Trace Collection, Analysis, and Visualization for Barrelfish

Distributed Systems Lab

David Stolz
stolzda@student.ethz.ch

Alexander Grest
agrest@student.ethz.ch

February 18, 2013

Abstract

Obtaining detailed information about the internal events occurring in an operating system is a prerequisite for understanding performance, overhead, and many subtle timing- related bugs and race conditions.

Barrelfish is a research operating system that is developed by ETH Zurich in collaboration with Microsoft Research. It has a simple tracing mechanism which works together with a visualization tool. The current tracing infrastructure for Barrelfish is not very sophisticated and unable to handle traces longer than a few thousand processor cycles.

In this lab, we extend the tracing functionality provided in Barrelfish and create a more future-rich visualization and analysis tool for the data.

Contents

Chapter 1

Introduction

1.1 Motivation

Obtaining detailed information about the internal events occurring in an operating system is a prerequisite for understanding performance, overhead, and many subtle timing- related bugs and race conditions. Most operating systems provide some kind of tracing infrastructure, e.g. DTrace for Solaris or FTrace for Linux.

It has been proposed to use a message-passing primitive instead of shared memory for communication inside an operating system as an response to increasing core counts, more heterogeneous hardware and less uniform memory systems [2]. Operating systems that use message-passing instead of shared memory such as Barrelfish are particularly suited for tracing. Because message-passing makes communication between components of the operating system explicit, it is easier to capture, visualize and analyse communication flows compared to an operating system that uses shared memory for communication.

1.2 The Barrelfish Operating System

We present an overview of the barrelfish operating system. This overview is many taken from [6].

Barrelfish is a research operating system developed in cooperation between the Swiss Federal Institute of Technology Zurich (ETH) and Microsoft Research. It embraces the networked nature of the machine and rethinks operating system architecture using ideas from distributed systems. Barrelfish is an implementation of the multikernel architecture. In a nutshell, the operating system is structured as a distributed system of cores which communicate using messages and share no memory. The multikernel architecture is guided by three design principles: Make all inter-core communication explicit, make OS structures hardware-neutral and view state as replicated instead of shared [1]. Each core runs its own kernel which is called a ”CPU driver”. The CPU driver runs in privileged mode and enforces protection, performs authorization, time-slices processes, and handles interrupts, page-faults, traps, and exceptions. It is single threaded, event driven and non-preemptable.

1.2.1 Monitor

On every core runs a distinguished user-mode monitor process. All inter-core coordination is performed by monitors. Monitors collectively coordinate system-wide state and encapsulate much of the mechanism and policy to be found in a typical monolithic kernel. On each core, replicated data structures, such as memory allocation tables, are kept globally consistent by means of an agreement protocol run by the monitors.

1.2.2 Dispatcher

A process is represented by a collection of dispatcher objects, one on each core on which it might execute. Dispatchers on a core are scheduled by the local CPU driver, invoking an upcall interface that is provided by each dispatcher.

1.2.3 Inter-dispatcher communication

Communication in Barrelfish is not between processes but between dispatchers and hence cores. All inter-dispatcher communication occurs with messages. Messages are carried over Interconnect Drivers (ICDs), specialized messaging subsystems which carry small data units between dispatchers. Interconnect drivers are highly optimized for particular hardware and do not expose a standard interface. Instead, interconnect drivers are abstracted behind a common interface, allowing messages to be marshalled, sent and received in a driver-independent way. As with conventional RPC systems, the interface for a particular communication binding is specified in an Interface Definition Language. A stub compiler, called flounder, compiles defined interfaces into a set of ICD-specific stubs.

Communication between dispatchers on the same core is performed using LMP (local message passing). Communication between dispatchers on different cores is performed using a variant of user-level RPC which uses shared memory to transfer cache-line-sized messages.

1.3 Current state of the tracing infrastructure in Barrelfish

The tracing library is tightly integrated with the Barrelfish operating system. It is statically linked into every application and can be used to log events and dump trace logs. The code of several applications designed to run on Barrelfish is instrumented with custom trace points. The kernel code also contains trace points for essential events such as context switches.

The application "Bfscope" has a special connection to the tracing library: Bfscope is a network server that acts as an interface to external machines. When connects to Bfscope, the output of the tracing framework is redirected via Bfscope, and directly forwarded to the remote machine. This allows developers to directly retrieve the output of the tracing library on a remote machine. Note however that running Bfscope is not mandatory, i.e. the tracing library is not depending on Bfscope in order to run; Bfscope simply adds network support to the tracing library.

Once users retrieve such logs (either by dumping them to the console or via Bfscope), they want to analyze them. Aquarium is a tool that allows to visualize traces. It can load a trace from a file or connect to Barrelfish machine running Bfscope.

The different components of the tracing infrastructure depicted in Figure 1.1.

1.3.1 Limitations

The current tracing infrastructure has issues that reduce code maintainability and limits its effectiveness for understanding performance, overhead or timing-related bugs.

Events are defined as constants

Events are defined as hard-coded constants which are distributed all over the code base. When defining new events, there is nothing that makes sure that these definitions do not conflict with already existing events. Events are also

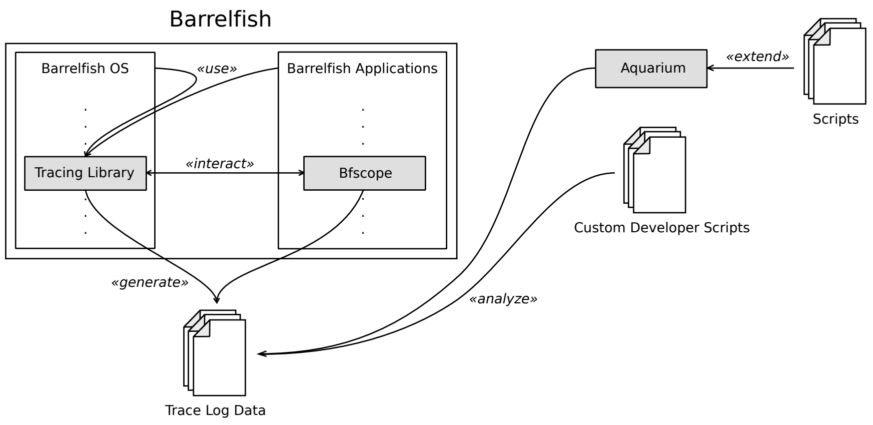


Figure 1.1: Overview how the different parts of the tracing framework interact. The grey systems (Tracing Library, Bfscope and Aquarium) are parts of the tracing framework that have been developed or modified in the course of this project.

hard-coded in the Aquarium visualization tool, which makes it practically impossible to change existing event definitions.

No support for continuous tracing

The tracing infrastructure is designed for ”one-shot” tracing in an interval between two well-defined events. After that interval buffers have to be flushed and reset. We argue that in certain situations continuous tracing of all events can be useful.

No support for excluding certain events from tracing

If tracing is turned on, all events are logged. It is not possible to exclude certain events.

Use of inter-processor interrupts (IPI)

When flushing buffers, the current tracing infrastructure sends interrupts to all cores. Upon receipt of such an interrupt, the kernel running on that particular core determines all running application and adds that information to the tracing buffer. This is a weird and unnecessary use of inter-processor interrupts that makes the implementation overly complex and platform dependant.

Limited feature set of Aquarium

Aquarium can visualize events and messages sent between cores. But it lacks useful features like showing only certain events that match a filter, grouping events belonging to the same task together, etc.

Performance issues of Aquarium

The Aquarium tool has sever performance issues. It frequently crashes when loading trace files that are larger than a few thousand processor cycles. Even when it successes loading a large trace file, analysing it with the help of Aquarium is practically impossible because zooming and moving the current view window is very slow.

Some developers have written custom scripts that they use instead of Aquarium to analyse trace file. However, this allows pure textual analysis and does not produce any visualizations.

Platform dependency of Aquarium

Aquarium is written for the Common Language Runtime. As most developers on the Barrelfish team work on Linux-based operating systems, they can not use Aquarium without using Mono (an open-source implementation of the Common Language Runtime) or running Aquarium on a Windows terminal server, making its performance problems even worse. Developers on the Barrelfish team have cited this as the most important reason why they don't use Aquarium for day-to-day development.

1.4 Aim

The goal of this project is to improve the existing tracing infrastructure in Barrelfish and to address the issues mentioned above.

We want to change the interface to the tracing subsystem as little as possible. Changes that would brake the existing code base are avoided. Also the structure of the log files is not changed significantly. This ensures that the existing scripts can be easily adapted to work with the new tracing infrastructure.

We develop a successor of Aquarium that is superior to the original version both in terms of performance and functionality.

In addition to the existing scripts, we developed a new tool named Aquarium, which is the successor of a tool with the same name. Besides offering various functionality (see Section 4), it can be extended by adding custom scripts to it. With the help of such scripts it should be possible to add many of the desired functionality to Aquarium without even changing its code (which is, of course, also a possibility).

Chapter 2

Related Work

2.1 Ftrace

Ftrace is an internal tracing framework for the Linux kernel. Ftrace was originally a function tracer but it now includes an infrastructure that allows for many other types of tracing, such as tracing context switches between tasks, tracing areas that disable interrupts and event tracing. In addition to that, Ftrace introduced `trace_printk()`, which can be used to write arbitrary output to the Ftrace ring buffer. This has a big performance advantage over `printk()` that was traditionally used for kernel debugging.

The function tracer works by having each function in the kernel call a special function `mcount()` [8]. All these calls are converted to a NOP at boot time to keep the system running at 100 % performance. When the function tracer is enabled, these call sites are converted back to trace calls.

```
# tracer: function
#
#          TASK-PID    CPU#      TIMESTAMP   FUNCTION
#              |        |            |           |
gnome-shell-1682  [003]  2019.094740: finish_task_switch
                                           <-__schedule
gnome-shell-1682  [003]  2019.094740: prepare_to_wait
                                           <-i915_wait_request
gnome-shell-1682  [003]  2019.094740: _raw_spin_lock_irqsave
                                           <-prepare_to_wait
gnome-shell-1682  [003]  2019.094741: _raw_spin_unlock_irqrestore
                                           <-prepare_to_wait
gnome-shell-1682  [003]  2019.094741: gen6_ring_get_seqno
                                           <-i915_wait_request
```

Listing 2.1: Sample output from the function tracer

The event tracer records events when the kernel steps on a ”tracepoint” embedded within the kernel. The Linux kernel currently contains more then 300 static tracepoints that are located in the scheduler, memory manager, file system, etc. A tracepoint is per default ”off” and has no effect except for adding a tiny performance overhead [3]. Tracepoints that should be recorded must be explicitly turned on.

2.1.1 trace-cmd

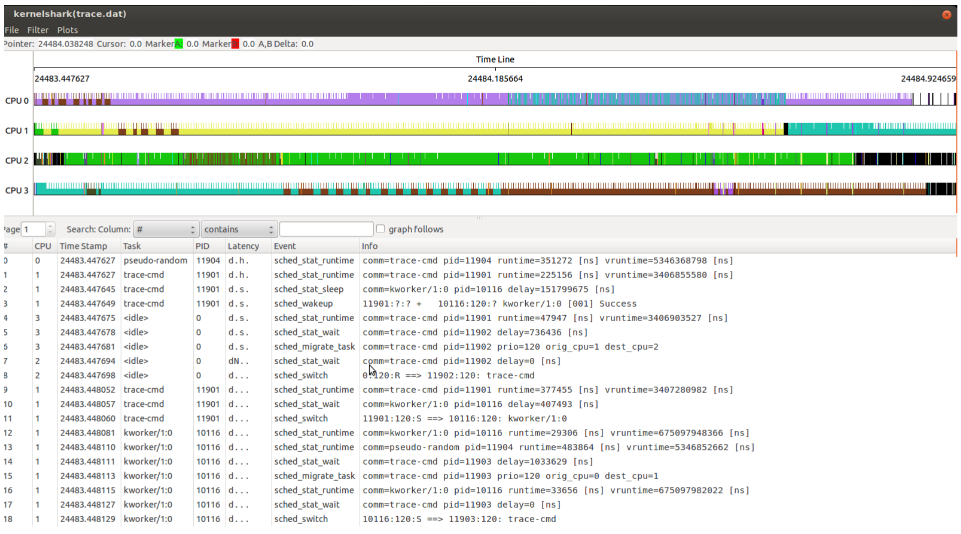
The API to interface with Ftrace is located in the Debugfs file system. This interfaces is very simple, but can be awkward to work with. `trace-cmd` provides more convenience. It is a command line front-end for Ftrace [10]. A standard use case for `trace-cmd` is to enable tracepoints for Ftrace and record the Ftrace data (typically in a file called `trace.dat`).

2.1.2 KernelShark

KernelShark is a GUI front end to `trace-cmd`. KernelShark can visualize data recorded with `trace-cmd` [11]. This can make it a lot easier to understand complex interactions inside the kernel.

KernelShark contains a graph view and a list view. In the graph view, each CPU visible to the operating system is represented by a plot line and each task is represented by a different color [9]. This makes it easy to determine which task was running on which CPU at any given point in time. The user can filter out any task that he is not interested in. Moreover, KernelShark also visualizes recorded events. If there are too many events within the resolution of the graph, the plots will appear as a rainbow coloured bar. It is possible to zoom into the graph to make more sense of the output in such a situation.

The list view displays all recorded events. Each entry in the list contains the time stamp of the event, the process ID of the task that was running when the event was recorded, the name of the event, etc. The list view can be configured with sophisticated filters to only show the events of interest.



Pointer: 24484.038248 Cursor: 0.0 MarkerA: 0.0 MarkerB: 0.0 A,B Delta: 0.0

Page 1 Search: Column: # contains

#	CPU	Time Stamp	Task	PID	Latency	Event	Info
0	0	24483.447627	pseudo-random	11904	d.h.	sched_stat_runtime	comm=trace-cmd pid=11904 runtime=351272 [ns] vruntime=5346368798 [ns]
1	1	24483.447627	trace-cmd	11901	d.h.	sched_stat_runtime	comm=trace-cmd pid=11901 runtime=225156 [ns] vruntime=3406855580 [ns]
2	1	24483.447645	trace-cmd	11901	d.s.	sched_stat_sleep	comm=kworker/1:0 pid=10116 delay=151799675 [ns]
3	1	24483.447649	trace-cmd	11901	d.s.	sched_wakeup	11901:?:? + 10116:120:? kworker/1:0 [001] Success
4	3	24483.447675	<idle>	0	d.s.	sched_stat_runtime	comm=trace-cmd pid=11901 runtime=47947 [ns] vruntime=3406903527 [ns]
5	3	24483.447678	<idle>	0	d.s.	sched_stat_wait	comm=trace-cmd pid=11902 delay=736436 [ns]
6	3	24483.447681	<idle>	0	d.s.	sched_migrate_task	comm=trace-cmd pid=11902 prio=120 orig_cpu=1 dest_cpu=2
7	2	24483.447694	<idle>	0	dN..	sched_stat_wait	comm=trace-cmd pid=11902 delay=0 [ns]
8	2	24483.447698	<idle>	0	d...	sched_switch	0:120:R ==> 11902:120: trace-cmd
9	1	24483.448052	trace-cmd	11901	d...	sched_stat_runtime	comm=trace-cmd pid=11901 runtime=377455 [ns] vruntime=3407280982 [ns]
10	1	24483.448057	trace-cmd	11901	d...	sched_stat_wait	comm=kworker/1:0 pid=10116 delay=407493 [ns]
11	1	24483.448060	trace-cmd	11901	d...	sched_switch	11901:120:S ==> 10116:120: kworker/1:0
12	1	24483.448081	kworker/1:0	10116	d...	sched_stat_runtime	comm=kworker/1:0 pid=10116 runtime=29306 [ns] vruntime=675097948366 [ns]
13	1	24483.448110	kworker/1:0	10116	d...	sched_stat_runtime	comm=pseudo-random pid=11904 runtime=483864 [ns] vruntime=5346852662 [ns]
14	1	24483.448111	kworker/1:0	10116	d...	sched_stat_wait	comm=trace-cmd pid=11903 delay=1033629 [ns]
15	1	24483.448113	kworker/1:0	10116	d...	sched_migrate_task	comm=trace-cmd pid=11903 prio=120 orig_cpu=0 dest_cpu=1
16	1	24483.448115	kworker/1:0	10116	d...	sched_stat_runtime	comm=kworker/1:0 pid=10116 runtime=33656 [ns] vruntime=675097982022 [ns]
17	1	24483.448127	kworker/1:0	10116	d...	sched_stat_wait	comm=trace-cmd pid=11903 delay=0 [ns]
18	1	24483.448129	kworker/1:0	10116	d...	sched_switch	10116:120:S ==> 11903:120: trace-cmd

Figure 2.1: Screenshot of KernelShark

2.1.3 Evaluation

Ftrace, in combination with `trace-cmd` and KernelShark, is a powerful tool for debugging the Linux kernel. It can be used as a profiler and help to identify performance bottlenecks. The extensive support for event tracing can also be used to get an understanding of complex operations (such as task migration) that happen in the kernel. This makes finding problem areas or tracking down a bug easier. The ability to record events that lead to a crash gives a better chance of finding exactly what caused the crash [8].

2.2 DTrace

DTrace is a kernel tracing framework created my Sun Microsystems for Solaris, but it has since been ported to other Unix-like operating systems. There are two common modes of operation: Trace system calls and trace kernel function calls. It can also be used for event tracing. DTrace is similar to Ftrace but exists in its own right because of licensing conflicts [4].

2.3 DProf

DProf [7] is a statistical profiling tool for Linux that can be used to analyze the memory efficiency of a system. It depends heavily on hardware support. It requires both the availability of Debug Registers and the possibility to use AMD's Instruction Bases Sampling (IBS) functionality.

Just like other memory profilers, DProf provides information to the user about the amount of cache misses, and which code and memory address where involved in the cache miss. The main goal of Dprof is also its main difference to other profilers: It tries to present the user with not only the information about which memory *address* led to the cache miss, but also which *data object* led to the cache miss. With the help of this information it is a lot easier for users to determine the objects that are responsible for memory issues, rather than having presented only memory addresses, or even simply the fact that memory caches happened.

Having the information available to which type of object a certain cache miss belongs, it is also possible to aggregate this information, allowing to present the user a condensed view. A view that only shows aggregated information per datatype can be used as an "entry-point" in understanding where the cache misses happen.

2.3.1 Data Collection

During the execution of a program two different categories of information are collected by DProf. They call the first category *access samples.* Access samples contain information about whether a given access to a memory location led to a cache miss or not. This information is collected with the help of IBS. Note that it is not DProf, but IBS, that decides when to generate the samples – DProf can only instrument IBS in such a way, that it sets the rate with which access samples are generated.

The second category are *access histories.* Access histories list all the instruction pointers that accessed a given memory address. Such histories are collected with the help of Debug Registers. Debug Registers allow DProf to trace a specified memory address, such that always when this memory address is accessed by a core, DProf receives a notification.

After having collected enough access samples and access histories, DProf merges the collected information to generate aggregated information. This step is necessary for two reasons: First it is obviously not very useful to present a huge list of single events to the user, and second it is a plain necessity. As the collection techniques are based widely on randomness[1], an aggregation step is required to alleviate this randomness.

2.3.2 Restrictions and Drawbacks

In order for the type detection (the mapping from address locations to object types) to work, it is assumed that all objects have a memory layout like C structs. This restriction does not only narrow the choice of language and compiler down significantly, it also makes it difficult to analyze complicated (e.g. nested) data structures.

The overhead of certain mechanics of DProf, such as setting up Debug Registers in order to trace a given memory address, is huge. Setting up Debug Registers costs between 130'000 and 220'000 cycles, and must be done for each data type multiple times, as enough statistical data must be collected. Even the faster operations, such as reacting to an IBS event, still take about 2'000 cycles.

As the overhead of DProf is quite significant, the collection rate might not be too high – i.e. DProf must be configured in such a way, that the profiling does not alter the actual performance too severely. As still enough data must be collected in order to achieve usable results, the program must be run for a long period of time. The time period must be indeed so long, that they require the workload to be cyclic [7]. This might be feasible for some of the applications, but excludes a wide variety of typical applications.

2.3.3 Evaluation

DProf allows for detailed memory profiling, but the used hardware support does not only limit the choice of architecture, it is also not a lightweight profiler. It might be useful as a special tool to have in your toolbox, but not for everyday profiling, where a more lightweight and general purpose profiling tool will be more useful.

[1]IBS decides when and what will be sampled and the tracked memory locations may not have generated a lot of information for the access history, if they weren't accessed anymore.

2.4 Google Perftools – PProf

When profiling a single application, the call graph is often the first thing that you want to analyze. For a given program, the call graph shows for each function how often it invoked its callees, and in addition, how much time was spent in each function. Such a call graph often allows for easy detection of bottlenecks in the application.

To generate a completely accurate call graph, one would have to instrument the code in such a way, that for each function call a logging mechanism is invoked. A different approach is to select a certain interval at which the code is analyzed, and to store the according call trace. The latter approach is used by the Google Perftools. Per default, every 10 milliseconds the program is interrupted and analyzed to determine its current call trace. Note that this is also a statistical profiler.

The profiling output can afterwards be visualized using the tool pprof, e.g. to draw the callgraph for the run of the application. The functionality of the Google Perftools CPU profiler is hence very similar to the well known GNU tool gprof [5].

Chapter 3

Design and Implementation of the Tracing Framework in Barrelfish

3.1 Overview

The tracing framework inside of Barrelfish existed already before this project has been started. In order to break as little as possible in existing code to work with the tracing system (e.g. tools that have been developed analyzing trace logs) we decided to change as little as possible on the interface of the tracing framework. In the end the structure of the trace logs that are generated did not change, but only some mappings between constants in code and their interpretation.

In this section we want to look at the part of the tracing framework that is implemented in Barrelfish, i.e. the actual functionality that developers use in order to create trace logs. One part of the tracing framework allows developers to trace events at any point in the code, where the data that is actually stored is defined in the Section 3.2. The second part is responsible for delivering the generated trace logs to Aquarium. To achieve the second goal we changed the existing Barrelfish application Bfscope in such a way, that it integrates with the new version of the tracing framework. Bfscope is described in Section 3.5.

The typical lifecycle of using the tracing framework in Barrelfish looks like this:

0.a (Optional) Prepare the tracing framework.

0.b (Optional) Specify which Subsystems should be logged.

1. Define the type of event that will trigger the start of tracing.

2. *Execute your code, that will log the events.*

3. Flush the logged events, e.g. on the console.

To get more information about the optional steps, see Sections 3.4.1 and 3.4.2. The first mandatory step is to define the type of the event that will start the logging process. Having a mechanism for starting and stopping the actual tracing may seem like a benefit, but not like a necessity at first – but our experiments have shown that even with rather small instrumentation of code (i.e. number of events that actually generate an entry in the trace log), having the tracing framework log events all the time is no option. Thus having the possibility to start and stop the tracing framework is essential. Having the flexibility of specifying a type of trace event that will trigger the start and stop of the logging is an additional benefit compared to having simple "start" and "stop" commands, as it allows developers to easily vary the portion of code they want to trace, without changing the placement of a "start" and "stop" command all the time.

While the second mandatory step is pretty self-explanatory, the third step is more interesting again: The old version of the tracing system allowed only for dumping the stored trace into a provided buffer. This functionality has now been improved in such a way that we offer developers a method to flush the current trace log, and the flush functionality automatically detects the best way to flush. Currently there are two possible destinations onto which can be flushed: The console and the network. The tracing framework detects automatically if Bfscope is running and someone is connected to it – if so, it flushes over the network – else it will flush to the console. The flushing functionality could also be extended, a possible idea would be to store the trace log in a file. In Figure 3.1 you can see a sequence diagram illustrating the process of invoking the new flushing functionality.

3.2 Definition of a Trace Event

Let us now define the structure of events that can be traced. Each event belongs to a Subsystem and an Event, and has an optional payload called Argument that can hold 32 bits of arbitrary information. Both the Subsystem and the Event are 16 bit identifiers, allowing us to have up to 65535 Subsystems and for each Subsystem 65535 Events. Note that the Events are relative to the Subsystems, i.e. a Subsystem called *kernel* might have the Event *Context Switch* with the identifier 0, but the same Event identifier 0 has an entirely different meaning in every other Subsystem.

Having all these different Subsystem and Event identifier available, we think

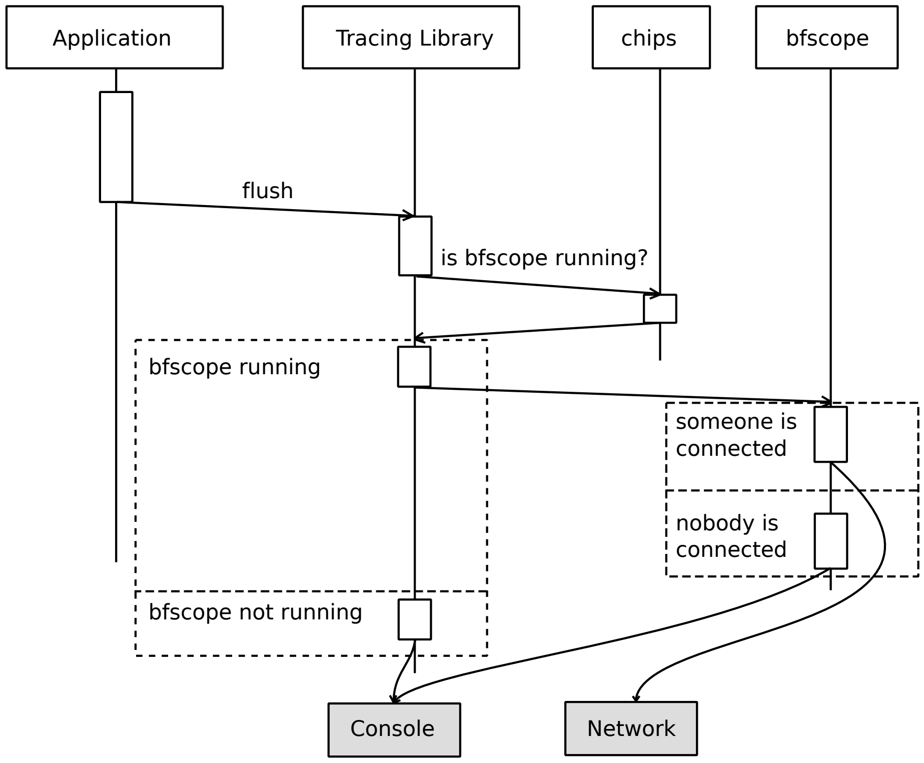


Figure 3.1: A sequence diagram illustrating the flow of events when using the flush functionality. “Application” is the application that is using the tracing framework, and chips is the Barrelfish nameserver. The grey boxes indicate the destination onto which is flushed.

that the tracing framework will have sufficient space to deal with future change in Barrelfish[1].

In addition to the Subsystem, Event and Argument information, the tracing framework adds a timestamp to each event that gets logged (the timestamp is measured in CPU cycles) and remembers the core on which the event was logged. The core is only implicitly stored, as we have a separate tracing buffer on each core, allowing us to identify the core for an event at a later stage automatically, without storing it for each event.

As timestamps are stored as a 64 bit number, we need a total of 128 bits (respectively 16 bytes) per event that has been logged. The data structure layout of a single event can be seen in Figure 3.2.

[1]Currently there exist 16 different Subsystems.

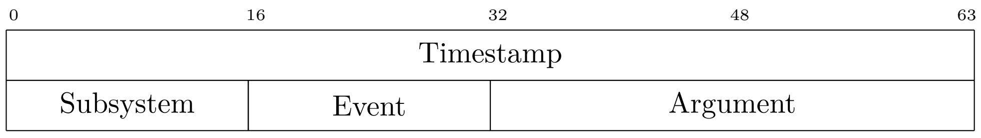


Figure 3.2: Representation of a single trace event in memory in Barrelfish.

3.3 Pleco: A new Domain Specific Language

3.3.1 Overview

As trace events are identified by the type of their Subsystem and Event (which is a two tier hierarchical structure), the best way to specify those Subsystems and Events is using a domain specific language. For this purpose we designed a new domain specific language called *pleco*, that resembles the domain specific language for error codes in Barrelfish (called fugu) a lot – due to the fact that it solves a very similar task.

Pleco allows programmer to easily add new Subsystems to the tracing framework and to extend existing Subsystems with new Events. Note that the Argument parameter of the trace events is not specified in Pleco, as this parameter is intended to be a payload, and not to be a means to distinguish different trace events. A small sample pleco file can be seen in Listing 3.1. In this file we define two Subsystems: `kernel` and `memserv`. Note that the keyword *subsystem* is used to define a new Subsystem. The Events for a Subsystem are defined in the block following its name. Events have both a name and a verbose description, following the keyword *event*. The textual description is *not used* in the tracing framework inside of Barrelfish, but Aquarium will use the textual description to display it when analyzing generated traces. Note that the textual description is not a strict requirement; if the empty string is provided, during the interpretation of the pleco file, the name of the event will be substituted for the textual description.

```
subsystem kernel {

        event CSWITCH                  "Context Switch",
        event BZERO                    "Buffer zeroing",
        event TIMER                    "",
        event TIMER_SYNC               "",

};

subsystem memserv {

        event ALLOC                     "",
};
```

Listing 3.1: A small example pleco file with two Subsystems.

3.3.2 Interpreting Pleco Files

Parsing and interpreting of pleco files is part of the Barrelfish build process, meaning that the according tools are written in Haskell and are integrated into the Hake build process. An overview of how pleco files are integrated into the Barrelfish toolchain can be seen in Figure 3.3. Note that the header file that is created during the build process is directly used in the very same build process, i.e. it is just an intermediate file.

3.3.3 The Generated Header File

For the pleco file of Listing 3.1, the header file shown in Listing 3.2 has been generated during the build process. In Barrelfish source code, this file can be included with the statement:

```
#include <trace_definitions/trace_defs.h>
```

Note that the macro that are created for events also contain the subsystem name, so that there will not be any name collisions when two different subsystem define an Event with the same name.

The generated numbers are not randomized. The reason for this is not that people can avoid using macros, but rather for a new feature that has been introduced into the tracing framework to work: enabling and disabling of Subsystems that are logged. See Section 3.4.2 for detailed information.

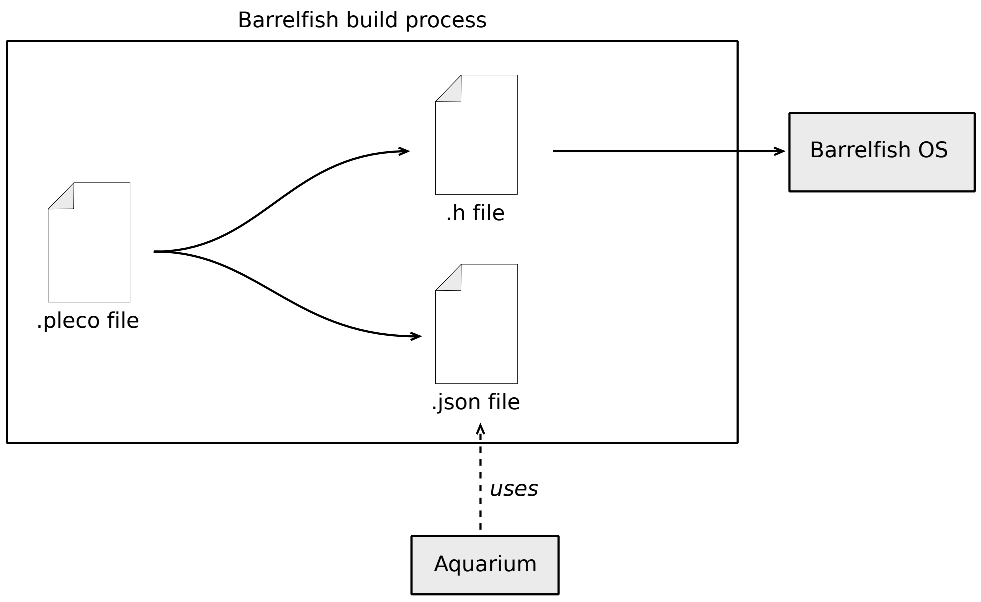


Figure 3.3: Pleco files get translated into both a C header file and a JSON file. This translation is taking place during the regular build process of Barrelfish.

```
#ifndef TRACE_DEFS_BARRELFISH__
#define TRACE_DEFS_BARRELFISH__

#define TRACE_SUBSYS_KERNEL 0
#define TRACE_EVENT_KERNEL_CSWITCH    0
#define TRACE_EVENT_KERNEL_BZERO      1
#define TRACE_EVENT_KERNEL_TIMER      2
#define TRACE_EVENT_KERNEL_TIMER_SYNC     3

#define TRACE_SUBSYS_MEMSERV      1
#define TRACE_EVENT_MEMSERV_ALLOC     0


#define TRACE_NUM_SUBSYSTEMS      2

#endif // TRACE_DEFS_BARRELFISH__
```

Listing 3.2: A header file that has been generated based on the pleco file shown in Listing 3.1.

3.3.4 The Generated JSON File

As the pleco file shown in Listing 3.1 does not only get translated into a header file, but also into a JSON file, we want to have a look at this file now. The JSON file that has been generated for said pleco file can be seen in Listing 3.3.

```
{
0 : {
    "name" : "kernel",
    "events" : {
        0 : "Context Switch",
        1 : "Buffer zeroing",
        2 : "TIMER",
        3 : "TIMER_SYNC"
    }
},
1 : {
    "name" : "memserv",
    "events" : {
        0 : "ALLOC"
    }
}
}
```

Listing 3.3: A JSON file that has been generated based on the pleco file shown in Listing 3.1. This file can be used by Aquarium to decode log traces.

As you can see, the textual description in the pleco file was used where provided, and where it wasn't, the name of the Event has been used as a substitution. The generated numbers are the same as the ones in the header file. This is no coincidence, as the usage of this JSON file is exactly to decode the numbers from the trace logs into human readable events again.

For the purpose of decoding the events, the old version of Aquarium had the mapping from numbers to human readable events directly hard coded into the source code. This new way of defining Subsystems and Events in pleco files allows programmers to omit duplicate work (and having to check that both programs are always consistent), and provides them with an automated way of having a consistent tracing framework and analysis tool.

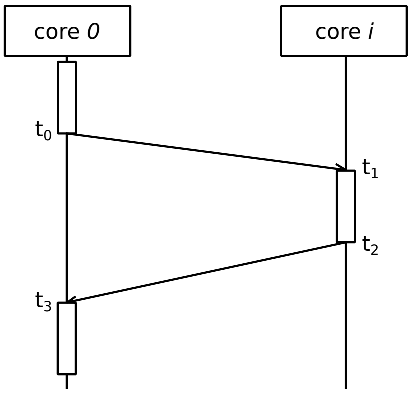


Figure 3.4: NTP clock synchronization. Four time measurements t_0 to t_3 are performed.

3.4 Feature Overview

3.4.1 Preparing the Tracing Framework

The tracing framework does not strictly need any extra preparation, nevertheless depending on the environment, a preparation might be necessary. For this reason we added the functionality to prepare the tracing framework. Currently the preparation process estimates the offset between the CPU cycle counters on the different cores. This functionality is not needed on machines that have synchronized cycle counters, but in the future it might be possible to run a single instance of Barrelfish on multiple machines, and in this case the different cycle counters will not be synchronized anymore.

The cycle counter offsets are all measured relative to core 0. To measure the offset between a core i and core 0, we execute the Network Time Protocol clock synchronization algorithm between the two cores. Figure 3.4 illustrates the steps of the clock synchronization between two cores. Four time measurements are performed and the estimated offset θ between the two cores is calculated as follows:

$$\theta = \frac{(t_1 - t_0) + (t_2 - t_3)}{2} \tag{3.1}$$

The tracing framework performs measurements between every core i ($i > 0$) and core 0 sequentially, so that the measurements are as precise as possible. The messages needed to perform those measurements are sent using the monitor, meaning that the tracing framework does not need to setup any new channel.

3.4.2 Enabling and Disabling of Events

With the new version of Aquarium it is possible to filter out events in the analysis for a given trace log. But it showed that this functionality is not sufficient, as there are use cases where applications log so many events, that filtering must already be performed on the fly, i.e. already during the tracing process itself. An application where this is currently necessary in Barrelfish is the tracing of the network stack. The current way of achieving this filtering is introducing preprocessor statements at different locations in the code. Having the new domain specific language available, we implemented a mechanism to enable and disable Subsystems directly at runtime, using the Subsystem identifier generated from the pleco file.

It is now possible to change which Subsystems are logged directly at runtime, removing the need of recompiling the entire tracing framework just because the type of events that a developer is interested changed. With the hierarchical structure of Subsystems and Events it was possible to implement this enabling facility in a lightweight manner, as the number of Subsystems is quite small.

3.4.3 Automatic Flushing

The flush procedure described in Section 3.1 can be triggered by manually calling the according trace framework function. In addition to the manually triggered flushing, we added a new functionality, namely the one that the trace buffer is flushed automatically. This functionality is implemented with in Bfscope, as we think the main use-case for automatic flushing is when the generated logs are automatically forwarded to a remote machine[2]. When a developer decides to enable the automatic flushing and Bfscope is running, Bfscope will automatically flush the content of the trace buffers periodically. This feature removes the need of having to call the flush procedure manually, but it developers should note that if timing is critical for your application, the automatic flushing functionality can lead to issues. The issues that can arise come from the fact that it is possible that Bfscope flushes in the middle of your application executing its code – this does not lead to a problem of correctness, but it can heavily skew the flow of events in the Barrelfish as a whole.

[2]Having the console cluttered with events from the tracing framework can render the application unusable rather quickly.

3.5 Bfscope

Bfscope is a Barrelfish program that enhances the functionality of the tracing framework by the possibility to directly flush trace logs over the network. Note that the tracing in the Barrelfish code itself runs independently of Bfscope – and it even notices when Bfscope is running and changes its behavior accordingly. Bfscope allows developers to connect from a remote machine to the Barrelfish OS, using a TCP connection and to get the trace logs directly onto the remote machine. Note that when a remote machine is connected, regular flush commands in Barrelfish will automatically be redirected onto the network, and you will not see the trace logs on the console any longer.

As the remote machine is merely a utility that wants to get the trace log data, there are no messages exchanged as part of a protocol – Bfscope simply sends the trace log data onto the TCP connection, once the flush command is issued (or periodically if automatic flushing is enabled). This has, beneath being a simple protocol, the additional benefit that it is no longer necessary to run Aquarium in order to be able to get the trace log onto a remote machine, but you rather can use any tool that allows you to open a TCP connection, such as netcat. Using such a tool will allow you to get the trace log data on a different machine, where you can either later analyze it with Aquarium, or with custom scripts.

Nevertheless the main intention is to directly connect to Bfscope using Aquarium, which can interpret and visualize the trace log data directly on the fly.

Chapter 4

Design and Implementation of the Analysis Tool Aquarium

4.1 Design of Aquarium

4.1.1 Goals

When we designed Aquarium we had several goals in mind, namely the following ones:

1. Support for live tracing.
2. Support for different ways of input (e.g. reading from file or receiving data over the network).
3. Being able to handle large trace log data.
4. Being extensible and easily customizable.
5. Aquarium must run on different operating systems.

We decided to tackle the first three goals with the design of the architecture of Aquarium, which we will discuss in Section 4.1.2. The fourth goal also did influence the architecture on one hand, but also led to the idea of making Aquarium scriptable, i.e. to create an interface that allows developers to add their own scripts to Aquarium. Since those scripts do not work on the raw trace log data, but rather on already from Aquarium interpreted data, it offers developers on one hand a more powerful means to write scripts in a very easy way, and on the other hand the scripts are directly integrated into the visualization of Aquarium, alleviating the need to write visualization code for custom developer scripts. In Section 4.2 we will discuss the different ways how Aquarium can be extended with scripts.

The fifth goal, i.e. the goal that people should be able to run Aquarium on different Operating Systems, such as Linux and Windows, arose from a shortcoming in the old version of Aquarium – namely that it was written in C# and only runs on Windows. To tackle this requirement we decided to implement our version of Aquarium in Java, so that cross platform portability will certainly not become an issue.

4.1.2 Architecture

When you analyze trace log data with Aquarium, the main object is a TracingSession object. Each trace log data is at runtime represented by exactly one TracingSession object. Figure 4.1 shows the most important classes that are dealing with getting from trace log data to the according TracingSession. A TracingSession is associated with a single event provider, currently there are two different input ways implemented:

- Reading trace log data from a file, using a LogfileReader.
- Reading trace log data directly from a Barrelfish machine, using a NetworkReader.

The actual interpretation of the trace log data is done using an EventParser; EventParser objects are independent of the type of data source. Note that the EventConfigurtion is the responsible for interpreting the JSON file that has been generated during the build process of Barrelfish, based on the pleco file.

The trace log data gets interpreted to Events and Activities, that are stored in the TracingSession object. Note that the flow of data is push based, i.e. it is the data source that actively creates new Events as soon as more data is available, and pushes the Events to the TracingSession. Having an active data source allows us to treat different types of data sources uniformly.

While Events are quite self explanatory, i.e. they are the Aquarium representation of the actual events in the trace log data, Activities are a new concept that we introduced in Aquarium. An Activity is a sequence of Events, that are grouped into a single activity. Activities are a typical constructs that are needed when analyzing trace log data; an example for that is when analyzing the network stack, the fact that memory has been allocated (a single event) might not be very interesting, but the duration of the entire construction of the packet (an Activity) is what is actually very interesting. Thinking about Activities, it becomes immediately clear that the different types of Activities must be flexibly definable. We achieved this by allowing developers to create their own scripts that decode Activities.

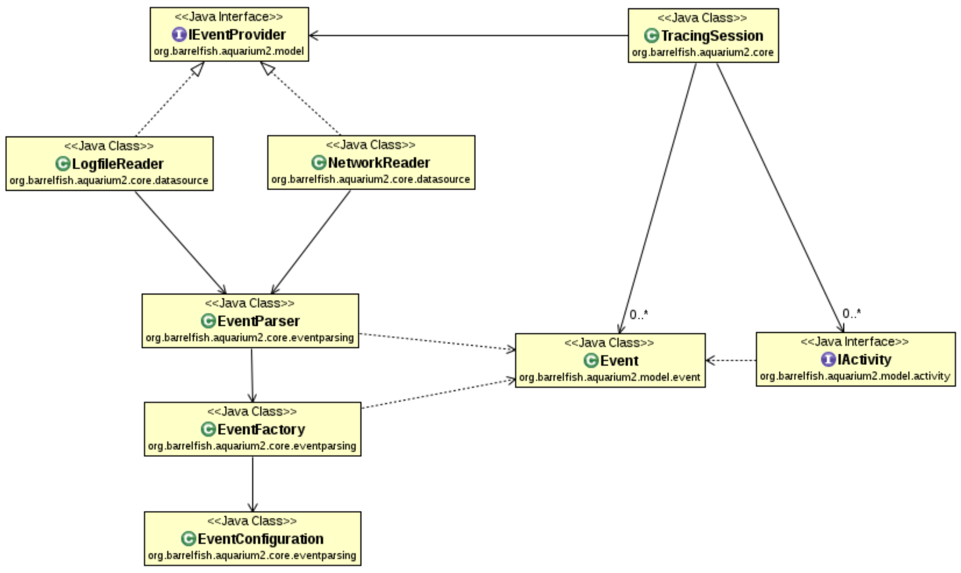
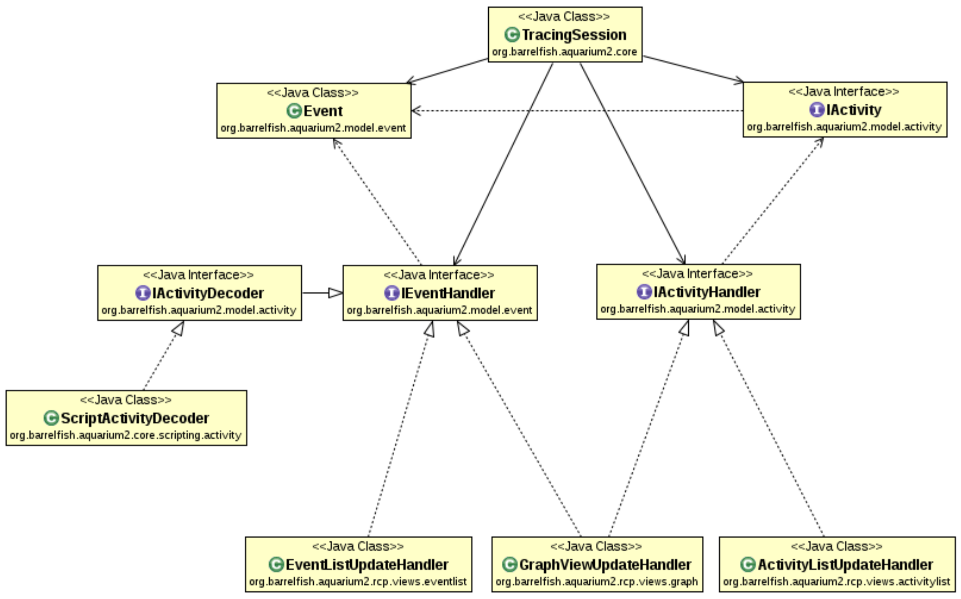

<<Java Interface>>
IEventProvider
org.barrelfish.aquarium2.model
<<Java Class>>
TracingSession
org.barrelfish.aquarium2.core
<<Java Class>>
LogfileReader
org.barrelfish.aquarium2.core.datasource
<<Java Class>>
NetworkReader
org.barrelfish.aquarium2.core.datasource
<<Java Class>>
EventParser
org.barrelfish.aquarium2.core.eventparsing
0..*
0..*
<<Java Class>>
Event
org.barrelfish.aquarium2.model.event
<<Java Interface>>
IActivity
org.barrelfish.aquarium2.model.activity
<<Java Class>>
EventFactory
org.barrelfish.aquarium2.core.eventparsing
<<Java Class>>
EventConfiguration
org.barrelfish.aquarium2.core.eventparsing

Figure 4.1: UML class diagram showing the main classes that are concerned with dealing with input.

Let us now look at how events are processed once the TracingSession retrieves a new Event. A class diagram illustrating the handling of Events and Activities can be seen in Figure 4.2. A TracingSession stores both a list with the Events that have been extracted from the trace log data, as a list with all the activities that have been created based on those events. Once an Event is received by the TracingSession, it notifies all registered EventHandlers to handle the new Event. Such EventHandlers can either be UI elements, such as an EventListUpdateHandler (an object being responsible to present a list of all interpreted Events in the UI), or an ActivityDecoder. ActivityDecoders are objects that create Activities, and one possibility for that is, as already mentioned, to have external scripts which decode Activities. If an ActivityDecoder creates a new Activity, this Activity will be added to the TracingSession and all registered ActivityHandlers will receive it. As you can see, the only module that is currently both receiving Events and Activities is the GraphViewUpdateHandler, an object that is responsible for visualizing the trace log data graphically.

When developing Aquarium, we initially planned to add a statistics module as well. Due to a lack of time, we had to omit it in the end. Nevertheless from the design it can be seen, that such a module could easily be added to Aquarium: It would simply have to be an EventHandler and an ActivityHandler. Note that the design with having the TracingSession at the core, we achieved that all handler classes are always in a consistent state. For example if an activity is seen by one handler, it is always also seen by all other handlers. This becomes especially handy when considering the filtering functionality of Aquarium. In Aquarium we added the functionality to filter out Events based on various different criteria, ranging from the core on which the Event happened, over the Subsystem type up to custom scripts that developers can write to create their own filter. When a filter is applied, it is always applied on the TracingSession, and not on e.g. a UI element. With this globally applied filtering mechanism a new Handler that is created to extend Aquarium would immediately benefit from the filtering functionality, without having to take care of it at all.



<<Java Class>>
TracingSession
org.barrelfish.aquarium2.core
<<Java Class>>
Event
org.barrelfish.aquarium2.model.event
<<Java Interface>>
IActivity
org.barrelfish.aquarium2.model.activity
<<Java Interface>>
IActivityDecoder
org.barrelfish.aquarium2.model.activity
<<Java Interface>>
IEventHandler
org.barrelfish.aquarium2.model.event
<<Java Interface>>
IActivityHandler
org.barrelfish.aquarium2.model.activity
<<Java Class>>
ScriptActivityDecoder
org.barrelfish.aquarium2.core.scripting.activity
<<Java Class>>
EventListUpdateHandler
org.barrelfish.aquarium2.rcp.views.eventlist
<<Java Class>>
GraphViewUpdateHandler
org.barrelfish.aquarium2.rcp.views.graph
<<Java Class>>
ActivityListUpdateHandler
org.barrelfish.aquarium2.rcp.views.activitylist

Figure 4.2: UML class diagram showing the main classes that are concerned with handling events and activities.

4.2 Extending Aquarium with Scripts

As mentioned in Section 4.1.2, it is possible to extend the functionality of Aquarium by adding custom scripts. The scripts are interpreted using the Java Scripting API, and currently JavaScript is the language for which support in Aquarium has been implemented. Based on the Java Scripting API support for other languages could be added.

4.2.1 Script Filters

Script filters are custom scripts that can be written by developers to filter out events in which they are not interested. Aquarium itself already provides the possibility to filter out events based on the following criteria:

- Filter out entire cores (e.g. filter out core 1).
- Filter out entire Subsystems (e.g. filter out the *kernel* Subsystem).
- Filter out Events from a Subsystem (e.g. filter out *ALLOC* Events from the Subsystem *memserv*).
- Filter out trace events based on their application (e.g. filter out all events that the application *monitor* logged).

If a user is not satisfied with these possibilities to filter out events, Aquarium can be extended with script filters. An example for a script filter would be to filter out all events, except those that are an *ALLOC* Event initiated by the monitor. Such scripts allow users to quickly spot specific events, even when they are analyzing large trace logs.

4.2.2 Script Activities

Another possibility to extend Aquarium with the help of scripts is to write custom activity scripts. Such a script works in the following way: It receives all events that exist in the trace log, in the order they exist in the trace log itself, and based on these events it can create activities, and deliver them to Aquarium. In Figure 4.2 we can see that such a Script is wrapped in a ScriptActivityDecoder inside of Aquarium, which is – as just described – an EventHandler.

An example for an activity script could be to create an activity for all the *MUTEX_LOCK* and *MUTEX_UNLOCK* pairs – in order to analyze the locking behaviour. For each activity, certain parameters such as the duration of each activity, is automatically calculated by Aquarium.

4.3 Working with Aquarium

In this section we briefly want to look at how some of the already described functionality looks in Aquarium with the help of some examples. Figure 4.3 shows a screenshot displaying a single trace log data file opened in Aquarium. The largest part of the GUI is used by the so called *GraphView*, presenting the information contained in the log in a two dimensional manner. From left to right we see the timestamps (measured in clock cycles), and on the vertical axis we see the different cores.

For each core we show the actual events that have been traced, indicated using black circles on the bar of the core. The color of the bar shows which application was running on the core on that time, where the colors are shown as well in the left menu labeled *Filter*. In addition to the per core events, arrows are drawn where messages have been sent between the cores, indicating the send and receive event. As the messages have the potential to clutter up the GUI quite a bit, the arrows can be hidden easily using the envelope button on the right top corner.

Below the GraphView we see a list representation of the event data. With the help of the *sync* button (shown on the right top corner of the list), the GraphView and the list can be linked, meaning that if you select an event in either of the two, the other view scrolls to that event. Using this functionality coarse navigation can be done using the GraphView, to then allow for detailed analysis by quickly looking at the list.

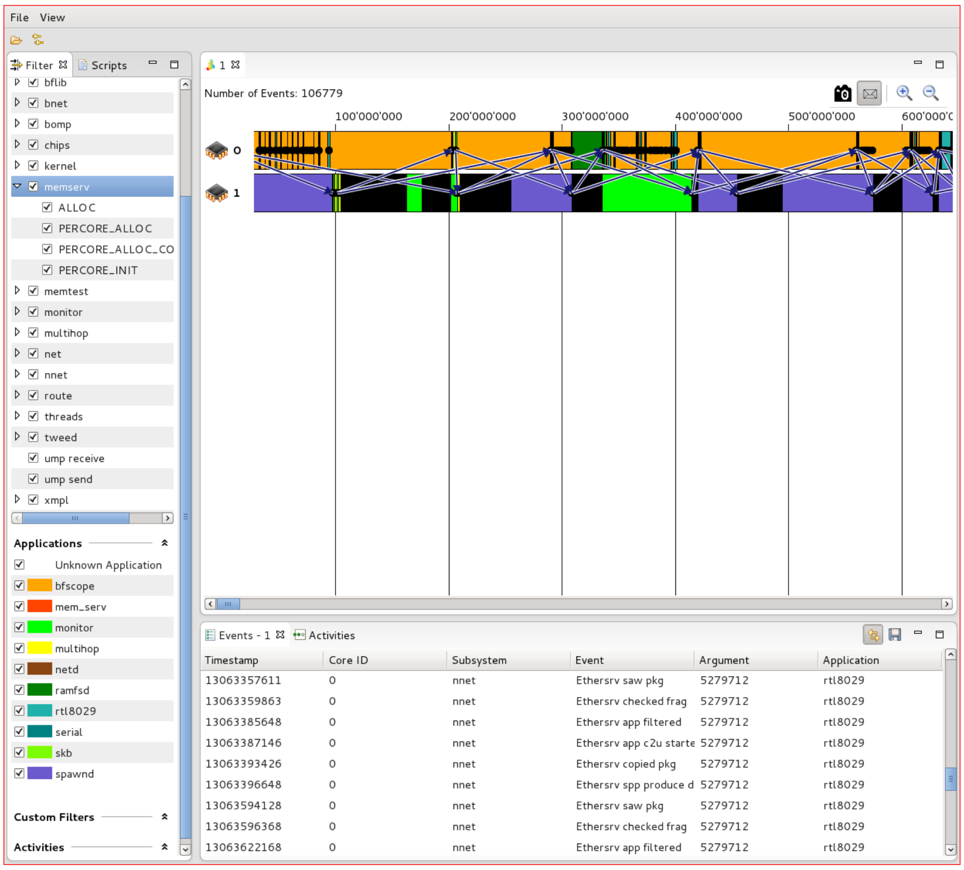
On the left part of the GUI we see the Filter menu. It allows to filter out events based on the different criteria, as already described. Scripts can be added using the Scripts tab, and afterwards they will directly appear in the Filter menu as well.

As we can see in the screenshot shown in Figure 4.4, for all the objects in the GraphView exist tooltips, when you hover over the according object with the mouse cursor. On this screenshot you can see that two custom script activities have been added, and they have already been evaluated. The created activities are integrated into the GraphView (on a per core basis) as well as in the Activity tab next to the Events list on the bottom of a Aquarium. All created activities can also be seen in a list fashion there.

In Figure 4.5 we activated several filters, thus compared to what we saw in Figure 4.4, the information displayed in Aquarium has been reduced. We filtered out several things:

- The entire core 0.
- The Event MUTEX_UNLOCK.
- The Subsystems for sending and receiving messages, hence the message arrows are filtered as a consequence as well.
- Events belonging to the application spawnd.

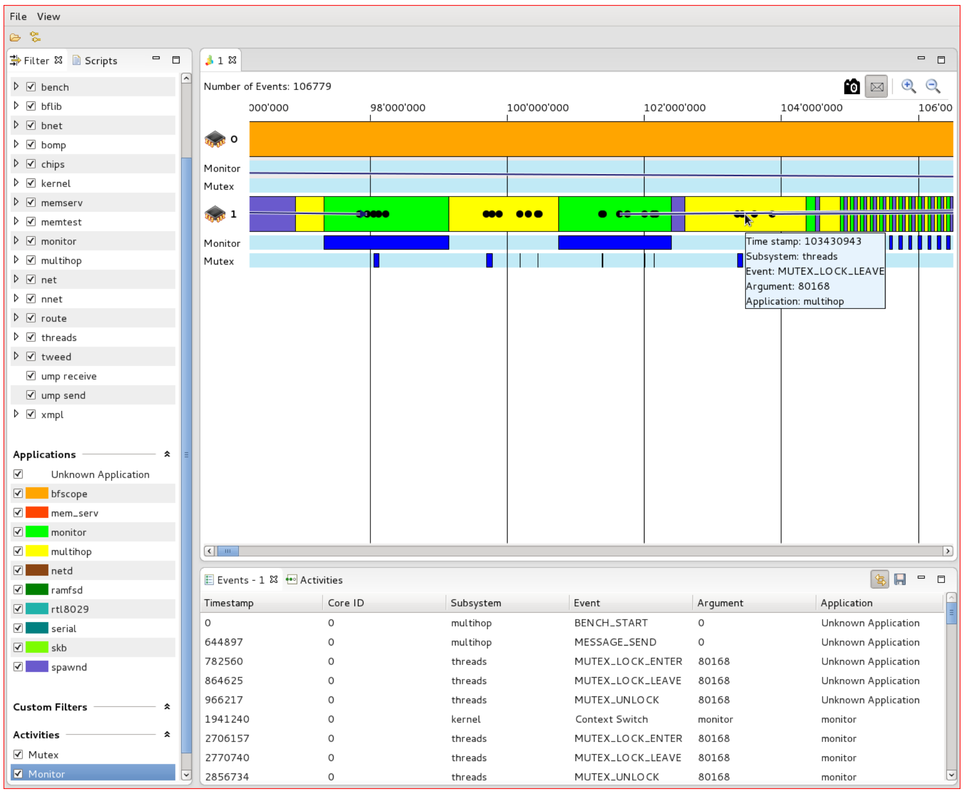
You can see that using the filter mechanism, it is possible to quickly find the part of the trace log data, that is interesting to you.



Number of Events: 106779

Timestamp	Core ID	Subsystem	Event	Argument	Application
13063357611	0	nnet	Ethersrv saw pkg	5279712	rtl8029
13063359863	0	nnet	Ethersrv checked frag	5279712	rtl8029
13063385648	0	nnet	Ethersrv app filtered	5279712	rtl8029
13063387146	0	nnet	Ethersrv app c2u starte	5279712	rtl8029
13063393426	0	nnet	Ethersrv copied pkg	5279712	rtl8029
13063396648	0	nnet	Ethersrv spp produce d	5279712	rtl8029
13063594128	0	nnet	Ethersrv saw pkg	5279712	rtl8029
13063596368	0	nnet	Ethersrv checked frag	5279712	rtl8029
13063622168	0	nnet	Ethersrv app filtered	5279712	rtl8029

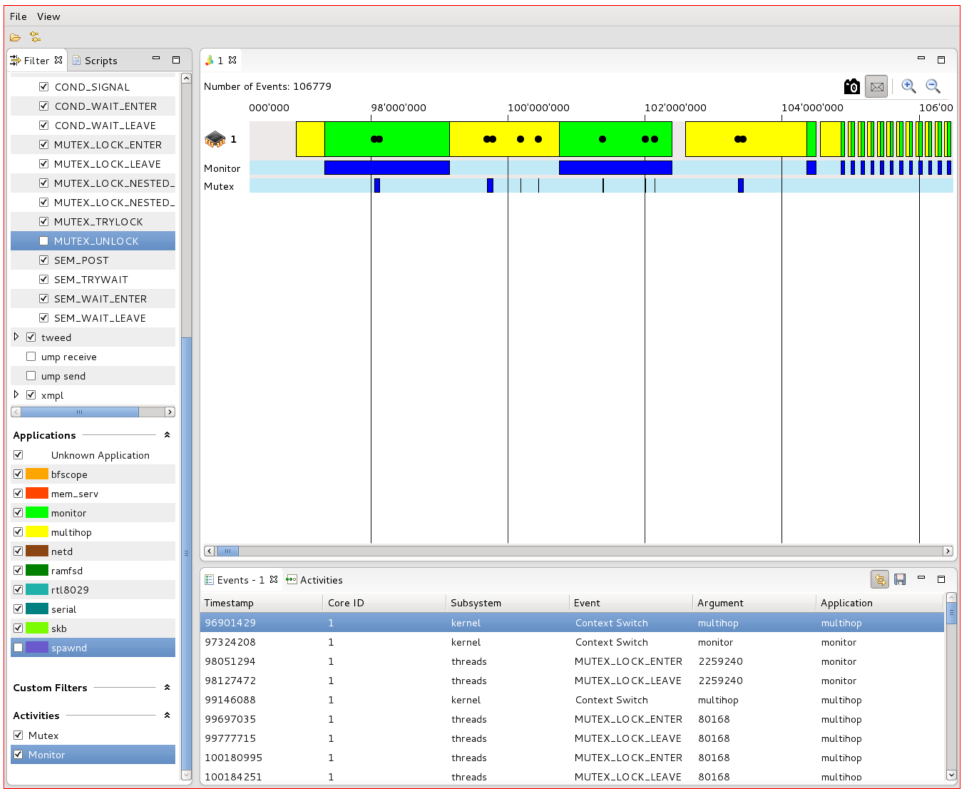
Figure 4.3: Screenshot of Aquarium displaying one trace log file.



Time stamp: 103430943
Subsystem: threads
Event: MUTEX_LOCK_LEAVE
Argument: 80168
Application: multihop

Timestamp	Core ID	Subsystem	Event	Argument	Application
0	0	multihop	BENCH_START	0	Unknown Application
644897	0	multihop	MESSAGE_SEND	0	Unknown Application
782560	0	threads	MUTEX_LOCK_ENTER	80168	Unknown Application
864625	0	threads	MUTEX_LOCK_LEAVE	80168	Unknown Application
966217	0	threads	MUTEX_UNLOCK	80168	Unknown Application
1941240	0	kernel	Context Switch	monitor	monitor
2706157	0	threads	MUTEX_LOCK_ENTER	80168	monitor
2770740	0	threads	MUTEX_LOCK_LEAVE	80168	monitor
2856734	0	threads	MUTEX_UNLOCK	80168	monitor

Figure 4.4: Screenshot of Aquarium displaying two script activities, one for mutex activities and one for the monitor application.



- ✓ COND_SIGNAL
- ✓ COND_WAIT_ENTER
- ✓ COND_WAIT_LEAVE
- ✓ MUTEX_LOCK_ENTER
- ✓ MUTEX_LOCK_LEAVE
- ✓ MUTEX_LOCK_NESTED_
- ✓ MUTEX_LOCK_NESTED_
- ✓ MUTEX_TRYLOCK
- ☐ MUTEX_UNLOCK
- ✓ SEM_POST
- ✓ SEM_TRYWAIT
- ✓ SEM_WAIT_ENTER
- ✓ SEM_WAIT_LEAVE
- ✓ tweed
- ☐ ump receive
- ☐ ump send
- ✓ xmpl

Applications

- ✓ Unknown Application
- ✓ bfscope
- ✓ mem_serv
- ✓ monitor
- ✓ multihop
- ✓ netd
- ✓ ramfsd
- ✓ rtl8029
- ✓ serial
- ✓ skb
- ☐ spawnd

Custom Filters

Activities

- ✓ Mutex
- ✓ Monitor

Timestamp	Core ID	Subsystem	Event	Argument	Application
96901429	1	kernel	Context Switch	multihop	multihop
97324208	1	kernel	Context Switch	monitor	monitor
98051294	1	threads	MUTEX_LOCK_ENTER	2259240	monitor
98127472	1	threads	MUTEX_LOCK_LEAVE	2259240	monitor
99146088	1	kernel	Context Switch	multihop	multihop
99697035	1	threads	MUTEX_LOCK_ENTER	80168	multihop
99777715	1	threads	MUTEX_LOCK_LEAVE	80168	multihop
100180995	1	threads	MUTEX_LOCK_ENTER	80168	multihop
100184251	1	threads	MUTEX_LOCK_LEAVE	80168	multihop

Figure 4.5: Screenshot of Aquarium where core 0 is filtered out, as well as certain events and applications.

Chapter 5

Performance Analysis

5.1 Introduction

This chapter analyzes the performance of the tracing framework only inside of Barrelfish. The analysis tool Aquarium is not analyzed for its performance, as it is intended to run *"offline"* in the sense that if it is fast enough the consume the data on live mode from a Barrelfish machine, it is considered to be fast enough. The impact on performance is also a lot bigger on the tracing inside of Barrelfish; This stems from the fact that if you do not want to analyze trace data, you simply do not start Aquarium, and if you want to analyze data, you are willing to wait until the analysis is performed. Looking at the tracing framework in Barrelfish, it is on one hand less easy to disable – once compiled into the system, a certain overhead will exist – and on the other hand it is important that the tracing does not affect measured code too heavily, or it will become useless.

5.2 Memory Overhead

The memory overhead for buffers inside the tracing framework is constant during the entire runtime of Barrelfish, as the only used buffers are allocated at startup of the system. The used buffer space currently consists of two main parts that exist for each core:

Application Buffer Up to 128 currently running applications can be stored per core.

Event Buffer Up to 8000 Events can be stored per core, where ringbuffer containing these events is cleared during a flush process.

To store an event or an application 16 bytes are used. As the tracing framework works independently of the actual number of cores, the number of cores is bounded assuming a limit of 64 cores. This leads to the following memory usage:

$$M = (128 + 8000) * 16\text{B} * 64 = 8323072 \approx 8\text{MB} \tag{5.1}$$

In addition to those buffers, a handful of pointers are stored, which in total use less than 1 KB of memory. Therefore that the total amount of memory that the tracing framework uses is 8 MB, which does not vary over time.

5.3 Execution Time Overhead

5.3.1 Cost to Trace a Single Event

We benchmarked the number of cycles that it takes to trace a single event in the tracing framework. We tested both the case where the Subsystem is enabled, i.e. we are interested in the event for which the `trace_event` function is called, and the case where we are not interested in the even that is traced, i.e. the Subsystem is disabled. The *"enabled"* case is straightforward to benchmark, but we also think the *"disabled"* case is interesting, as it might be often the case that code is instrumented with a lot `trace_event` calls, even though you are currently not interesting in analyzing this part of the code. Since we added the functionality do dynamically disable the appropriate Subsytems, it is also important to know by what degree the execution of the code will be slower, compared to removing the statement from the actual code.

The results of the benchmark can be seen in Figure 5.1. The benchmark shows that on the machine `nos5`, the average number of cycles that it takes to trace an event, when the Subsystem is enabled is 40.384. The average number of cycles for a call, when the Subsystem is disabled is 9.950.

It can be seen that both benchmarks returned very stable results - there are a few outliers, but the vast majority of the events are closely around the average. Both benchmarks have been run twice with 1000 measurements each run.

5.3.2 Cost to Flush

The cost to flush the collected trace data can vary a lot depending on the destination onto which is flushed: Directly on the console, using Bfscope to send it over the network, etc. As the tracing framework is not intended to be used in a way where flushing is performed during a measurement, but afterwards, we did not do measurements for the different flushing methods. We only want to mention that flushing, especially over the network, is not to be considered a lightweight operation that can be done at any time during your code, without potentially affecting the outcome of the tracing heavily.

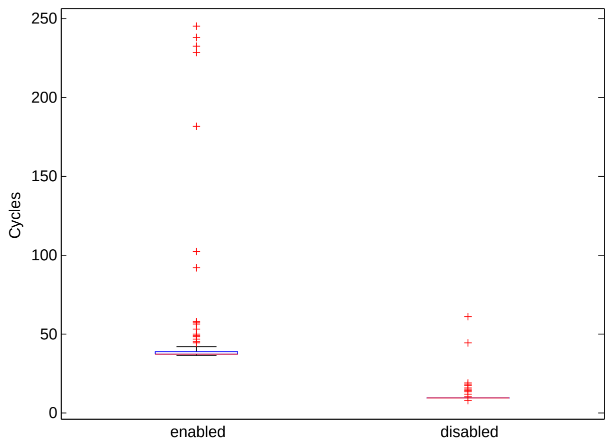


Figure 5.1: Boxplots showing the number of cycles that it takes to trace a single event. On the left: The Subsystem is enabled, i.e. the event is stored in the buffer. On the right: The Subsystem is disabled, i.e. the event is not stored in the buffer.

Chapter 6

Conclusion

We improved the tracing infrastructure for Barrelfish. The changes we made to the Barrelfish source code improve maintainability and usefulness of the tracing infrastructure.

We think that the new Aquarium tool is very useful for development and debugging because it visualizes events occurring in Barrelfish and therefore allows the developer to understand performance, overhead or timing-related bugs. It is far superior to similar tools available for other platforms, such as KernelShark for Linux. It is very extensible, e.g. it allows to import user scripts to define custom filter or define a custom grouping of events that belong to a particular task.

6.1 Future Work

Different flushing policies Currently we have two flushing policies in Bfscope: Entirely manual and totally automatic, meaning that Bfscope flushes each time it gets scheduled. Different flushing policies, such as flushing when the buffers are full to a certain percentage could be thinkable.

Statistics Module We originally planned to implement a statistics module directly in Aquarium instead of only having a data export functionality. Such a module could be implemented rather easily based on the extensible design of Aquarium. Adding script support to such a module could allow for easy customization.

Different Scripting Languages The scripting framework could be adapted to support not only JavaScript but also different script languages.

Semantic Event Analysis Aquarium could try to analyze trace log data instead of only visualizing it, trying to present suggestions about possible anomalies and strange patterns in sequences of events that have occurred.

Bibliography

[1] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on OS Principles*, Big Sky, MT, USA, October 2009.

[2] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. why isn't your os? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verità, Switzerland, May 2009.

[3] Mathieu Desnoyers. Using the linux kernel tracepoints. `http://www.mjmwired.net/kernel/Documentation/trace/tracepoints.txt`, October 2012.

[4] Paul Fox. Crisp, dtrace, and other technobabble. `http://crtags.blogspot.ch/2012/04/dtrace-ftrace-ltrace-strace-so-many-to.html`, April 2012.

[5] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler, 1982.

[6] Alexander Grest. A routing and forwarding subsystem for a multicore operating system. Bachelor thesis, Swiss Federal Institute of Technology (ETH), August 2011.

[7] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the ACM EuroSys Conference (EuroSys 2010)*, Paris, France, April 2010.

[8] Steven Rostedt. Debugging the kernel using ftrace - part 1. `http://lwn.net/Articles/365835`, December 2009.

[9] Steven Rostedt. Kernelshark. `http://people.redhat.com/srostedt/kernelshark/HTML`, May 2010.

[10] Steven Rostedt. trace-cmd: A front-end for ftrace. `https://lwn.net/Articles/410200`, October 2010.

[11] Steven Rostedt. Using kernelshark to analyze the real-time scheduler. `http://lwn.net/Articles/425583`, February 2011.