weightlessness

OSS Onboard Software Systems

RAM Random Access Memory

RCP Rich Client Platform

RTOS Real Time Operating Systems

SC Institute for Software Technology

ScOSA Scalable On-Board Computing for Space Avionics

TSDL Trace Stream Description Language

Bibliography

[1] *About Page of the DLR Homepage*. 08/22/2023. URL: https://www.dlr.de/de/das-dlr/ueber-uns/organisation.

[2] Boyuan Chen and Zhen Ming Jiang. "A survey of software log instrumentation". In: *ACM Computing Surveys (CSUR)* 54.4 (2021), pp. 1–34.

[3] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. "Event logs for the analysis of software failures: A rule-based approach". In: *IEEE Transactions on Software Engineering* 39.6 (2012), pp. 806–821.

[4] Edmund Clarke et al. "Progress on the state explosion problem in model checking". In: *Informatics: 10 Years Back, 10 Years Ahead* (2001), pp. 176–194.

[5] Bas Cornelissen et al. "A systematic survey of program comprehension through dynamic analysis". In: *IEEE Transactions on Software Engineering* 35.5 (2009), pp. 684–702.

[6] Wim De Pauw and Steve Heisig. "Zinsight: A visual and analytic environment for exploring large event traces". In: *Proceedings of the 5th international symposium on Software visualization*. 2010, pp. 143–152.

[7] Mathieu Desnoyers. *Common Trace Format (CTF) Specification (v1.8.3)*. 07/20/2022. URL: https://diamon.org/ctf/#specification.

[8] Jonas Fabian Diemer. *Predictable architecture and performance analysis for general-purpose networks-on-chip*. Verlag Dr. Hut, 2016.

[9] Naser Ezzati-Jivan, Genevieve Bastien, and Michel R Dagenais. "High latency cause detection using multilevel dynamic analysis". In: *2018 Annual IEEE International Systems Conference (SysCon)*. IEEE. 2018, pp. 1–8.

[10] Tobias Franz et al. "Tasking Modeling Language: A toolset for model-based engineering of data-driven software systems". In: *OBDP2021 - 2nd European Workshop on On-Board Data Processing*. 2. 06/2021. URL: https://elib.dlr.de/145077/.

[11] Vijay Gehlot and Carmen Nigro. "An introduction to systems modeling and simulation with Colored Petri Nets". In: *Proceedings of the 2010 Winter Simulation Conference*. 2010, pp. 104–118. DOI: 10.1109/WSC.2010.5679170.

[12] Claude Girault and Rüdiger Valk. *Petri nets for systems engineering: a guide to modeling, verification, and applications*. Springer Science & Business Media, 2013. Chap. 1-2.

[13] Ivo Gomes et al. "An overview on the static code analysis approach in software development". In: *Faculdade de Engenharia da Universidade do Porto, Portugal* (2009).

[14] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. "Gprof: A Call Graph Execution Profiler". In: *SIGPLAN Not.* 17.6 (06/1982), pp. 120–126. DOI: 10.1145/872726.806987. URL: https://doi.org/10.1145/872726.806987.

[15] Zain A. H. Hammadeh et al. "Event-Driven Multithreading Execution Platform for Real-Time On-Board Software Systems". In: *Proceedings of the 15th annual workshop on Operating Systems Platforms for Embedded Real-time Applications*. 07/2019, pp. 29–34. URL: https://elib.dlr.de/128249/.

[16] Zain Alabedin Haj Hammadeh. "Deadline Miss Models for Temporarily Overloaded Systems". PhD thesis. Technische Universität Braunschweig, 2019.

[17] Zain Alabedin Haj Hammadeh. *Tasking Framework — Documentation*. 07/18/2022. URL: https://github.com/DLR-SC/tasking-framework/blob/master/doc/Tasking_Framework.docx.

[18] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of computer programming* 8.3 (1987), pp. 231–274.

[19] *Homepage of the Institute of Software Technology*. 08/22/2023. URL: https://www.dlr.de/sc/en/desktopdefault.aspx/.

[20] Brittany Johnson et al. "Why don't software developers use static analysis tools to find bugs?" In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 672–681.

[21] Lea Jungmann. "Visualization of a 3D Map and a Drone Flight Path in Augmented Reality & Integration of a Tracing Mechanism Into The Tasking Framework". Student Research Paper submitted to DHBW Mannheim. 2022.

[22] Simon Künzli and Lothar Thiele. "Generating event traces based on arrival curves". In: *13th GI/ITG Conference-Measuring, Modelling and Evaluation of Computer and Communication Systems*. VDE. 2006, pp. 1–18.

[23] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer, 2001.

[24] *Main Page of the perf Wiki*. 08/16/2023. URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[25] Charles E McDowell and David P Helmbold. "Debugging concurrent programs". In: *ACM Computing Surveys (CSUR)* 21.4 (1989), pp. 593–622.

[26] Antonio Pecchia et al. "Industry practices and event logging: Assessment of a critical software development process". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 2. IEEE. 2015, pp. 169–178.

[27] Wolfgang Reisig. *Understanding petri nets*. Springer, 2016, pp. 13–14.

[28] Wolfgang Reisig and Grzegorz Rozenberg. *Lectures on petri nets i: basic models: advances in petri nets*. Springer Science & Business Media, 1998.

[29] *The Babeltrace 2 Documentation*. 08/24/2022. URL: https://babeltrace.org/.

[30] *The checkstyle Documentation*. 08/02/2023. URL: https://checkstyle.sourceforge.io/.

[31] *The Eclipse EASE Wiki*. 07/31/2023. URL: https://wiki.eclipse.org/EASE.

[32] *The LTTng Documentation*. 08/26/2022. URL: https://lttng.org/docs/v2.13/.

[33] *The TraceCompass User Guide*. 08/01/2023. URL: https://archive.eclipse.org/tracecompass.incubator/doc/org.eclipse.tracecompass.incubator.scripting.doc.user/User-Guide.html.

[34] Stephan Theil et al. *Project ATON (Autonomous Terrain-based Optical Navigation): Final Report*. Tech. rep. Institute of Space Systems, 04/2018, pp. 116–119. URL: https://elib.dlr.de/138972/.

[35] Neil Walkinshaw et al. "Improving dynamic software analysis by applying grammar inference principles". In: *Journal of Software Maintenance and Evolution: Research and Practice* 20.4 (2008), pp. 269–290.