Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Arbeitsgruppe Parallelität und Programmiersprachen
Professor Dr. Rita Loogen

Bachelorarbeit

Analysis of Large Eden Trace Files

Bastian Reitemeier

Bremen, 28. September 2015

Table of Contents

List of Figures

List of Tables

Listings

1 Introduction

Trace analysis is a useful tool for debugging and optimizing Haskell programs, especially when dealing with parallelism. Several tools exist to parse and analyze eventlog files generated by the Glasgow Haskell Compiler (GHC) runtime system [2],[3],[4],[5],[1]. While offering features similar to the popular ThreadScope, which does not handle Eden traces, the Eden Trace Viewer (EdenTV) offers additional features specifically designed to analyze, benchmark and troubleshoot programs written in Eden [2],[3]. This includes the analysis of Eden traces not only on the level of individual threads, but includes additional views to analyze traces on the machine and process level. It also offers the ability to display Eden-specific inter-process communication via messages, so that data flow between machines and processes can be analyzed and enhanced.

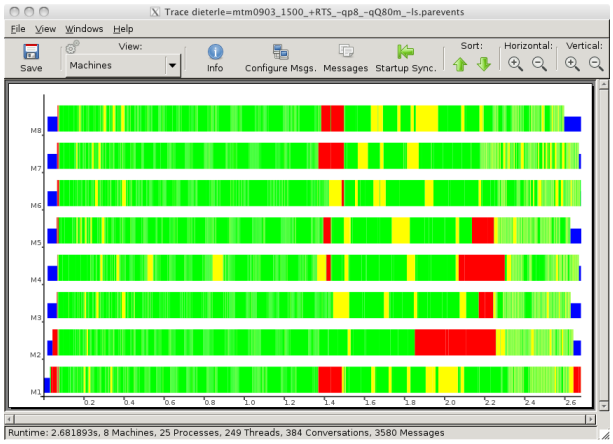


Figure 1: Screenshot of EdenTV, displaying the machine view of a trace.

However, there are some areas in which the EdenTV could still be improved upon. The first of which is the limitation of the file size: Traces generated by the GHC runtime system tend to be rather large in size - in the order of several hundred MB to several GB per minute runtime. Additionally, parallel programs created with Eden generate multiple traces, because every core (or machine) has its own runtime system. Yet, EdenTV is only capable of displaying traces which fit into the RAM of the computer it is running on. In order to be analyzable programs have to be limited to a short running time - in the order of several seconds to a few minutes, depending on the number of traces generated and the amount of RAM available to the researcher - to be displayed satisfactorily. Secondly, EdenTV renders trace visualizations to a single image buffer with fixed resolution. As a result, events which have a duration that translates into an image width of less than a single pixel cannot be displayed properly. Prior experience in using EdenTV suggests that the addition of these features would improve the research process.
Lastly, EdenTV is an organically grown piece of research software, which has been edited and improved upon by several authors over different periods of time. It is thus a powerful tool for doing analysis on Eden traces, but it also contains many different code styles and paradigms which hinders readability and extensibility. This thesis aims at providing a tracing tool with a deliberately designed and well-documented interface, so that additional analysis tool sets can be added in the future.
Consequently, a new implementation of a trace viewer should include the following features:

a) the ability to open files of arbitrary size, without being constrained by the available memory

b) the ability to analyze traces at all levels of temporal resolution, i.e. the ability to zoom arbitrarily deep into a trace

c) A well-documented and extensible interface for extending and improving trace analysis

This thesis will document implementation of such a program, which from here on will be referred to as Eden-Tracelab.
Eden-Tracelab is split into three major parts: The parser and event processor, the web service and the HTML5 viewer. This thesis will be structured along those parts. First, the explanation will go into some of the technical background in chapter 3, introducing the GHC and Eden runtime, and the structure of GHC trace files. The source code for the parser will be explained simultaneously, as it fits with the explanation of the of the file format nicely. In chapter 4 the overall structure of Eden-Tracelab is going to be discussed, starting with the general architecture of Eden-Tracelab. Then, every part of Eden-Tracelab will be explained in detail, starting with the remaining functionality of the event processor, namely the processing of the parsed events to higher order events and the transfer of those events into the database. The rest of the chapter will cover the functionality and implementation of the web service, as well as the functionality and implementation of the HTML5 viewer. Chapter 5 contains a comparison between Eden-Tracelab and EdenTV using an example trace.

2 Technical Background

2.1 Eden

Eden is an extension to the Haskell programming language. It is designed to offer the programmer the ability to define and instantiate parallel processes [6]. To understand how this is accomplished, it is beneficial to take a look at the main functions and data types provided by Eden. The `process` function takes an ordinary function, and returns a `Process`.

```
process :: (Trans a, Trans b) => a -> b -> Process a b
```

Listing 1: Defining Eden processes

The `Trans` type class is derived from the `NFData` type class – values being communicated between processes in Eden are evaluated to normal form prior to sending. To instantiate a process, the `(#)` function is used.

```
(#) :: (Trans a, Trans b) => Process a b -> a -> b
```

Listing 2: Instanciating Eden processes

To understand how these two work together, the following parallel mergesort program is provided in the standard reference:

```
mergesort :: (Ord a, Trans a) => [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort xs = sortmerge (process mergesort # xs1)
                         (process mergesort # xs2)
    where (xs1,xs2) = unshuffle xs
```

Listing 3: Parallel mergesort in Eden

While process definition and creation are explicit within Eden, process communication is mostly implicit, and handled transparently to the programmer. This is a very powerful paradigm for the creation of parallel functional programs, but it also explains the need for a research tracing tool: To optimize a parallel functional program in Eden, it can be very useful to inspect the state of the processes during the execution of the program, and to inspect the data flow between processes. This enables a programmer to identify if a process is waiting on a result, and to optimize the program to keep these conditions to a minimum.

Eden programs can not only be run on a multicore machine, but also across multiple distributed machines, in which case the program has to be compiled with support for a network middleware. Currently the PVM and MPI middlewares are supported.

2.2 Eden and the GHC Runtime System

To understand the analysis done on Eden traces in this thesis, it is first necessary to understand the main concepts within the GHC runtime system. This section serves as a glossary for important terms within the runtime. It will also explain how the Eden compiler extends this runtime, and introduces the important concepts in the Eden runtime.

Threads

Eden threads are not Operating System(OS) threads, but lightweight threads managed by the GHC runtime. Within the GHC runtime, threads are represented by Thread State Objects (TSOs), which are garbage collected objects within the runtime [7]. Haskell threads are more lightweight than OS threads. The GHC wiki states that their memory footprint that is about 100x times lighter than an OS thread [7].

Single- and multithreaded runtime

The GHC runtime exists in two different versions: The single- and multithreaded runtime [8]. It should be noted that these names refer to number of OS threads within the runtime, both versions support lightweight GHC threads. The multithreaded runtime is needed to enable SMP (symmetric multiprocessor) parallelism in compiled GHC programs. It can be enabled by the `-rtsopts -threaded` compiler flags and the following execution flags:

```
+RTS -Nx -RTS
```

when compiling the program, where x is the number of cores to run the program on. The x parameter can also be omitted, in which case the runtime will dynamically choose the number of capabilities when executing the program, based on the number of processors of the machine it is executed on [9].

Capability

A capability is an abstraction over a piece of hardware capable of executing Haskell programs. When dealing with multicore processors a capability represents a CPU core. The number of capabilities in the runtime is controlled by the `-N` flag [7].

The Eden runtime

As of now, the Eden runtime is restricted to the single-threaded GHC runtime. In other words, a single instance of the Eden runtime will only manage a single capability. An Eden trace still contains events belonging to an additional capability, numbered -1. This is the system capability – a kind of virtual capability, reserved for events that are not assigned to a specific capability. Eden parallelism is achieved by running multiple instances of the runtime concurrently - either on multiple cores, or spread out across multiple physical machines. An Eden trace therefore contains a number of conventional GHC traces, one for each instance of the runtime.

Processes

Eden adds additional abstractions to the aforementioned concepts, the most important one being processes. A process is created by the `process` and `(#)` functions. Within the runtime, a process is a set of lightweight threads.

Machines

During the execution of an Eden program, a machine corresponds to a single instance of the runtime. Therefore, a machine is at the core of the execution of a Eden program. These concepts present a simple

hierarchy, that is fundamental to the analysis of Eden Traces: A *machine* holds a number of *processes*, which in turn contain a set of *threads*.

States

At each moment each entity in the runtime is said to have a run state: `Idle`, `Runnable`, `Running` or `Blocked`. Threads are the simplest entities with regard to the concept of run state: They can either be `Runnable`, `Running` or `Blocked`. All other entities (processes and machines) define their state recursively through the state of their constituents. The following table is taken from the documentation of the original EdenTV. It shows how the different states correspond to color in the graphical interface [1].

Color	Machine	Process	Thread
Blue	Idle (Total processes = 0)	Idle (Total threads = 0)	n / a
Yellow	System Time (threads runnable)	Runnable (At least one thread)	Runnable
Green	Running (one thread)	Running (one thread)	Running
Red	Blocked (all threads)	Blocked (all threads)	Blocked

Table 1: Thread/ process/ machine state color coding[1]

2.3 Tracing

When a GHC program is compiled with the *-eventlog* flag, it produces a trace file to document the events occurring in the runtime system during the execution of the program [10]. During execution, the runtime has a set of fixed sized buffers to store events in memory – one for each capability, and an extra one containing the events not linked to a capability. These 'unassigned' events include the invocation details, like the version of the runtime and the shell environment variables of the environment the program was executed in. Each time a buffer is full, it will be written to disk. This behavior is reflected in the chronological structure of the `*.eventlog`-files. Inside the file, the events are laid out in blocks, which can be attributed to a single capability. Within a single block, events will appear in chronological order. Blocks belonging to a single capability will also be in chronological order in the file. However, blocks belonging to different capabilities will not be in chronological order within the file. For example, the buffer belonging to the global pseudo-capability will often not fill up during the course of the execution of a program, because after the runtime has been initiated, no further global events occur until right before termination. Therefore the events attributed to the global capability will be appended to the end of the file, even though they are the very first ones occurring during program execution. Blocks belonging to other capabilities may also occur interspersed and may not adhere to chronological order.

This structure dictates some restrictions on any low-memory parser. If a parser only parses the events of

a single capability, it can parse them in chronological order without having to store additional events in memory. Yet, using a single parser to parse the events of all capabilities within the event-stream presents additional challenges: The events within multiple blocks will have to be parsed and stored in memory, before they can be merged in chronological order. There is a simple and elegant solution to this problem, however. By running multiple parser instances simultaneously - one for each capability within the eventlog - it is possible to parse in chronological order without much performance overhead, by keeping a single event in memory for every instance of the parser. Eden-Tracelab then always processes the 'youngest' event, and then parses the next event of that capability. When dealing with Eden `*.eventlog`-files, there is another advantage: Each `*.eventlog`-file contains only two distinct capabilities: The global pseudo-capability, and the capability the single-threaded runtime is executed on. In the eventlog file the two capabilities are assigned the numerical ids -1 (for the global capability) and 0 (for the actual physical capability). Therefore, the sequential parser only needs to keep two events within its internal state.

2.4 Parsing `*.eventlog`-files

This sections documents the binary format that the GHC runtime system uses to serialize events during run-time. As this format is only documented in the *EventLog.h* header file which is part of the GHC runtime system source code this section will give a brief overview of the format. The custom eventlog parser that was implemented in the course of this thesis will be documented as well. As this parser was written in a descriptive way, a side-by-side explanation of the `*.eventlog`-format and the parser implementation provides the best overview of these concepts.

The hackage packages *ghc-events* and *ghc-events-parallel* (and the associated module *GHC.RTS.Events*) already contain a parser for the `*.eventlog`-format. However, during the course of this thesis, a custom parser for this format was implemented. There are several reasons for this decision.
The existing parser is written in a very elegant way, yet it is difficult to modify. Changing the existing parser into one that could be used to parse a single event at a time proved to be an awkward task as it required stripping away many layers of abstraction. Furthermore, this approach seemed to produce hard to read code with lots of boilerplate. Since one of the stated goals of this thesis was for Eden-Tracelab to be easily maintainable, another approach was taken.
It should be noted that most of the parser implementation is a direct port of the existing parser into attoparsec [11], and most of the concepts are taken directly from that implementation. The decision to port the parser was based on the need for a more iterative, event-by-event, parser and one that is a little more 'hackable' - even if arguably less elegant - than the existing parser.

Attoparsec offers a small set of default parsers with a large set of powerful combinators [11]. Parsers written using attoparsec explicitly only accept strict bytestrings because it is heavily optimized for performance. Still attoparsec offers a module to feed these parsers chunk-by-chunk with strict bytestrings from a lazy bytestring which makes it very comfortable to write low-memory, high-performance parsers without much boilerplate. It also provides a powerful mechanism for automatic backtracking.

2.4.1 The Header

The file is split into two main sections, the header and the event stream. Each `*.eventlog`-file has to contain a list of event types which may be emitted by the version of the GHC runtime it was created by. Figure 2 shows the structure of the header while listing 4 shows the code that parses it.

```
1  headerParser :: A.Parser Header
2  headerParser = do
3      _ <- A.string $ C.pack "hdrb" -- begin header
4      _ <- A.string $ C.pack "hetb" -- begin event type list
5      typeList <- A.many1' eventTypeParser
6      _ <- A.string $ C.pack "hete" -- end header
7      _ <- A.string $ C.pack "hdre" -- end event type list
8      return $ Header typeList
9
10 eventTypeParser :: A.Parser EventType
```

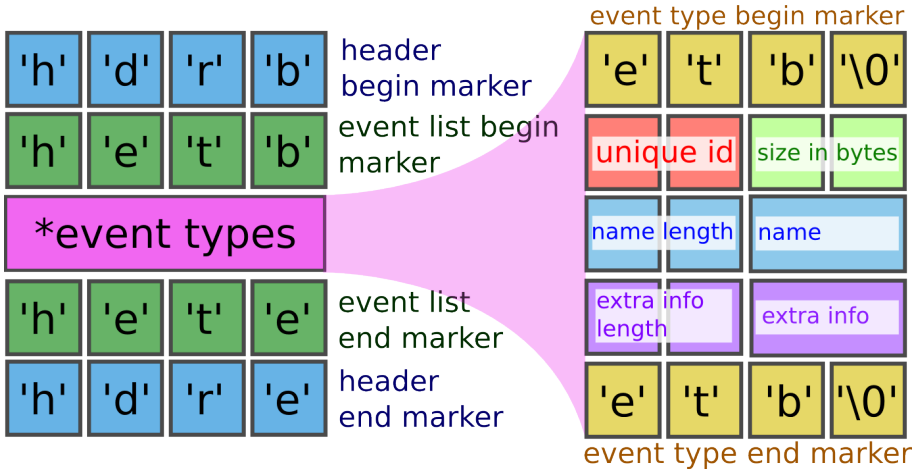


Figure 2: The structure of the header. A square represents a single byte.

```haskell
eventTypeParser = do
    _        <- A.string $ C.pack "etb\NUL" --begin event type
    id_     <- word16be <$> A.take 2
    eventTypeSize    <- word16be <$> A.take 2
    sizeName <- word32be <$> A.take 4
    name    <- A.take (fromIntegral sizeName)
    sizeExtraInfo <- fromIntegral <$> word32be <$> A.take 4
    _ <- A.take sizeExtraInfo
    _        <- A.string $ C.pack "ete\NUL" --end event type
    let v = eventTypeSize == 0xFFFF
    return $ EventType id_ (C.unpack name) (if v then Nothing else Just eventTypeSize)
```

Listing 4: The code for parsing the header.

Each event type consists of three main elements: A unique id (represented by a two byte integer), a length and a name. Each event type also has an additional field describing its name. This field was created to be used in future versions of the runtime but as of the time this thesis was conceived, it always contains an empty string.

2.4.2 The Event Stream

After successfully consuming the header, the parser interprets the stream of events. The event stream, as well as the header, is indicated by an associated start and end marker. Each event is composed of three parts: The id of the type of this event (which can be looked up in the previously parsed header), the timestamp and the actual content, which is encoded differently for every event type. The actual encodings for each type can be found in the source for the ghc-events parser and the accompanying header files, which are also part of the GHC runtime. In general there are two types of events: Fixed sized events, which have a predefined size for every field, and variable length encoded events, which carry an additional field describing their length. These are mostly used to encode strings and messages. because the Header might contain events belonging to a future version of GHC, that are not yet implemented in this parser, the event types and their accompanying lengths are put into a hash map that can be used to look up the length of unknown events during parsing. The parser module contains a large list of known event types

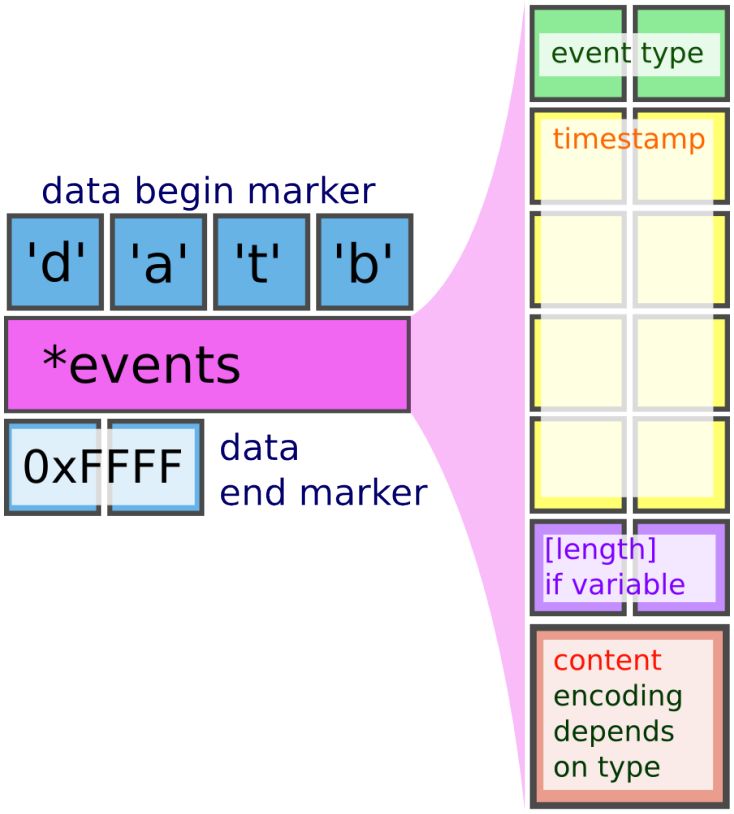


Figure 3: Graphical representation of the binary event stream

and their parser. Each entry in the list is a tuple containing the id of the event type, and a parser for that specific event. The following is a shortened version of that list:

```
knownParsers = [ (0, (\cap timestamp -> do -- CreateThread
                        threadId <- U.parseW32
                        return $ Event timestamp (CreateThread threadId))),

                 (1, (\cap timestamp -> do -- RunThread
                        threadId <- U.parseW32
                        return $  Event timestamp (RunThread threadId))),
                 (2, (\cap timestamp -> do -- StopThread
                        threadId <- U.parseW32
                        blockreason <- U.parseW16
                        i <- U.parseW32
                        return $  Event timestamp (StopThread threadId
                                  (if (blockreason > maxThreadStopStatus782)
                                    then NoStatus
                                    else if blockreason==9
                                        then BlockedOnBlackHoleOwnedBy i
                                        else mkStopStatus782 blockreason)
                                  ))),
                     ...
                 ]
```

Listing 5: The list of parsers for all known event types.

After the header has been parsed, the event ids in the header are checked against the ones in the list of known parsers: If an event id appears in the header but does not occur in the list of known parsers, a new tuple is created. This new tuple contains the id of the unknown event, and a parser that produces an `UnknownEvent` when encountering that id. That parser either consumes a fixed amount of bytes - if the header advertises that event as fixed in size - or a variable amount - if the header says it has variable size. This operation is expressed as a fold over the events that appear in the header, the result of which is used to produce an immutable array, with event ids as keys, and parsers as values.

```
type ParserTable = IA.Array Word16 EParser

mkParserTable :: Header -> ParserTable
mkParserTable h = IA.array (0,100) $ foldr addToList knownParsers (eventTypes h)
    where addToList :: EventType -> [(Word16, EParser)] -> [(Word16, EParser)]
          addToList (EventType id_ _ s) list = if (null $ filter (\x -> fst x == id_)
    list)
            then (id_ , makeUnknownParser id_ s) : (filter (\x -> fst x /= id_) list)
            else list
```

Listing 6: Creating an immutable array of parsers.

This array allows for rapid lookup of the correct parser once the event id of the event to be parsed is known. To parse a single event, the parser then carries out following three steps: Parse the 2 byte event type, parse the 8 byte timestamp, then pick the correct parser from the `ParserTable` structure and apply it to the remaining bytestring:

```
parseSingleEvent :: ParserTable -> Capability -> A.Parser (Maybe Event)
parseSingleEvent pt cap = do
    type_ <- U.parseW16
    if (type_ == 0xFFFF)
        then return $ Nothing
        else do
            timestamp <- U.parseW64 -- the timestamp
            event     <- (pt IA.! type_) cap timestamp
            return $ Just event
```

Listing 7: Parsing of single event.

2.4.3 Speeding up the parser

While the attoparsec library offers a fast interface for parsing binary files, it does not offer any functionality to interpret multi-byte bytestrings as representations of encoded numbers. In this case, the parser needed a (preferably fast) method of interpreting 2,4 and 8 byte long bytestrings as representations of unsigned 16, 32 and 64 bit integers. There is a package on hackage called attoparsec-binary that offers exactly this functionality, but for the purpose of this thesis a slightly faster approach was taken. Because Haskell has no fast and direct way to convert a bytestring into its numerical equivalent, a small C library was developed that uses the C standard library to convert bytestrings (or in this case, their equivalent `CStrings`) into integral types.

```
#include<stdint.h>
```

```c
#define _BSD_SOURCE
#define __USE_BSD
#include<endian.h>
#include"fastconvert.h"
uint16_t w16be(char* input){
    return be16toh(*((uint16_t*)input)); }
uint32_t w32be(char* input){
    return be32toh(*((uint32_t*)input)); }
uint64_t w64be(char* input){
    return be64toh(*((uint64_t*)input)); }
```

Listing 8: Using C to speedily convert bytestrings into unsigned integral types

```haskell
foreign import ccall unsafe "fastconvert.h w16be" w16be :: CString -> IO CUShort
foreign import ccall unsafe "fastconvert.h w32be" w32be :: CString -> IO CUInt
foreign import ccall unsafe "fastconvert.h w64be" w64be :: CString -> IO CULong

w16 :: B.ByteString -> Word16
w16 bs = case (unsafePerformIO $ C.useAsCString bs w16be) of
        (CUShort x) -> x

w32 :: B.ByteString -> Word32
w32 bs = case (unsafePerformIO $ C.useAsCString bs w32be) of
        (CUInt x) -> x

w64 :: B.ByteString -> Word64
w64 bs = case (unsafePerformIO $ C.useAsCString bs w64be) of
        (CULong x) -> x

parseW16 :: A.Parser Word16
parseW16 = w16 <$> A.take 2
{-# INLINE parseW16 #-}

parseW32 :: A.Parser Word32
parseW32 = w32 <$> A.take 4
{-# INLINE parseW32 #-}

parseW64 :: A.Parser Word64
parseW64 = w64 <$> A.take 8
{-# INLINE parseW64 #-}
```

Listing 9: FFI bindings for the C conversion functions

The module *Bachelor.Util* includes FFI bindings to this C library, so that these functions can be called during parsing. This C-binding approach means that the parser loses some of the safety usually associated with Haskell. Calling these functions on bytestrings of lengths different from those they were supposed to be running on will produce false results or might even cause a segfault when running the program. However, these functions are exclusively called with the according parsing function `take` (which, when called as `take n`, will either return a bytestring of length `n` or fail the parsing attempt), so that there is a reasonable expectation for the bytestrings to be of the correct length.

2.4.4 Testing the parser

After porting the parser to attoparsec, the next step was to test the implementation. For this, *hspec* [12] was chosen. *Hspec* is a testing library for Haskell, that is designed to be similar to Ruby's RSpec, and which also works with HUnit and QuickCheck. It provides a literate, declarative way to describe tests. Because parsing files from disk is a process that inherently involves side-effects, it is harder to define useful testable properties. In this case however, because the code that is tested should (for the most part) reproduce the behavior of an existing module (namely the parser for the `*.eventlog`-format defined in GHC-events-parallel), developing test becomes more simple. In the end, the code was tested against the following two assertions: That the parser will always consume the entire file it is reading - thereby returning a successfully parsed `EventLog` and an empty bytestring. Secondly, that, when run against the same file, ghc-events-parallel will produce the same output. Listing 10 shows the high-level code that calls these two test-functions.

```
main = hspec $ do
    describe "Bachelor.Parsers" $ do
        describe "eventLogParser" $ do
            mapM_ testSingleFile testfiles

testSingleFile fn = do
        it ("returns the same result as GHC.RTS.Events.readEventLogFromFile when parsing
    " ++ fn)  $ do
            (compareEventLogs fn) `shouldReturn` (Nothing, Nothing)
        it ("will consume the entire file " ++ fn) $ do
            (consume fn)`shouldReturn` True
```

Listing 10: Hspec testing code for the parser

A challenge when writing the tests for the parser module was to define the comparison with the ghc-events-parallel parser in such a way that it would produce meaningful, comprehensible output. While hspec contains functionality to test the result of a single IO action, it does not contain a straightforward way of comparing the results of two IO actions. The solution to this problem was to define a meaningful function of type

```
FilePath -> IO x
```

where x is a type that either informs the user that the two parsers returned the same result, or contains some meaningful information about how the test failed. It would, for example, not always be useful to get a list of all mismatches between the two parsers as a result - if the newly implemented parser misjudged the length of a single event type, all events in the file that occur after the first instance of this event would be parsed wrongly, even if their parsers were correctly implemented. It therefore seemed more reasonable to always terminate the test on the first mismatch, and have it displayed as a result, using a function of the following type:

```
compareEventLogs :: String -> IO ((Maybe Event), (Maybe Event))
```

Listing 11: Signature of the test function

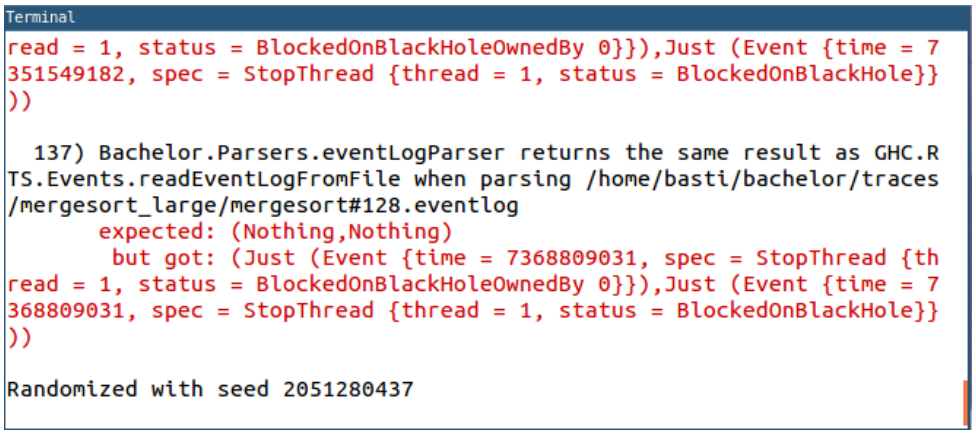


Figure 4: Example of a parser bug found by testing with hspec: Different versions of GHC encode the ThreadStopStatus differently.

Which returns (Nothing, Nothing) only if the eventlogs generated by the two parsers are identical, and otherwise returns the offending tuple of events. A tuple of the form (Just e,Nothing) or (Nothing, Just e) is returned if one of the eventlogs is shorter than the other and has run out of events.

```
1  compareEventLogs :: String -> IO ((Maybe Event), (Maybe Event))
2  compareEventLogs fn = do
3      --flatten both logs to a single list.
4      ref <- events <$> dat <$> flattenBlocks <$> reference fn
5      cus <- events <$> dat <$> removeBlocks  <$> custom fs
6      return $ compareEventList ref cus
7
8
9  compareEventList :: [Event] -> [Event] -> (Maybe Event, Maybe Event)
10 compareEventList [] [] = (Nothing,Nothing)
11 compareEventList (e:es) [] = ((Just e),Nothing)
12 compareEventList [] (e:es) = (Nothing,(Just e))
13 compareEventList (e:es) (c:cs) | e==c      = compareEventList es cs
14                               | e/=c = case (e,c) of
15                                ((Event _ StopThread{}),(Event _ StopThread{})) ->
16                                    compareEventList es cs
17                                _ ->  (Just e, Just c)
18
```

Listing 12: Comparison function for the two parsers.

3 Eden-Tracelab

This section will discuss the architecture of Eden-Tracelab, as well as the inner workings of each component.

3.1 Architecture

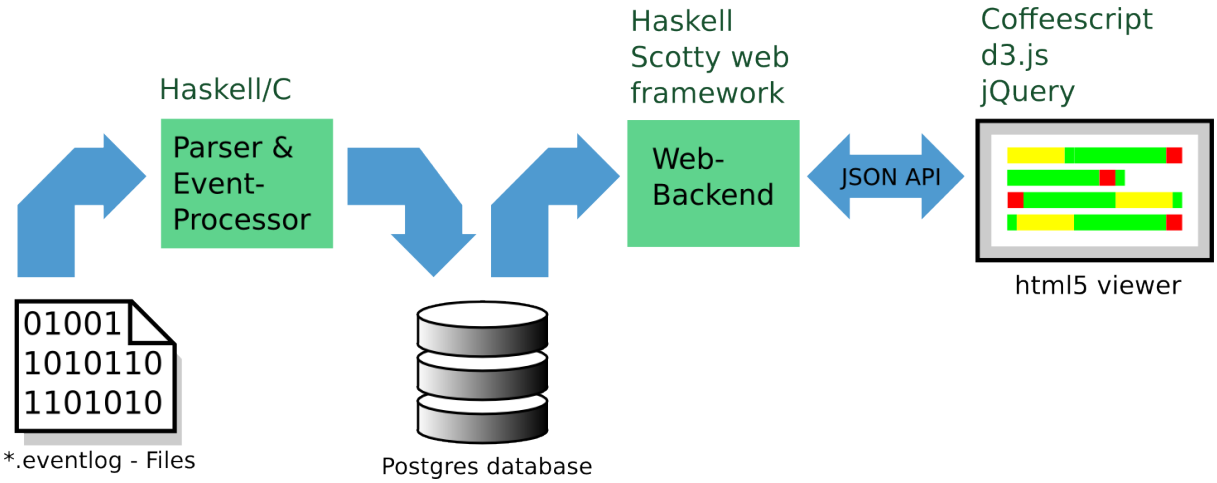


Figure 5: The general architecture of Eden-Tracelab.

Figure 5 shows the overall architecture of Eden-Tracelab. Eden-Tracelab consists of three distinct units: The parser and event processor, the web service, and the HTML5 viewer.
A detailed description of the parser module can be found in chapter 3. This chapter will give an in-depth explanation of how parsed events are processed into higher order GUI events, and how those events are exposed through the web service, and then displayed in an HTML5 viewer.

```
1   Bachelor.Types
2                   -- Defines a set of types to represent the state
3                   -- of the Eden runtime, as well as types to
4                   -- represent viewable types of higher order events in the
5                   -- GUI.
6   Bachelor.SeqParse
7                   -- Provides facilities to analyze a *.eventlog file on
8                   -- a per-event basis, and to generate higher order events.
9   Bachelor.Parsers
10                  -- Provides parsers for Events in *.eventlog files
11  Bachelor.DataBase
12                  -- Provides an interface to write GUIEvents into the
13                  -- database.
14
```

Listing 13: Overview of the parser/processor module

Figure 6: Overview of the seqparse-module, which provides facilities for sequentially parsing eventlog files, and generating higher order events to be stored in the database

3.2 Event-Processing

Processing a single `*.eventlog`-file and parsing its contents event-by-event is only the first step. In order to be stored on disk and be displayed in Eden-Tracelab these events then need to be converted into higher order events. The creation of higher order events is necessary for two reasons. Firstly, performance is increased by generating events that directly correspond to visual objects in Eden-Tracelab, as this ensures that no additional computation is necessary when displaying trace visualizations. Secondly, creating a set of events that be can be stored in a simple database schema allows for fast retrieval of visual events while still allowing for powerful and extensible analysis.

3.2.1 Events Observed

To understand the creation of higher order events in the processing, it is necessary to first understand the events that are observed by Eden-Tracelab, and how these interact with the state of the runtime.

Event	**Effect**
RunThread *tid*	Set the thread with id *tid* to Running.
WakeupThread *tid* othercap	Take the previously blocked thread with the id *tid* and set it to Running. In the multithreaded runtime, the thread might also be woken up on another capability - but not in the single-threaded Eden runtime.
StopThread *tid blockreason*	Set the thread with id *tid* to Blocked or Runnable, depending on *blockreason*. If *blockreason* is ThreadFinished, remove the thread from the runtime
ThreadRunnable *tid*	Set the thread with id *tid* to Runnable.
CreateMachine *mid*	Add the machine with the id *mid* to the runtime.
KillMachine *mid*	Remove the machine with the id *mid* from the runtime.
CreateThread *tid* & AssignThreadToProcess *tid pid*	Create the thread with the id *tid* and add it to the process with id *pid*.
CreateProcess *pid*	Add the process with id *pid* to the runtime
KillProcess *pid*	Remove the process with id *pid* from the runtime

Table 2: Events observed by EdenTV and Eden-Tracelab

Most of these events also have secondary effects on the runtime. For example, if a process gets killed, all threads contained in the process will also be killed. The next chapter will document how the parser/processor deals with these events.

3.2.2 Higher Order Events

When examining a screenshot of the current version of EdenTV, one can identify two different types of visual objects: Colored bars representing a machine, process or thread in a specific state at a point in time for a certain duration, and lines representing message passing between machines. To keep the amount of disk-IO and processing necessary to interactively display a trace to a minimum, these objects directly correspond to database entries in Eden-Tracelab. There are a number of differences between the raw events in the event stream of the binary `*.eventlog`-file, and the higher order events written to the database. First, the raw event stream contains events that do not have a duration. They simply describe a change within the runtime system from one state to another at a fixed point in time. The higher order events generated by the parsing algorithm have a duration in time. While the raw events do require some kind of context before they can be displayed in a visualization, the higher order events contain enough information so that they can be rendered as a single visual piece of information. But before these events can be written into the database, they first have to be created by the parser. Because a small memory footprint was one of the set goals in the creation of Eden-Tracelab, it was necessary to define a minimum data structure to represent the inner state of the runtime system.

The algorithm to parse and process the events generated by the parser is as follows (in pseudecode):

```
WHILE (EVENTS AVAILABLE):
    EVENT := GET NEXT CHRONOLOGICAL EVENT
    IF (EVENT CHANGES RTSSTATE):
        OLDRTSSTATE = RTSSTATE
        RTSSTATE = ADJUST RTSSTATE(EVENT)
        [HOI] = CREATE HIGHER ORDER EVENTS(OLDRTSSTATE, RTSSTATE)
        WRITE [HOI] TO DISK
```

The most complex part of this process is the creation of higher order events, and the adjustment of the RTS state in reaction to an event.
The higher order events storable in the database (and displayable in the viewer) are encoded in the following data type:

```
1 data GUIEvent = GUIEvent{
2     mtpType   :: MtpType,
3     startTime :: Word64,
4     duration  :: Word64,
5     state     :: RunState
6     } | NewMachine MachineId | NewProcess MachineId ProcessId
7       | NewThread MachineId ProcessId ThreadId deriving (Eq, Show)
```

Listing 14: Encoding of higher order events.

where MtpType is a type for a value that describes whether the event occurred on a thread, process, or machine, and provides the necessary ids to locate it in the hierarchical structure of the runtime:

```
1 data MtpType = Machine MachineId
2     | Process MachineId ProcessId
```

```
3     | Thread  MachineId ThreadId deriving (Eq, Show)
```

Listing 15: The type to define where an event occured

3.2.3 Internal Representation of the RTS State

As previously mentioned, the events in the database directly correspond to the visual events displayed in Eden-Tracelab. Therefore, the data structure to represent the state of the runtime system also has to correspond to these states. Eden-Tracelab has a set of four states that a machine, process or thread can be in - represented by four different colors:

```
1 data RunState = Idle | Running | Blocked | Runnable
2     deriving (Show, Eq)
```

Listing 16: Run state encoding

Eden-Tracelab, same as the original EdenTV is planned to have three different views: the machine view, the process view, and the thread view. Two of those are currently implemented in Eden-Tracelab, the machine view and the process view. The internal representation of the state of the runtime also has to adhere to this hierarchical structure of machines, processes and threads:

```
1 data RTSState = RTSState {
2     _rts_machine   :: MachineState,
3     _rts_processes :: ProcessMap,
4     _rts_threads   :: ThreadMap
5     } deriving Show
```

Listing 17: Data structure describing the runtime state

Note that the naming of the fields within those record types is slightly unwieldy. This is due to two reasons: First, these data types are subject to the automatic creation of lenses[13] using the `makeLenses` command, which requires them to start with an underscore. Secondly, as record field names in Haskell are functions in a global namespace, records in the same module who share field names have to avoid name collisions. In this case, this is accomplished by prefixing each field name with a prefix, followed by another underscore.

The `ProcessMap` and `ThreadMap` types are strict hash maps from `ThreadIds` to `ThreadStates` (or `ProcessId` to `ProcesState` respectively:)

```
1 type ThreadMap    = M.HashMap ThreadId ThreadState
2 type ProcessMap   = M.HashMap ProcessId ProcessState
```

Listing 18: Collections of threads and processes.

In order to keep track of the running state of a machine -whether it is `Idle`, `Runnable`, `Running` or `Blocked`- the `MachineState` data type contains a number of counter fields, representing the number of `Runnable`, `Running` or `Blocked` processes within the machine. The `PreMachine` constructor ensures that a valid state of the runtime can be initialized before the first machine has been started.

```
1 data MachineState = MachineState {
2     _m_state     :: RunState,
```

```
    _m_timestamp :: Timestamp,
    _m_pRunning  :: Int,
    _m_pRunnable :: Int,
    _m_pBlocked  :: Int,
    _m_pTotal    :: Int
    } | PreMachine deriving Show
```

Listing 19: Data structure describing the machine state

The data type describing the state of a single process works in almost the same way, with counters for the threads belonging to this process. In addition to the fields similar to the `MachineState` it also contains a field called `_p_parent`, which holds the `MachineId` of the parent machine.

```
data ProcessState = ProcessState {
    _p_parent    :: MachineId,
    _p_state     :: RunState,
    _p_timestamp :: Timestamp,
    _p_tRunning  :: Int,
    _p_tRunnable :: Int,
    _p_tBlocked  :: Int,
    _p_tTotal    :: Int
    } deriving (Show, Eq)
```

Listing 20: Data structure describing the process state

The `ThreadState` is similar to the state of a process, but because threads do not contain any 'smaller' entities than themselves, they do not need any counter variables.

```
data ThreadState  = ThreadState {
    _t_parent      :: ProcessId,
    _t_state       :: RunState,
    _t_timestamp   :: Timestamp
    } deriving Show
```

Listing 21: Data structure describing the thread state

3.2.4 From Raw Events to Higher Order Events

After successfully defining a useful representation of the runtime system as Haskell data types, it becomes necessary to consider how a single event would influence the runtime system. A single event has two non-exclusive possibilities of interacting with the runtime system: It can change the runtime state or it can generate one or more higher order `GUIEvent` to be written into the database. All events that are observed by Eden-Tracelab change the state of the runtime system, but not all of them generate higher order events. The signature of the function that handles these events is therefore:

```
type Handler = RTSState -> AssignedEvent -> (RTSState,[GUIEvent])
```

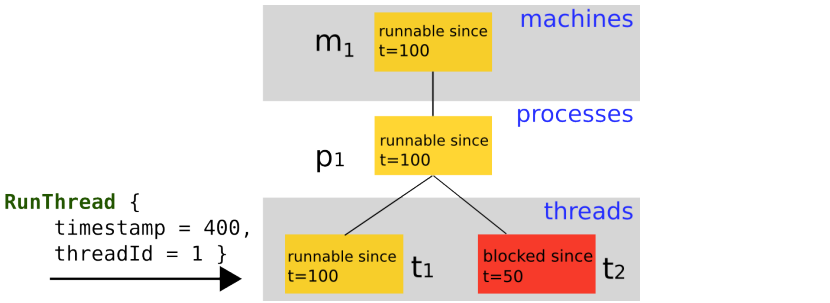
Listing 22: Signature of the event handling function

The `AssignedEvent` type is a wrapper around the `Event` type from GHC.Events.Parallel which adds some additional information, specifically the capability the event was generated by, and the machine it is from.

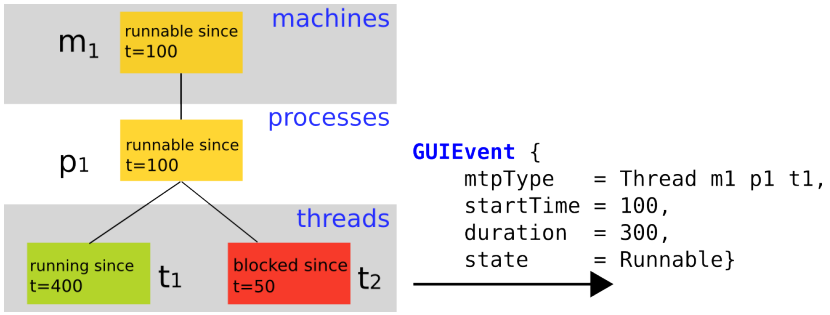
```
data AssignedEvent = AssignedEvent {
    _event :: Event,
    _machine :: MachineId,
    _cap :: Int
    }
```

Listing 23: Wrapping the event type

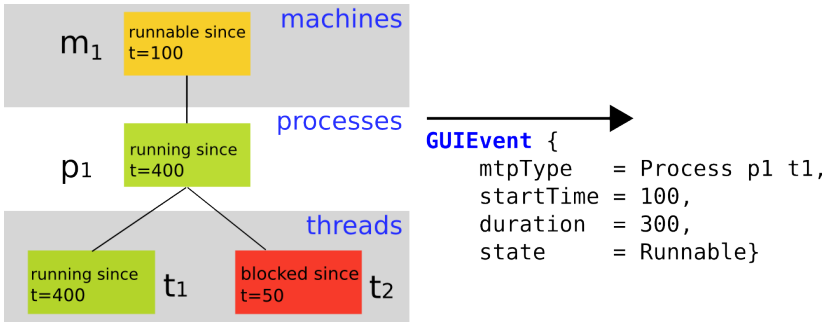
Every event interacts with the runtime in a different way, and has a different amount of `GUIEvents` it potentially generates. To demonstrate how an event might interact with the runtime system, Fig. 7 shows how a `RunThread` event might interact with the runtime system, and which events it might generate. This example was chosen because events that interact with the state of a thread introduce the most complexity into the handling of events: They have the potential to change the state of their parent processes, which in turn might cause the parent machine to change its run state.



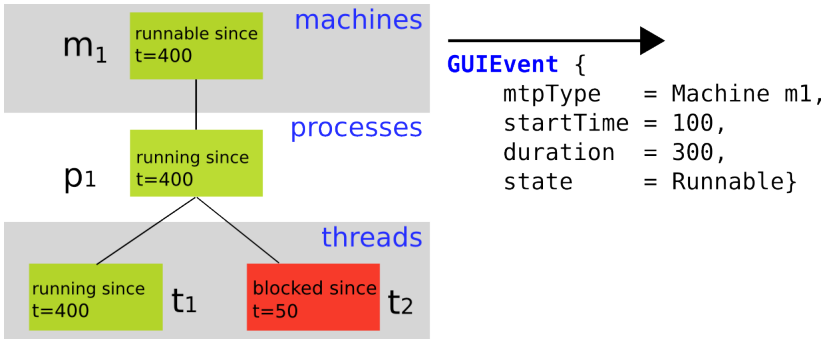
(a) The runtime system state contains a single machine, with a single process. The process contains two threads, one blocked, one runnable. The next event being that the parser is going to read is a RunThread event for the thread t_1.



(b) The state of the thread is adjusted, and a single GUIEvent is generated, describing the time from 100 to 400 where the state was runnable.



(c) The process p_1 now has a single running thread, and is therefore running. The state of the process has to be changed, and another GUIEvent has to be created, describing the runnable state of the process from time 100 to 400.



(d) The machine m_1 now has a single running process, and is therefore running. The state of the machine has to be changed, and another GUIEvent has to be created, describing the runnable state of the machine from time 100 to 400.

Figure 7: A RunThread event interacts with the runtime system and generates three GUIEvents, as well as a new runtime state.

A number of events behaves in the same way as the `RunThread` Event, by changing the state of a single thread, potentially causing a chain of state changes to 'bubble up' through the runtime system. All of these events are handled by the same function, namely `changeThreadState`.

```
changeThreadState :: RTSState -> MachineId -> ThreadId -> RunState-> Timestamp
                     -> (RTSState, [GUIEvent])
changeThreadState rts mid tid state ts =
    if M.member tid (rts^.rts_threads)
        then let
            oldThread           = (rts^.rts_threads) M.! tid
            oldState            = oldThread^.t_state
            pid                 = oldThread^.t_parent
            oldProcess          = (rts^.rts_processes) M.! pid
            (newThread,tEvent)  = setThreadState mid tid oldThread ts
                state
            (newProcess,pEvent) = updateThreadCountAndProcessState
                mid pid ts oldProcess (Just oldState) (Just state)
            oldProcessState     = oldProcess^.p_state
            newProcessState     = newProcess^.p_state
            (newMachine,mEvent) = updateProcessCountAndMachineState mid ts
                (rts^.rts_machine) (Just oldProcessState)
                (Just newProcessState)
            rts' = set rts_machine newMachine $
                   set (rts_threads.(at tid))   (Just newThread)  $
                   set (rts_processes.(at pid)) (Just newProcess) $ rts
        in (rts', mList [tEvent, pEvent, mEvent])
    --ignore 'homeless' threads.
    else (rts,[])
```

Listing 24: A thread changes its state as a result of an event.

This code makes heavy use of lenses to deal with the manipulation of the internal parser state. Additionally, all identifiers are chosen in a very literal fashion – without (hopefully) being too verbose – to keep the code readable. Similar functions exist for all other events that are observed by the event processor.

3.2.5 Database Schema

Figure 8 shows the database schema used by the event processor to store event data.

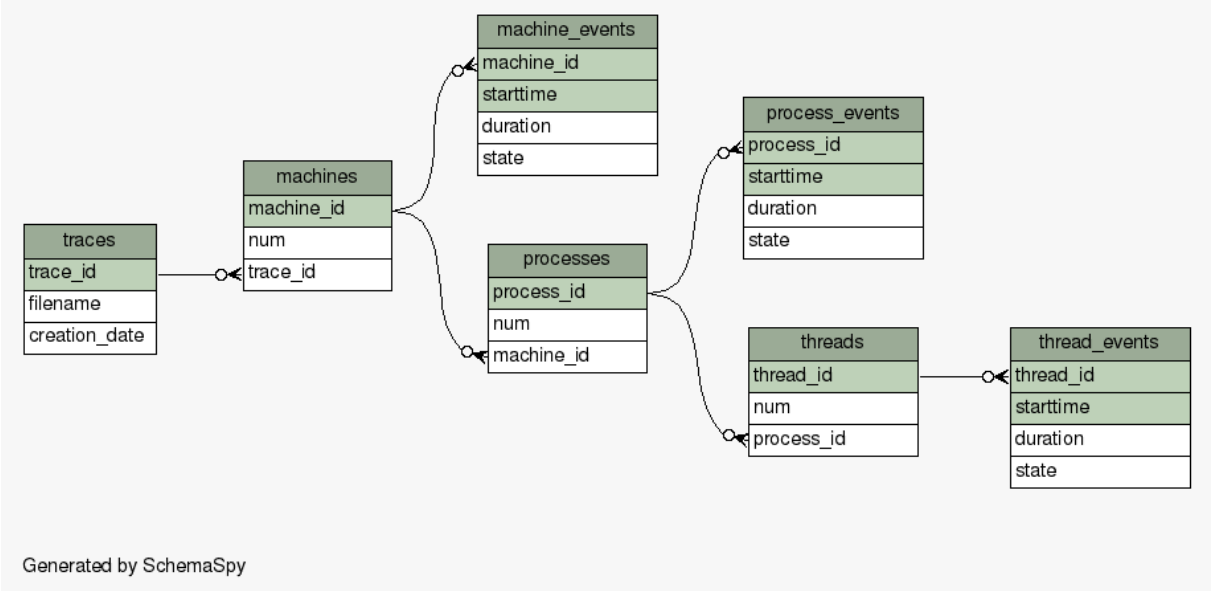


Figure 8: The schema of the event database.

The database schema is in Third Normal Form, and separately represents the hierarchical structure of the traces contained within the database as well as the events belonging to each entity. The traces, machines, processes and threads relations contain the structure of each trace: Depicting which trace contains which machines, which machines contain which processes, and which processes contain which threads. Note that the naming here is slightly confusing: In the context of each `*.eventlog`-file, each machine, process and thread has a unique id. But due to the fact that each trace contains multiple `*.eventlog`-files, and the database contains multiple traces, these ids cannot be used as primary keys in the database. The machine_id, process_id and thread_id columns contain a unique auto-incrementing primary key distinct from the previously mentioned ids within the *.eventlog-files. Those keys are being stored in the columns named num. The starttime columns of each event relation are also indexed to increase performance of event retrieval.

3.2.6 Saving Higher Order Events into the Database.

The process of inserting higher order events into the database can best be understood by looking at the main data structure of the `Bachelor.DataBase` module.

```
bufferlimit = 10000

data DBInfo = DBInfo {
    db_threadbuffer  :: [(MachineId, ThreadId,  Timestamp, Timestamp, RunState)],
    db_processbuffer :: [(MachineId, ProcessId, Timestamp, Timestamp, RunState)],
    db_machinebuffer :: [(MachineId, Timestamp, Timestamp, RunState)],
    db_traceKey      :: Int,
    db_machines      :: M.HashMap MachineId Int,
    db_processes     :: M.HashMap (MachineId,ProcessId) Int,
```

```
    db_threads    :: M.HashMap (MachineId,ThreadId) Int,
    db_connection :: Connection
    }
```

Listing 25: The DBInfo structure stores the connection to the database as well as lists of not yet inserted events and a set of primary keys

This data structure consists of three main parts: a set of lists, which serve as buffers for events before they are written to the database, a set of maps of primary keys for machines, processes and threads, as well as a single key identifying the trace in the database, and lastly the connection to the database.
The buffers exist for performance reasons: Batch insertion into the database is about a magnitude faster than inserting each event on its own. The reason for the existence of these hash maps is related to the database schema laid out in the previous chapter: Because the machine id stored in the `*.eventlog`-file does not identify the machine in the database - there might be an arbitrary number of machines belonging to other traces having the same id - an additional key is created in the database. This key is needed when a machine event has to be inserted into the database, therefore it is kept in the map db_machines. The same is true for processes and threads: The tuple (machine id, process id) identifies a process in a trace, but not in the database, while the tuple (machine id, thread id) identifies a thread in the trace, but not in the database. The primary keys for processes and threads are being stored with these tuples as keys.

The main function of the database module is the following:

```
insertEvent :: DBInfo -> GUIEvent -> IO DBInfo
insertEvent dbi g@(NewMachine mid)        = insertMachine dbi mid
insertEvent dbi g@(NewProcess mid pid)    = insertProcess dbi mid pid
insertEvent dbi g@(NewThread mid pid tid) = insertThread  dbi mid pid tid
insertEvent dbi g@(GUIEvent mtpType start dur state) =
    case mtpType of
        Machine mid -> case ((length $ db_machinebuffer dbi) >= bufferlimit) of
            True  -> do
                putStrLn "inserting Machine events"
                insertMachineState dbi
            False -> return dbi {
                db_machinebuffer = (mid,start,dur,state) : db_machinebuffer dbi
                }
        Process mid pid -> case ((length $ db_processbuffer dbi) >= bufferlimit) of
            True  -> do
                putStrLn "inserting Process events"
                insertProcessState dbi
            False -> return dbi {
                db_processbuffer = (mid,pid,start,dur,state) : db_processbuffer dbi
                }
        Thread  mid tid -> case ((length $ db_threadbuffer dbi) >= bufferlimit) of
            True  -> do
                putStrLn "inserting Thread events"
                insertThreadState dbi
            False -> return dbi {
                db_threadbuffer = (mid,tid,start,dur,state) : db_threadbuffer dbi
```

```
                }
```

Listing 26: Inserting events into the database.

Note that this function distinguishes between two major types of events: Events with constructors `NewMachine`, `NewProcess`, `NewThread` are inserted into the database immediately. This is due to the fact that they create new primary keys for machines, processes or threads, which might be needed when inserting other events. For example, the event `NewMachine 2` should be inserted before any event on machine 2 is being inserted, so that the primary key already exists when such an event occurs. The second type are events that do not describe the structure of the trace, but directly correspond to events shown in the viewer. Those events do not create any new keys in the database, and therefore they do not have to be inserted immediately. Instead, they can be inserted into a buffer, and are only inserted when the buffer size exceeds a threshold: In this case, 10000 events. The following function performs insertion when the buffer of thread events has reached the limit:

```
insertThreadState :: DBInfo -> IO DBInfo
insertThreadState dbi = do
    let conn = db_connection dbi
        inlist = map (\(mid,tid,start,duration,state) ->
            ((db_threads dbi) M.! (mid,tid), start, duration, stateToInt state))
                $ db_threadbuffer dbi
    executeMany conn insertThreadStateQuery inlist
    return dbi {db_threadbuffer = []}

insertThreadStateQuery :: Query
insertThreadStateQuery =
    "Insert into thread_events(thread_id, starttime, duration, state)\
    \values( ? , ? , ? , ? );"
```

Listing 27: Inserting a thread event into the database.

The functions for inserting events regarding events occurring on processes and machines are very similar. Note that the primary key is looked up in the hash map upon insertion, and that the state is encoded as an integer.

The function for inserting a new thread into the database shows how a a new primary key is inserted into the threads table, and how it is then subsequently stored in the database module:

```
insertThreadQuery :: Query
insertThreadQuery =
    "Insert into Threads(num, process_id)\
    \values( ? , ? ) returning thread_id;"

insertThread :: DBInfo -> MachineId -> ProcessId -> ThreadId -> IO DBInfo
insertThread dbi mid pid tid = do
    let conn       = db_connection dbi
        processKey = (db_processes dbi) M.! (mid,pid)
    threadKey <- head <$> query conn insertThreadQuery (tid, processKey)
    case threadKey of
        Only key -> do
```

```
            return $ dbi {
                db_threads    = M.insert (mid,tid) key (db_threads dbi)
                }
        _       -> error "thread insertion failed"
```

Listing 28: Inserting a new thread into the database

To ensure that every event has been written when the event processing finishes, the database module contains the function `finalize` that writes the contents of all buffers into the database.

```
finalize :: DBInfo -> IO DBInfo
finalize dbi = do
    dbi <- insertMachineState dbi
    dbi <- insertProcessState dbi
    dbi <- insertThreadState  dbi
    return dbi
```

Listing 29: Inserting remaining events into the database

3.2.7 Processing an Entire Trace

This chapter explains how all the previously introduced components in the event processor work together. This happens in the `Bachelor.SeqParse` module which takes the iterative parser, the creation of higher order events and the database facilities to analyze a trace and write all the created higher order events to the database. The main function in this module is the following:

```
run :: FilePath -> IO()
run dir = do
    -- filter the directory contents into eventlogs.
    paths <- filter (isSuffixOf ".eventlog") <$> Dir.getDirectoryContents dir
        -- prepend the directory.
    let files = map (\x -> dir ++ x) paths
        -- extract the machine number.
        mids  = map extractNumber paths
    -- create a parserState for each eventLog file.
    pStates <- zip mids <$> mapM createParserState files
    -- connect to the DataBase, and enter a new trace, with the current
    -- directory and time.
    conn <- DB.mkConnection
    PG.withTransaction conn $ do
        dbi <- DB.insertTrace conn dir
        --parse the events belonging to a single machine, and
        --insert the created higher order events into the database
        dbi <- foldM handleMachine dbi pStates
        dbi <- DB.finalize dbi
        return ()
```

Listing 30: Processing an entire trace.

Note that this function takes a directory as an argument, not a `*.parevents` file. Eden traces are stored as `*.parevent` files, which are a zip archive containing the `*.eventlog`-file of all cores/machines included in the trace. Originally, the parser was supposed to lazily read files from the zip archive directly. However,

it turned out to be difficult to find a library that lazily reads files from zip archives with a low memory footprint. In the current version, `*.parevent`-files have to be unzipped by hand into a directory, which can then be passed to the parser/event processor. Next, the parser opens each file as a lazy bytestring (in the `createParserState` function) and attaches an empty runtime system state. The relevant data types can be seen here:

```
type CapState = (LB.ByteString, Maybe Event)
data ParserState = ParserState {
    _p_caps     :: CapState,       -- the 'system' capability.
    _p_cap0     :: CapState,       -- capability 0.
    _p_rtsState :: RTSState,       -- the inner state of the runtime
    _p_pt       :: ParserTable     -- event types and their parsers,
                                   -- generated from the header.
        }
```

Listing 31: Data structure to describe the parser state

As previously discussed, the parser state encompasses a lazy bytestring for each capability (system capability and capability #0) as well as the chronologically latest event from both.
After such a state has been generated for each `*.eventlog`-file file, a database connection is established, and the files are parsed and processed one by one. The relevant function in the code shown above is the `handleMachine` function, which is a thin wrapper around the `parseSingleEventLog` function.

```
handleMachine :: DB.DBInfo -> (MachineId, ParserState) -> IO DB.DBInfo
handleMachine dbi (mid,pstate) = do
    print $ "Processing Machine no ." ++ (show mid)
    dbi <- parseSingleEventLog dbi mid pstate
    return dbi

{-
 - This is the main function for parsing a single event log and storing
 - the events contained within into the database.
 - -}
parseSingleEventLog :: DB.DBInfo -> MachineId -> ParserState -> IO DB.DBInfo
-- event blocks need to be skipped without handling them.
-- System EventBlock
parseSingleEventLog dbi mid pstate@(ParserState
    (bss,evs@(Just (Event _ EventBlock{})))
    _ rts pt) = parseSingleEventLog dbi mid $ parseNextEventSystem pstate
-- Cap 0 EventBlock
parseSingleEventLog dbi mid pstate@(ParserState
  _ (bs0,e0@(Just (Event _ EventBlock{})))
    rts pt) = parseSingleEventLog dbi mid $ parseNextEventNull pstate
-- both capabilities still have events left. return the earlier one.
parseSingleEventLog dbi mid pstate@(ParserState
    (bss,evs@(Just es@(Event tss specs)))
    (bs0,ev0@(Just e0@(Event ts0 spec0)))
    rts pt) = if (tss < ts0)
        then do
            let aEvent = AssignedEvent es mid (-1)
                (newRTS, guiEvents) = handleEvents (pstate^.p_rtsState) aEvent
```

```
                pstate' = set p_rtsState newRTS $ pstate
            dbi <- foldM DB.insertEvent dbi guiEvents
            parseSingleEventLog dbi mid $ parseNextEventSystem pstate'
        else do
            let aEvent = AssignedEvent e0 mid 0
                (newRTS, guiEvents) = handleEvents (pstate^.p_rtsState) aEvent
                pstate' = set p_rtsState newRTS $ pstate
            dbi <- foldM DB.insertEvent dbi guiEvents
            parseSingleEventLog dbi mid $ parseNextEventNull pstate'

-- no more system events.
parseSingleEventLog dbi mid pstate@(ParserState
    (bss,evs@Nothing)
    (bs0,ev0@(Just e0@(Event ts0 spec0)))
    rts pt) = do
            let aEvent = AssignedEvent e0 mid 0
                (newRTS, guiEvents) = handleEvents (pstate^.p_rtsState) aEvent
                pstate' = set p_rtsState newRTS $ pstate
            dbi <- foldM DB.insertEvent dbi guiEvents
            parseSingleEventLog dbi mid $ parseNextEventNull pstate'
-- no more cap0 events.
parseSingleEventLog dbi mid pstate@(ParserState
    (bss,evs@(Just es@(Event tss specs)))
    (bs0,ev0@Nothing)
    rts pt) = do
            let aEvent = AssignedEvent es mid (-1)
                (newRTS, guiEvents) = handleEvents (pstate^.p_rtsState) aEvent
                pstate' = set p_rtsState newRTS $ pstate
            dbi <- foldM DB.insertEvent dbi guiEvents
            parseSingleEventLog dbi mid $ parseNextEventSystem pstate'
-- no more events.
parseSingleEventLog dbi mid pstate@(ParserState
    (bss,evs@Nothing)
    (bs0,ev0@Nothing)
    rts pt) = return dbi
```

Listing 32: Parsing an entire eventlog

As can be seen here, the main work being done by this function is first to decide which capability to accept the next event from, so that chronological order is preserved. After that, the event is being processed, with the `handleEvents` function (see section 'Higher Order Events'). The newly generated higher order events are then passed on to the database handler, which decides when to input them into the database. Lastly, this process is repeated recursively with the next event, until the end of this `*.eventlog`-file has been reached. After all eventlogs have been treated in this manner, the `run` function calls `finalize` on the database, to insert any events that are still remaining in the buffer. At this point, the entire trace has been analyzed and fed into the database, so that it can be retrieved by the web service and viewed using the HTML5 viewer.

3.3 Web Service

After the higher order events have been written into the database, they need to be served to the HTML5 viewer. To accomplish this, a web service was developed using the Scotty web framework. This chapter documents the architecture of this web service, as well as the API serving the event data as JSON.

3.3.1 API

The desired feature set of the Eden-Tracelab viewer puts a series of requirements on the API. First, to be able to choose a previously created trace analysis from the database when starting up the viewer, a method for retrieving information about existing traces is needed. Before event information about a specific trace can be extracted, the overall structure of the trace needs to be retrieved. Specifically, the following details need to be established: the number of machines that the trace contains; their ids; the processes they contain; their ids; the threads they contain; their ids. This information is kept in the HTML5 viewer to reduce the amount of processing necessary when dynamically viewing a trace and loading new data from the web service via AJAX-queries. Then, when viewing a trace, the specific events for each of the three views need to be extracted respectively. It should be noted that at this time only the first two views have been implemented, and that therefore there are no API calls for retrieving thread view data. Each of these commands needs to be able to accept a set of parameters. To constrain the number of events rendered, and thereby reducing processing and rendering time, the web service needs to be able to only return events longer than a given duration and intersecting with a given time window. The parameters are therefore: a starting time of a time window, an according end time, and a minimum duration.

3.3.2 API Methods and Parameters

The API methods of the web service can be split into two main categories: Methods that retrieve metadata about the available traces - which traces are available and what their internal structure is. The second category of methods is used to retrieve higher order events which can be directly rendered as blocks of state within the HTML5 viewer. The following table shows all currently available API calls:

Method	URL	Description	Parameters
POST	/traces	Returns a list of traces that have already been analyzed	<None>
POST	/traceinfo	Returns the structure of all machines, processes and traces that occurred during the runtime of this trace.	**id : the trace_id of the trace**
POST	/duration	Returns the length of the entire trace in nanoseconds.	**id : the trace_id of the trace**
POST	/mevents	A list of machine events matching the given parameters.	**id : the trace_id of the trace** **start : the start time in ns** **end : the end time in ns** **minduration : the minimum duration of events to retrieve in ns**
POST	/pevents	A list of process events matching the given parameters.	**id : the trace_id of the trace** **start : the start time in ns** **end : the end time in ns** **minduration : the minimum duration of events to retrieve in ns**

Table 3: API calls

3.3.3 Example API Calls

The following example API calls demonstrate the JSON returned by the web service. Each example includes the URL being called, the parameters being passed, and an example result. The JSON has been pretty-printed to better illustrate its structure.

URL	parameters
/traces	None

```
[
    {
        "date": "2015-09-22 12:29:04.301641",
        "filename": "/home/basti/bachelor/traces/mergesort_small/",
        "id": 5
    },
    {
        "date": "2015-09-22 13:27:55.0637",
        "filename": "/home/basti/bachelor/traces_lukas/bitonic_12_10000000_8_+RTS_-N4_-qQ500M_-ls_-RTS/",
        "id": 6
    }
]
```

Listing 33: Example output for the /traces API call

Returns an array of trace objects, each containing a unique id, a filename of the directory they were created from, and their creation date.

URL	parameters
/traceinfo	id=8

```
[
    [ 1,
        [
            [
                1, [ 1, 2, 7 ]
            ],
            [
                2, [ 3, 4, 9 ]
            ],
        ]
    ]
]
```

Listing 34: Example output for the /traceinfo API call

Returns an array of nested arrays, that represent the structure of the machine-process-thread hierarchy of the trace. The result above describes a trace that contains a single machine with id 1, which contains two processes: One with id 1 - containing threads 1, 2 and 7 - and another process with id 2 - containing thread 3,4 and 9

URL	parameters
/mevents	id=1, start=0, end=10000000, minduration=10

The start, end and duration parameters are in nanoseconds.

```
[
    [
        2,
        2021,
        831954433,
        0
    ],
    [
        7,              //machine id
        2021,           //start time in nanoseconds
        832017342,      //duration in nanoseconds
        0               //state
    ]
]
```

Listing 35: Example output for the /mevents API call

Returns an array of arrays. The inner arrays each represent an event. The first entry identifies the machine they belong to, the next entry is the starting time, the third entry provides the duration of the event, and the last entry is an encoding of the state for that time.

URL	parameters
/pevents	id=1, start=0, end=10000000, minduration=10

The start, end and duration parameters are in nanoseconds.

```
[
    [
        1,
        1,
        1806995,
        7995,
        3
    ],
    [
        1,          //machine id
        1,          //process id
        1814990,    //start time in nanoseconds
        997500,     //duration in nanoseconds
        1           //state
    ]
]
```

Listing 36: Example output for the /pevents API call

Returns an array of arrays. The inner events represent a process event. Similar to /mevents, but also contains the id of the process.

3.3.4 Implementation

As previously noted, the web service is implemented using the Scotty web framework. The entire API is implemented in the main function of the module containing the web service. The central part of every Scotty application is the `scotty` function, which takes a port number to start a web server on, and a ScottyM() Action - a monad to describe requests and responses in. The following code is a stripped down version of the code that serves the HTML5 viewer on the Scotty server:

```
main = do
    conn <- connect myConnectInfo
    scotty 3000 $ do
        get "/" $ do
            file "./view/index.html"
```

Listing 37: Serving index.html

The `get` function answers a get request on the specified path, in this case the root of the web server, and the file function responds with the specified file, in this case the `index.html` file which contains the HTML for the HTML5 viewer. `conn` is the connection to the postgres database, which is used to handle the previously described API calls. Similar to the event processor, the API web service uses the postgresql-simple [14] module to interact with the database. To explain how the API is implemented, the following documents the implementation of a single API call, the `/mevents` function. (Note that this again is a stripped down version.)

```
main = do
    conn <- connect myConnectInfo
    scotty 3000 $ do
        post "/mevents" $ do
            id       <- param "id"
            start    <- param "start"
            end      <- param "end"
            minduration <- param "minduration"
            evs <- liftIO $ getMachineEvents conn id start end minduration
            json $ evs
```

Listing 38: Implementation of the /mevents API call

Similar to the `get` function, the `post` function answers an http request on a specified path. In this case, it handles a POST request on the path "/mevents". The `param` function extracts a named parameter from the POST request. After extraction of the parameters, the `getMachineEvents` function is called to query the desired events from the database:

```
getMachineEvents conn trace_id start end minduration = do
    evs <- query conn [sql|SELECT NUM, STARTTIME, DURATION, STATE FROM (MACHINE_EVENTS
    JOIN MACHINES
    ON MACHINE_EVENTS.MACHINE_ID = MACHINES.MACHINE_ID)
    WHERE ? <= (STARTTIME + DURATION)
```

```
    AND   DURATION  >= ?
    AND   STARTTIME <= ?
    AND   MACHINES.MACHINE_ID in
    (SELECT MACHINE_ID FROM MACHINES WHERE
    TRACE_ID = ?)|] (start, minduration, end, trace_id)
    return evs
```

Listing 39: Retrieving machine events from the database

This function uses the SQL quasiquoters included in postgresql-simple to directly embed SQL in the Haskell source. The json function, which answers the request by encoding its argument in JSON, uses the powerful aeson [15] JSON library for encoding.

3.4 HTML5 Viewer

The HTML5 viewer for graphical inspection of the generated traces was not developed in Haskell, but rather written in CoffeeScript [16], and then compiled to JavaScript. This choice was made because the ecosystem of GUI toolkits in Haskell is (in the authors opinion) still lacking, and building larger interfaces in Haskell can be time intensive. This is especially true when a hardware-accelerated context is needed while the same thing is easily achievable in modern browsers.

Several different technologies were chosen for the developed of the graphical interface. The web service was developed in CoffeeScript, a "little language that compiles into JavaScript" [16]. Unlike JavaScript, CoffeeScript is safer and more elegant to develop in, as it eliminates some of the most common pitfalls when developing JavaScript (such as scoping, missing var declarations, and use of the ==-operator). In addition to these safety features, CoffeeScript is also often shorter and more readable than the equivalent JavaScript code. The added syntactic sugar also exposes the functional parts of JavaScript. Because CoffeeScript compiles directly to JavaScript, it can seamlessly interoperate with any given JavaScript library. For this thesis, two of those libraries where chosen: *D3.js* [17] for generating visualizations of data, and *jQuery* [18] for modifying the DOM and generating AJAX-queries.

D3.js stands for Data-Driven Documents, a library for generating data visualizations in JavaScript. It offers functionality for creating data-driven graphics in the browser. While the default rendering target within D3 is svg (scalable vector graphics), its data-binding methods can also be used to create graphics on other targets, such as the HTML5 canvas. The tool developed in this thesis renders trace visualizations on an HTML5 canvas. This decision was made after discovering that rendering multiple 10,000 svg objects will, on a current browser still takes a noticeable amount of time, and manipulating this amount of objects within the DOM (as when zooming into a trace visualization) will create a laggy and unresponsive UI. However, when the trace events are displayed on a HTML5 canvas, the renderings themselves do not have to be kept and manipulated within in the DOM - they are simply redrawn as soon as the user performs any interactions with the interface. Due to the fact that the canvas object in most browsers is heavily optimized for rendering performance, this creates a far smoother UI experience than the direct svg approach.

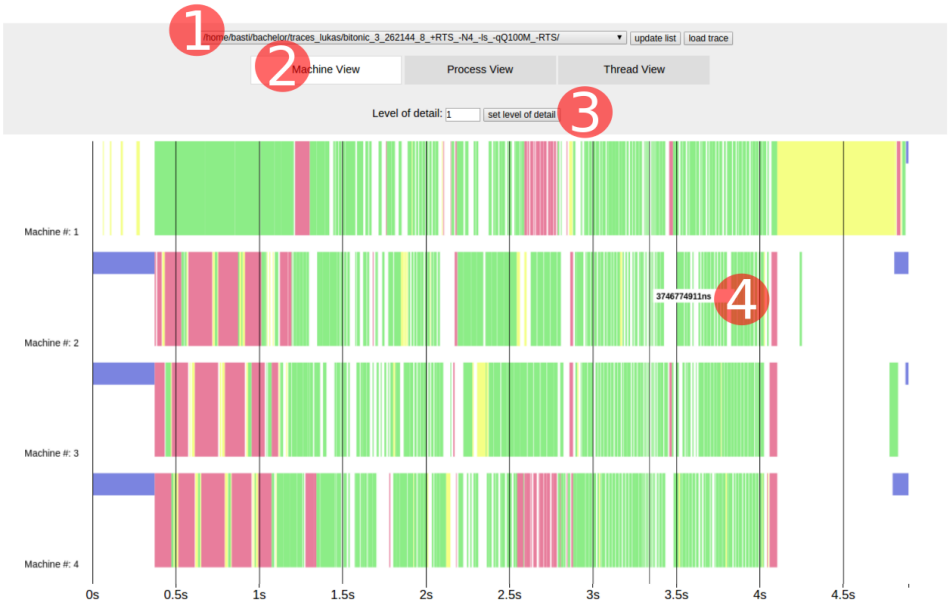
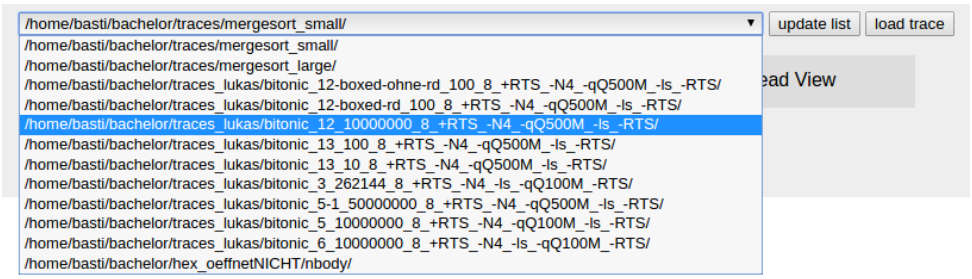


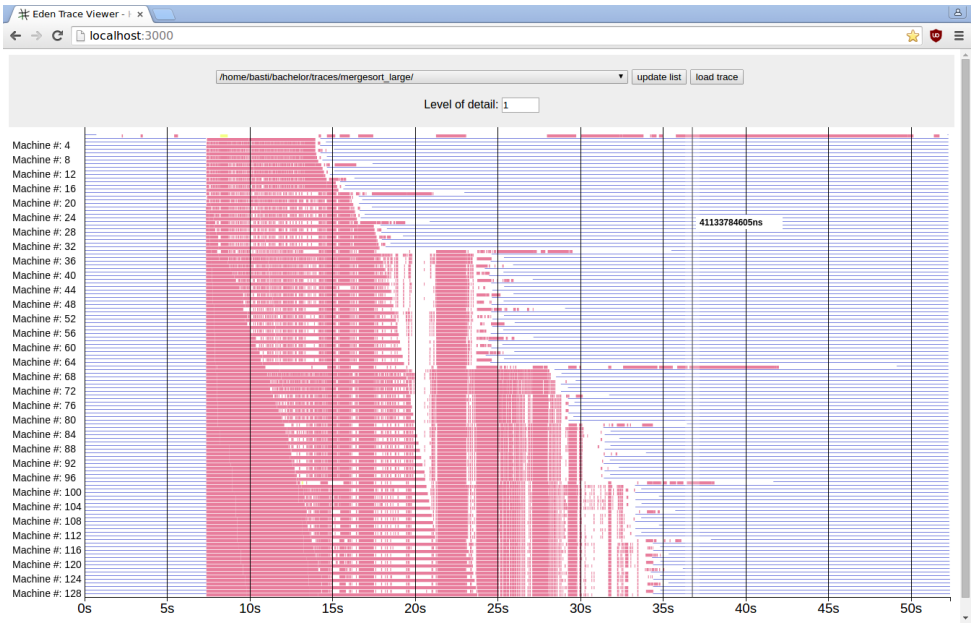
Figure 9: Screenshot of the HTML5 viewer

3.4.1 UI Overview

Figure 9 shows the interface of the newly developed tool. On startup, the user is presented with a selection of previously analyzed traces, marked with the number 1 in the above screenshot. This list can be refreshed by pressing the update button - for example when a new trace has been inserted while the HTML5 viewer was running. Selecting a trace, and clicking the 'load trace' button will load a visualization of the selected trace. By default, the trace loads in the machine view. This can be changed with the buttons marked by the number 2. At this point, only the first two views have been implemented, clicking the third one will bring up an alert. The number 3 marks the level detail setting which will be explained below. The actual trace visualization area is marked by the number 4. The trace visualization is a graph with the running time of the program on the x axis, and the different machines or processes (depending on the view) on the y axis. Each colored block represents a period of time where the machine or process was at a specific state. The color coding of the states is identical to EdenTV, and can be seen in table 1. When the user moves her cursor over a trace, a vertical line appears at the cursor position, allowing for easy comparison between machines or processes. The time in nanoseconds at that specific position is displayed to the right of the cursor. After the trace is loaded, the user can zoom into the selected trace and move it horizontally. This can be accomplished using panning-/ and zooming gestures. On a PC the user would simply use the scroll wheel of her mouse for zooming and dragging the visualization while the left mouse button is pressed for panning. Because the user interface is implemented using D3.js 'behavior' system, zooming and panning gestures can also be input by multi touch-gestures on phones and tablets, dragging the trace from left to right, or moving two fingers apart to zoom.



(a) On first load, Eden-Tracelab presents the user with a selection of previously analyzed traces.



(b) Selecting a specific trace and pressing 'load trace' will load a visualization of that trace.

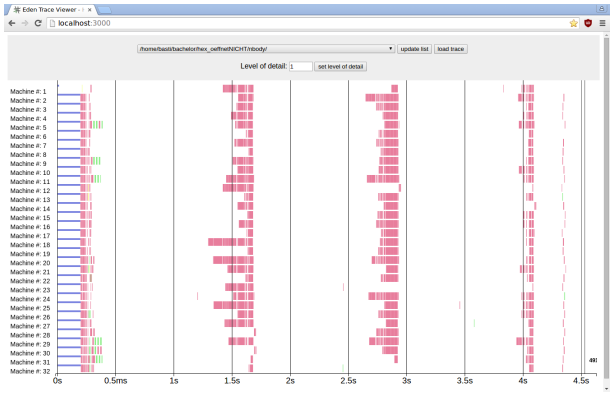
Figure 10: Loading a trace visualization in Eden-Tracelab

3.4.2 Level of Detail

The interface also offers the option to specify a desired level of detail, by adjusting it with a spin input and pressing the button next to it. To understand what this button does, one has to understand how Eden-Tracelab handles event loading to achieve performance. By default, the level of detail is set to one. This means that the HTML5 interface will only request events from the web service that, when drawn, would have a length of a single pixel or longer. Setting the level of detail to a value of n will cause the viewer to request all events which correspond to a duration longer or equal to $1/n$th of a pixel. The exact formula for the minimum duration t of events to be requested is therefore:

$$t = \frac{t_1 - t_0}{WIDTH * n}$$

Where t_0 is the starting time of the currently displayed axis, t_1 is the ending point of the currently displayed axis, and $WIDTH$ is the width of the interface in pixels.



(a) Trace visualization of a program running a N-body simulation on 32 cores. At a level of detail of 1 it is obvious that there are distinct moments in time where all cores were blocked. However, it is completely unclear what their state was in the meantime.

(b) Setting the level of detail to 10 reveals that the machines were mostly running in between being blocked.

Figure 11: Demonstrating the level of detail setting

While simply leaving the level of detail at 1, and benefiting from the performance improvements within the viewer might often be enough, in some situations it is desirable to manually set the level of detail to a desired value. Eden-Tracelab offers the ability to set the level to any value smaller or equal to 100. When the performance constraints forbid further incrementing the level of detail above a certain value, it

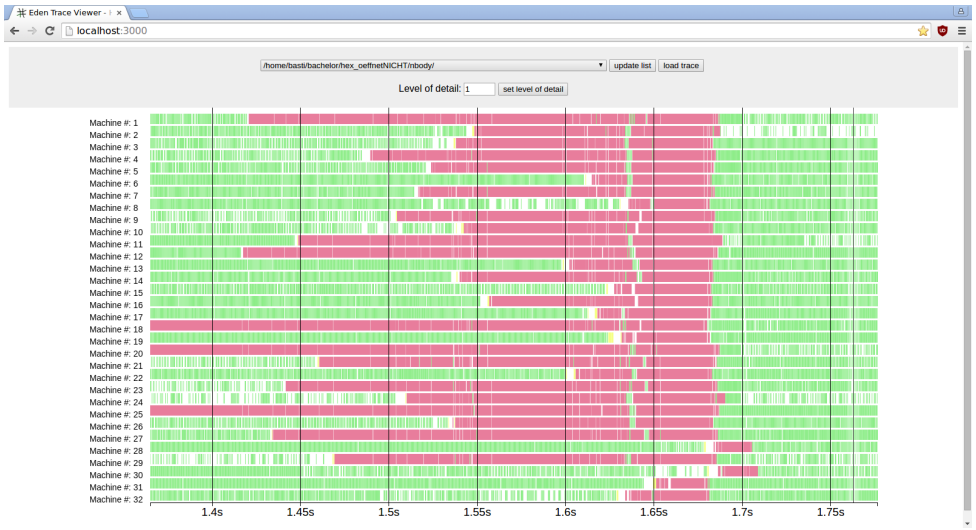


Figure 12: Zooming in to display smaller events. This is the same trace as in Figure 11, but note that the level of detail is still at 1.

is still possible to investigate sections of the trace by zooming into them, and revealing previously unseen events. Figure 11 shows how to inspect a trace visualization by increasing the level of detail, while figure 12 shows the same trace after zooming on a previously white section, without previously incrementing the level of detail.

3.4.3 Interface Behavior

To make the interaction with the interface a smooth experience, and to keep the number of requests generated as small as possible, the following behavior was developed: Whenever a request is executed, the interface is locked, and the user cannot provide any further gestures. A loading animation is displayed until the request is returned and handled. This is also true when the request occurred not due to a panning or zooming gesture, but because the user loaded a new trace, or changed the level of detail. This ensures that even when interacting with large traces at high levels of detail, the interface will never lag behind a large number of requests bogging down the database. Because with this rule alone the interface would freeze with every zooming gesture and thus provide a very uncomfortable user experience an additional rule to ensure a more friction-free experience was implemented: When an input gesture occurs, there is no immediate request send. Instead, a timer is started - If in the next 500ms no additional request is generated, the request is send out and the event data is refreshed. If another event does occur during these 500ms, the timer is reset. This ensures that the user can chain arbitrary zooming and panning gestures to achieve a desired view, without the interface freezing out. Only then the desired data is being loaded. Regardless of whether a request is generated or not, the interface is redrawn on every zooming event, with the available data. This ensures that when a user zooms in on a region, she can still see the less granular data of the lower zoom level before the new data is loaded.

```
1        timer = null
2
3        reload = () ->
4            lock_ui()
```

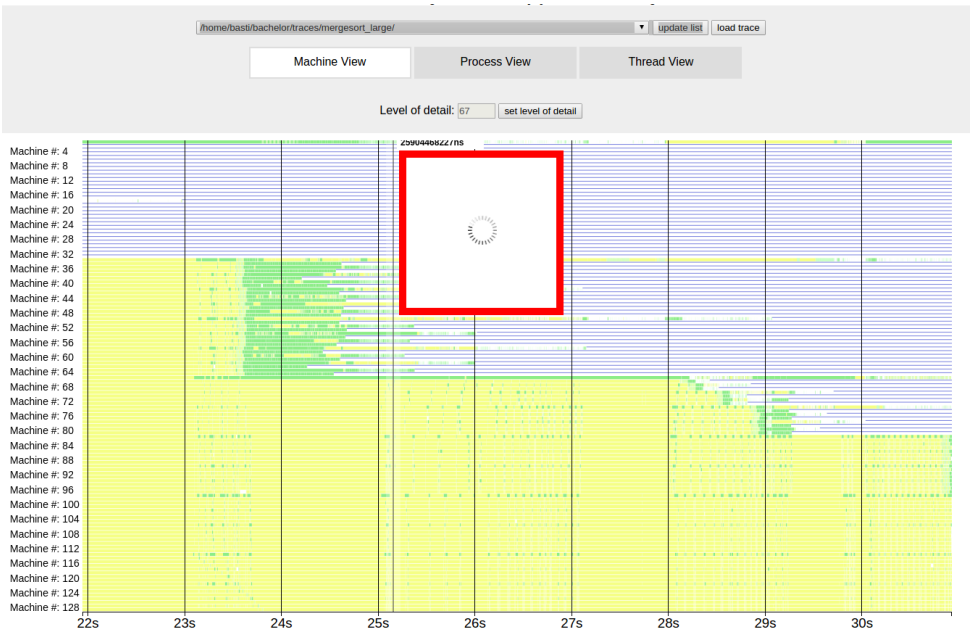


Figure 13: The UI is blocked when waiting for a request to complete.

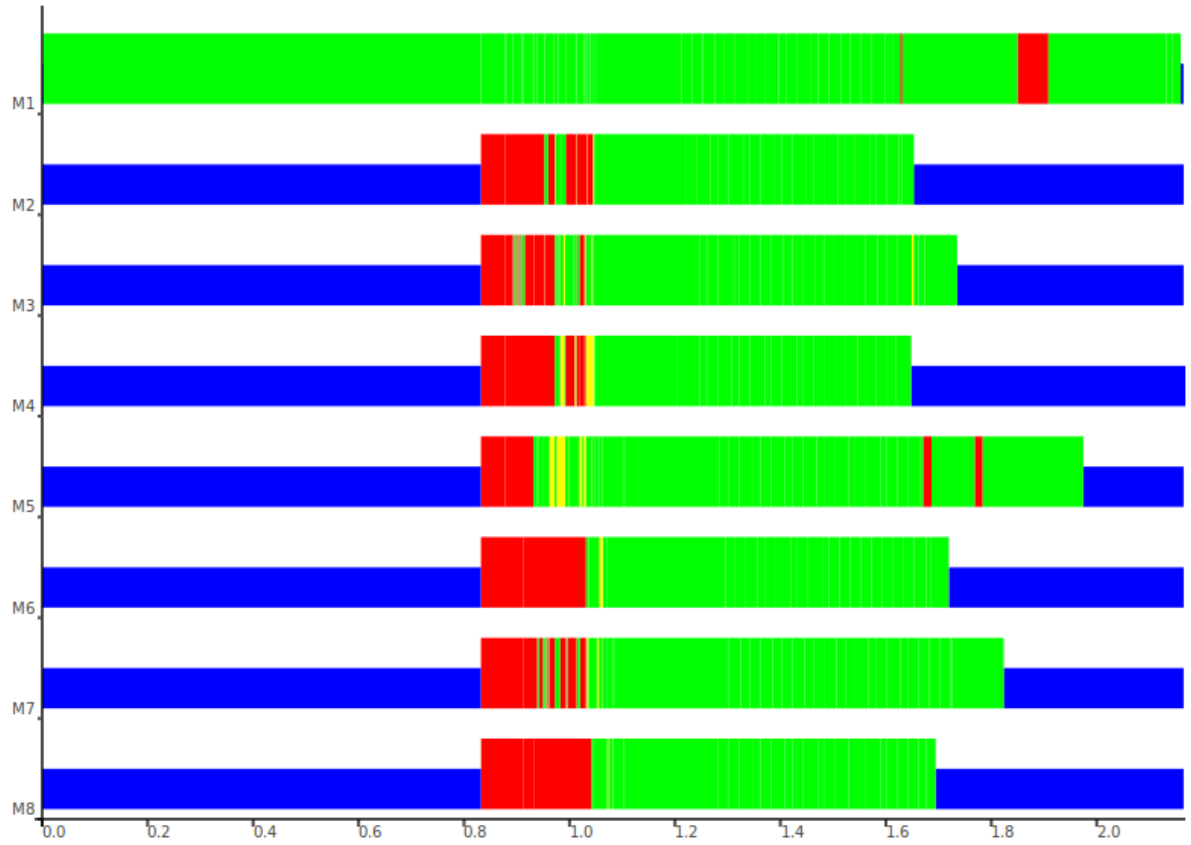
```
        domain = x.domain()
        params =
            id    : trace_metadata.id
            start : Math.floor domain[0]
            end   : Math.floor domain[1]
            minduration : calculate_minimum_duration(domain[0],domain[1])
        $.post("/mevents", params, (data, status) ->
                    if status != "success"
                        alert "failed to load machine events."
                        unlock_ui()
                        return
                    mevents = data
                    draw()
                    unlock_ui()
        )

    zoomHandler = () ->
        if timer != null
            clearTimeout(timer)
        domain = x.domain()
        x.domain(domain)
        xAxisContainer.call(xAxis)
        if ui_locked
            return
        #get the new minimum and maximum x-coordinates.
        timer = setTimeout(reload, ZOOM_TIMEOUT)
        draw()
        return
```

Listing 40: Implementation of the UI timeout

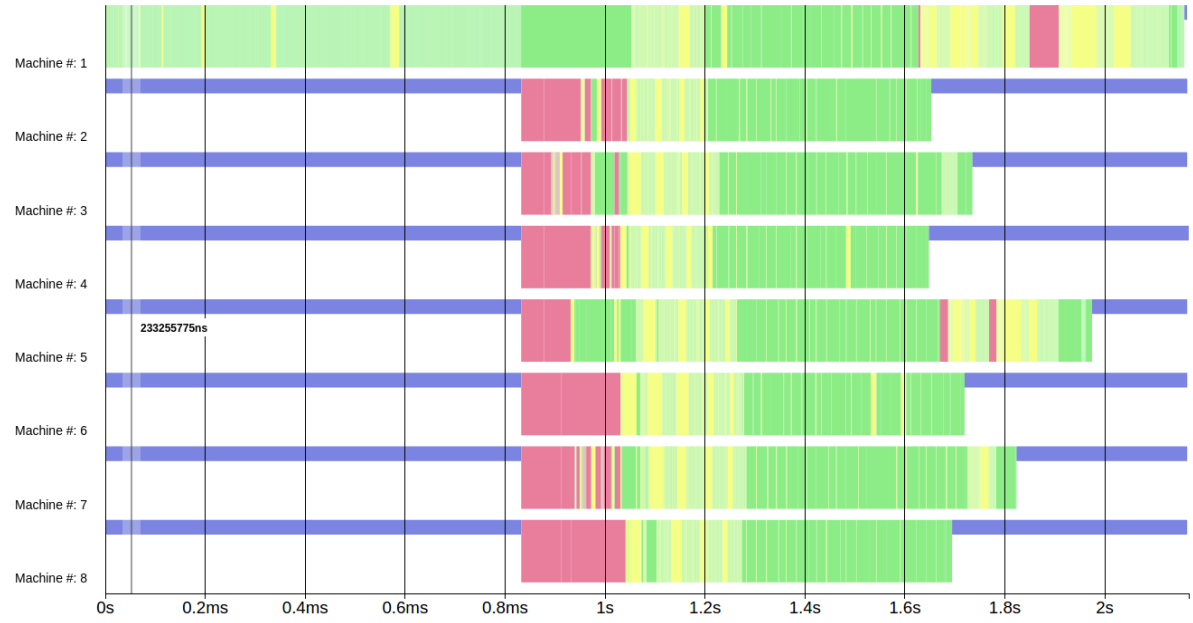
4 Comparison with EdenTV

This section takes an example trace, and shows the differences and similarities between EdenTV and Eden-Tracelab. Figure 14 shows the machine view of the trace generated by a mergesort algorithm run on eight cores. The overall structure of both traces looks similar: The creation date and lifetime of the machines are identical in both images. There also seems to be a complete match between the sections that both tools consider to be blocked. Yet, some variation exists: Eden-Tracelab considers some sections as `Runnable` that EdenTV displays as `Running`. The reason for this probably lies in the parser code of Eden-Tracelab: Different versions of the GHC runtime encode the `blockReason` field in the `StopThread` event differently (This was not part of the original specification, and was caused by a bug in a specific GHC version). Eden-Tracelab currently only parses events created by the GHC-7.8.2 runtime correctly (which is not the current version of GHC, but the version of GHC that the current version of Eden is based on.) Earlier and later versions of GHC will have different encodings of the `StopReason`, and therefore Eden-Tracelab will interpret these reasons wrongly.

Another difference between EdenTV and Eden-Tracelab is that EdenTV does not allow for arbitrarily deep zooming, while Eden-Tracelab was specifically developed to include this feature. Figure 17 illustrates this feature by showing five screenshots of the same trace at different zoom level, revealing more and more detail.

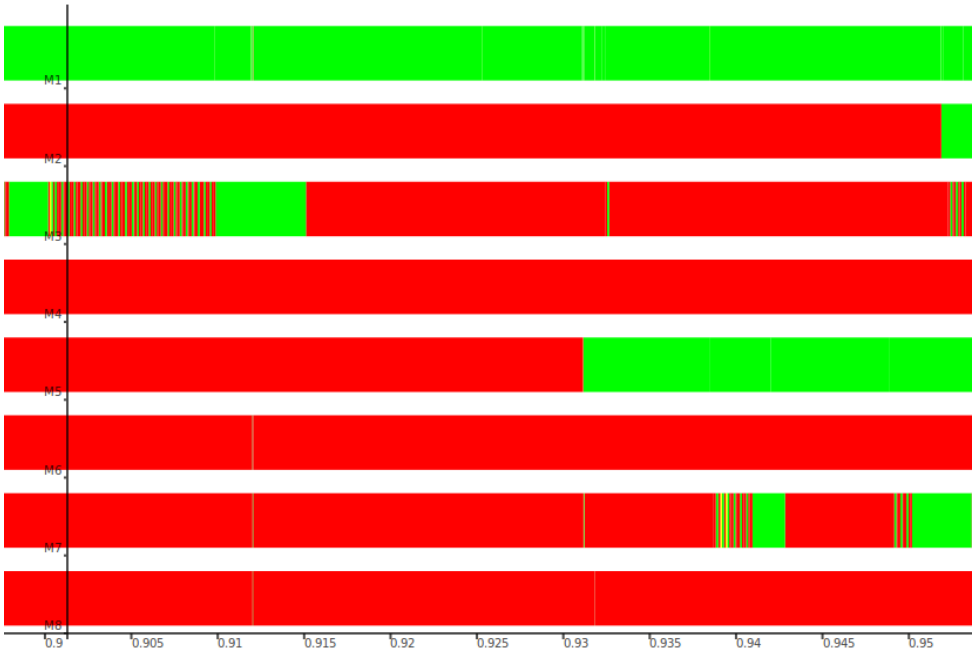


(a) The mergesort trace in EdenTV



(b) The mergesort trace Eden-Tracelab

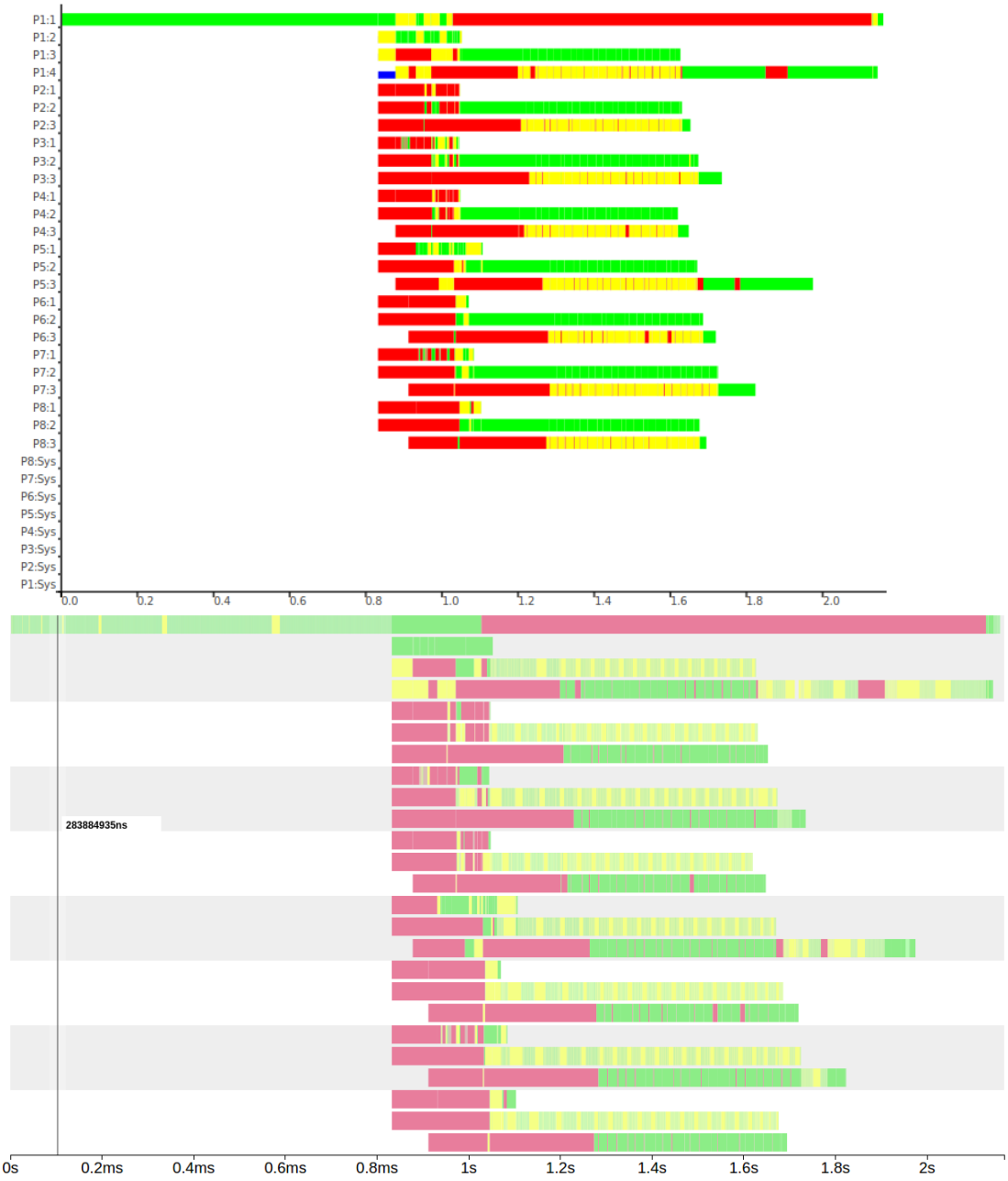
Figure 14: The same trace as seen in EdenTV and Eden-Tracelab



(a) EdenTV zoomed into the interval between 0.9s and 0.95s

(b) The same view in Eden-Tracelab

Figure 15: The same trace as seen in EdenTV and Eden-Tracelab



(a) Processes belonging to the same machine will be grouped, and the background of those groups is shaded to tell them apart more easily

Figure 16: Comparison of process views.

Figure 17: Eden-Tracelab allows for arbitrarily deep zooming.

5 Future Work

While all the stated goals of this thesis have been met by the provided implementation, the following provides an overview of possible future extensions and improvements to Eden-Tracelab.

Missing views

First and foremost, EdenTV still has more features than Eden-Tracelab. Due to the time constraints of this thesis, the full feature set could not be reproduced. The two main missing elements are the thread view, and inclusion of messages. Both would provide an increase in the research opportunities of Eden-Tracelab. The thread view provides the programmer with more fine-grained information on the state of her processes, which might help in identifying performance problems. The current version of Eden-Tracelab offers the ability to precisely ascertain the state of a given process or machine at any point in time. However, the communication in between those processes often provides the context to interpret that information - knowing that a process is blocked at a given point in time is not as useful a piece of information as knowing why its blocked.

Level of detail

On low level of detail settings the program often encounters the following problem: Rather large areas of the trace are simply not colored in, due to the fact that those areas contain only events shorter than the threshold set by the level of detail. To counteract this, while still not retrieving every single event from the database, an analysis feature for detecting those areas and averaging could be developed, similar to ghc-events-analyze , which was developed by de Vries et al.[5]

Performance

Analyzing traces with Eden-Tracelab is still rather time intensive: The traces that were analyzed for demonstration purposes in this thesis showed that analyzing a trace and transferring the results into the database took place at a rate between 15-20 MB/minute of trace file data. While this is an acceptable pace for smaller traces it means that a trace of one gigabyte would take about an hour to be processed.

GUI Behavior

Currently Eden-Tracelab reloads all the events displayed during the current time window if a zoom event is registered, or if the level of detail is changed. While this is fine for demonstration purposes, future versions should be able to filter out unnecessary events, and then only reload the ones not yet in memory. For example, after a zoom-in has occurred, Eden-Tracelab should filter out all the events that would be rendered off screen and only then load the ones that are smaller than those that would be displayed in the previous zoom setting.

6 Conclusion

The tool developed in this thesis addresses and solves some of the major issues encountered in the original EdenTV. While the original EdenTV is bound by the amount of RAM available on the system when opening larger files, Eden-Tracelab is not. Due to its iterative parsing approach, Eden-Tracelab is executed with a fixed memory footprint, regardless of file size. While EdenTV limits the maximum zoom level through its fixed pixel buffer, Eden-Tracelab is able to zoom into a trace arbitrarily deep. This is accomplished by re-drawing the current view on every zoom event. Eden-Tracelab was developed to have a clear and extensible interface. There is potential for future researchers to add additional analysis options to the interface: By extending the web service with additional API calls, and using the visualization functions of D3.js, new analysis options can be added with reasonable effort. This thesis also doubles as a reference and documentation for the implementation of Eden-Tracelab.

While this thesis accomplishes its stated goal of creating a proof-of-concept implementation for an alternative trace-visualization tool that can handle arbitrarily large files, it does not provide all features available within EdenTV: So far, only two of the three views have been implemented, and there is no feature to analyze message passing between machines. EdenTV remains the more powerful analysis tool in these aspects. However, in its current state Eden-Tracelab already provides a good basis for implementing these and other features. Thus it is plausible that feature parity with EdenTV can be reached, without losing any of the additional benefits of Eden-Tracelab.

References

[1] The Eden Team, "Eden trace viewer documentation." `http://www.mathematik.uni-marburg.de/~eden/?content=trace_doku&navi=trace`, 2012.

[2] B. Struckmeyer, "Implementierung eines werkzeugs zur visualisierung und analyse paralleler programmläufe in haskell," Master's thesis, Philipps-Universität Marburg, 2006.

[3] J. Berthold and R. Loogen, "Visualizing parallel functional program runs: Case studies with the eden trace viewer," *Parallel Computing: Architectures, Algorithms and Applications. Advances in Parallel Computing*, vol. 15, pp. 121–128, 2007.

[4] D. Jones Jr, S. Marlow, and S. Singh, "Parallel performance tuning for haskell," in *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pp. 81–92, ACM, 2009.

[5] D. C. Edsko de Vries, "Performance profiling with ghc-events-analyze." `http://http://www.well-typed.com/blog/2014/02/ghc-events-analyze/`, 2014.

[6] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí, "Parallel functional programming in eden," *Journal of Functional Programming*, vol. 15, no. 03, pp. 431–475, 2005.

[7] "Ghc wiki - comments/rts/scheduler." `://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Scheduler`, 2015(updated).

[8] S. Marlow, S. Peyton Jones, and S. Singh, "Runtime support for multicore haskell," in *ACM Sigplan Notices*, vol. 44, pp. 65–78, ACM, 2009.

[9] The GHC Team, "Ghc users guide - 4.17 running a compiled program." `http://http://downloads.haskell.org/~ghc/latest/docs/html/users_guide/`, 2015(updated).

[10] D. J. Satnam Singh, Simon Marlow, "The glasgow haskell compiler wiki - eventlog-ghc." `https://ghc.haskell.org/trac/ghc/wiki/EventLog`, 2015(updated).

[11] B. O'Sullivan, "attoparsec: Fast combinator parsing for bytestrings and text." `https://hackage.haskell.org/package/attoparsec`, 2015(updated).

[12] "Hspec: A testing framework for haskell." `https://hspec.github.io`, 2015(updated).

[13] E. Kmett, "Lenses, folds and traversals." `http://lens.github.io`, 2012.

[14] L. P. S. Brian O'Sullivan, "postgresql-simple: Mid-level postgresql client library." `https://hackage.haskell.org/package/postgresql-simple`, 2015(updated).

[15] B. O'Sullivan, "aeson." `https://github.com/bos/aeson`, 2015(updated).

[16] J. Ashkenas, "Coffeescript." `http://coffeescript.org`, 2015(updated).

[17] M. Bostock, "D3.js - data driven documents." `http://d3js.org`, 2015(updated).

[18] The jQuery Foundation, "jquery." `https://jquery.org`, 2015(updated).

[19] "scotty: Haskell web framework inspired by ruby's sinatra, using wai and warp." https://hackage.haskell.org/package/scotty, 2015(updated).

Deutsche Zusammenfassung

EdenTV stellt wirksame Analysemöglichkeiten für die Performance-optimierung von in Eden geschriebenen parallelen funktionalen Programmen zur Verfügung. Dazu gehören die Möglichkeit, die Zustände von Maschinen, Prozessen und Threads eines Programms und die Nachrichtenkommunikation zwischen Prozessen zu visualisieren. Dennoch besitzt EdenTV einige Einschränkungen, die die Möglichkeiten der Traceanalyse beeinträchtigen. So kann EdenTV nur Tracedateien bis zu einer gewissen Dateigröße öffnen, begrenzt durch den zur Verfügung stehenden Arbeitsspeicher. Außerdem zeichnet EdenTV die Trace-Visualisierung in einen Pixelbuffer mit fester Größe, der beim zoomen lediglich skaliert wird. Daher gibt es eine untere Schranke für die Länge der in EdenTV darstellbaren Events. Im Laufe dieser Arbeit wurde das Programm Eden-Tracelab zur Traceanalyse entwickelt, das diese Mängel nicht enthält.

Diese Arbeit dokumentiert das Design und die Implementierung von Eden-Tracelab, und die dabei verwendeten Technologien und Algorithmen. Anders als EdenTV ist Eden-Tracelab nicht als monolithische Anwendung implementiert, sondern in einer Server-Client Architektur als Webanwendung. Eden-Tracelab kann in drei Module aufgeteilt werden: Den Parser/Eventprozessor, den Webservice, und den HTML5 Viewer. Aufgabe des Parser/Eventprozessors ist das Parsen der binären `*.eventlog`-files, und das Generieren von Events höherer Ordnung, die in der Datenbank abgelegt und im HTML5 Viewer dargestellt werden können. Zum Parsen wurde ein iterativer Ansatz gewählt, so dass der Parser zu jedem Zeitpunkt nur einige wenige Events im Arbeitsspeicher halten muss. Dazu wurde ein Parser für das eventlog-Format mithilfe der Parserbibliothek attoparsec verfasst. Der Eventprozessor nimmt die vom Parser in chronologischer Reihenfolge ausgelesenen Events entgegen, und modifiziert damit ein internes Modell des Eden Runtime Systems. Im Laufe dieses Prozesses werden Events höherer Ordnung generiert, die anschließend in die Datenbank geschrieben werden, mit Hilfe eines Datenbankmoduls das mit der Datenbankbibliothek postgresql-simple implementiert wurde. Der Webservice, welcher auf dem Web-Framework Scotty basiert, dient als Schnittstelle zwischen der postgres-Datenbank und dem HTML5 Viewer. Der HTML5 Viewer wurde in CoffeeScript entwickelt, unter Zuhilfenahme der Javascriptbibliotheken jQuery und D3.js.

Zum Zeitpunkt der Abgabe dieser Arbeit sind zwei der in EdenTV enthaltenen Traceansichten erfolgreich implementiert: Die Maschinen- und Prozessansicht. Die beiden Entwicklungsziele wurden erreicht: Eden-Tracelab kann sowohl Dateien beliebiger Größe öffnen (nur beschränkt durch den für die Datenbank zur Verfügung stehenden Festplattenspeicher), als auch beliebige Vergrößerungsstufen von Traces darstellen.