

TERM PROJECT

COAST Framework 64-Bit

Philipp Schönenberg, Patrik Wenger

supervised by
Marcel Huber, Prof. Peter Sommerlad

Spring 2016

Abstract

The IFS maintains a C++ web application framework called COAST. It was developed for 32-bit hardware platforms and features an extensive collection of test suites. Our main task consists of:

1. Adding support for 64-bit platforms, while ensuring correct functionality of at least a predefined set of core tests.
2. Measure the performance differences for each of the ported test suites and explain the causes.

The optional goals include porting additional test suites, creating a history of performance measurements over time, improving one of the internal data structures, as well as preparing the code base for modern C++.

The students approached this project by informing themselves about the difference between 32-bit and 64-bit platforms and common porting issues. Subsequently, the affected tests were fixed iteratively. Several performance measurement utilities, ranging from the simple `time` command to more sophisticated solutions from the *perf* and *Valgrind* tool suites, have been evaluated and applied appropriate.

Although easy to fix, the causes for most of the broken tests were hard to track down. They consisted of data type discrepancies, disabled warnings, and a race condition, most of which resulted from the assumption that `long` is always a 32-bit quantity. All tests from both the mandatory and the optional goals, and the example application *CoastRecipes* work. On 64-bit, where applications now have access to the full potential of 64-bit systems, including a much higher memory limit, the memory footprint is roughly 1.5× the one on 32-bit. However, the speed-wise performance differences are insignificant. The performance measurement scripts that have been contributed can be used to selectively (e.g. a certain set of test suites) obtain differences between 32-bit and 64-bit, as well as a historical record of performance characteristics.

Contents

1	Scope	1
	Goals	1
	Optional Goals	2
I	Management Summary	3
1.1	Initial Situation	4
1.2	Software Development Process	4
1.3	Project Phases	4
1.3.1	Inception	4
1.3.2	Elaboration	5
1.3.3	Construction	5
1.3.4	Transition	6
1.4	Results	6
II	Technical Report	7
2	Context	8
2.1	Initial Situation	8
2.2	Software Architecture	9
2.2.1	Basic Layer	10
2.2.2	System Layer	11
2.2.3	Communication Layer	11
2.2.4	Multi-Threading Blocks	11
2.3	Problem Description	11
2.3.1	Non-Functional Requirements	12
3	About 32/64-Bit	13
3.1	History	13
3.2	Advantages	13
3.3	Disadvantages	14
3.4	Motivation: Why 64-bit?	14
3.5	Data Models	14
4	64-Bit Port	16
4.1	Concept	16
4.2	Common Issues	17
4.2.1	Disabled Warnings	17
4.2.2	Dynamic Size Types for Fixed Size Needs	17
4.2.3	Pointer Arithmetic	18
4.2.4	Alignment	18
4.2.5	Numeric Constants	18
4.2.6	Storing Integers in <code>double</code>	19

4.2.7	Storing Pointers in <code>int</code>	19
4.2.8	Bitwise Shifting	19
4.2.9	Timestamps	19
4.2.10	Integer Overflow	20
4.3	Verification of 64-bit runtime	21
4.3.1	Simple <code>int</code> Array	21
4.3.2	Using <code>Anything</code>	22
4.4	Implementation	23
4.4.1	<code>CoastFoundationBaseTest</code>	23
4.4.2	<code>CoastStorageTest</code>	24
4.4.3	<code>CoastCompressionTest</code>	25
4.4.4	<code>CoastRegexTest</code>	25
5	Performance Measurements	26
5.1	Legacy	26
5.2	Concept	26
5.2.1	<code>time</code>	27
5.2.2	<code>gprof</code>	27
5.2.3	<code>oprofile</code>	27
5.2.4	<code>perf</code>	27
5.2.5	<code>Valgrind</code>	28
5.3	Implementation	28
5.3.1	Architecture	28
5.3.2	First Script	29
5.4	Results and Analysis	31
5.4.1	Test Environment	31
5.4.2	<code>time</code>	31
5.4.3	<code>perf</code>	32
5.4.4	<code>Valgrind</code>	36
5.5	Conclusion	42
6	Optional Goal: Performance History	44
6.1	Use Case #1: Analyzing a Range of Revisions	44
6.1.1	Specifying Commit Ranges	44
6.1.2	Interpolating Results	45
6.2	Use Case #2: Analyzing a List of Revisions	45
6.3	Use Case #3: Archiving the Results	46
6.4	Non-Functional Requirements	46
6.5	Concept	46
6.5.1	File Format	46
6.5.2	Applicability	47
6.6	Implementation	47
6.6.1	Using Current Script Revisions for Old Code	47
6.7	Usage	48
6.7.1	Selecting the Measurement Method	48
6.7.2	Selecting the Test Suites	49
6.7.3	Selecting the Processor Architecture	50
6.7.4	Example of Use Case #1	50
6.7.5	Example of Use Case #2	51
6.7.6	About Use Case #3	51
7	Optional Goal: Migrating Further Tests	52
7.1	Concept	52
7.2	Implementation	53
7.2.1	<code>CoastSecurityTest</code>	53
7.2.2	<code>CoastQueueingTest</code>	55

7.2.3	CoastRendererTest	55
8	Optional Goal: C++11/14 Support	59
8.1	Analysis	59
8.2	About <code>std::auto_ptr</code>	59
8.2.1	Migrating to <code>std::unique_ptr</code>	60
8.2.2	Reducing Verbosity	60
8.3	A More Transparent Alternative to Preprocessor Switches	61
8.3.1	Detailed Solution	62
8.4	Removing Obsolete Information	64
8.4.1	<i>PC-Lint</i> Magic Comments	64
8.4.2	The <code>register</code> Keyword	64
9	Optional Goal: Improving <code>Anything</code> Internals	65
10	Conclusion	66
III	Appendix	67
A	Self Reflection	68
A.1	Thank You	68
B	Formalities	69
B.1	Declaration of Originality	69
B.2	Permissions	70
C	Project Plan	71
C.1	Organization	71
D	Infrastructural Problems	72
D.1	Redmine: MySQL driver	72
D.2	Software Versions on VM: Upgrade	72
D.3	Redmine Bugs and Another Upgrade	72
D.3.1	Buggy Pre-Installed Version	72
D.3.2	Gantt Charts	73
D.4	SSH Access to VM	73
D.5	Mails from Redmine	73
D.6	Cevelop	74
D.7	Jenkins	74
E	perf stat diff	75
E.1	CoastEBCDICTest	75
E.2	CoastFoundationAnythingOptionalTest	76
E.3	CoastFoundationBaseTest	77
E.4	CoastFoundationIOTest	77
E.5	CoastFoundationMiscellaneousTest	78
E.6	CoastFoundationPerfTest	79
E.7	CoastFoundationTest	80
E.8	CoastFoundationTimeTest	80
E.9	CoastMTFoundationTest	81
E.10	CoastRegexTest	82
E.11	CoastStorageTest	83
E.12	CoastSystemFunctionsTest	83
F	COAST Setup Cookbook	84
F.1	Getting Started	84

F.1.1	About <i>Boost</i>	84
F.1.2	Installing Dependencies	84
F.1.3	Cloning COAST	85
F.1.4	Further Dependencies	86
F.1.5	Example Webapp: <i>CoastRecipes</i>	86
F.1.6	Trouble Shooting	86
F.2	Development	87
F.2.1	Headless	87
F.2.2	Getting and Running Cevalop	87
F.2.3	Running Test Suites	88
F.2.4	Trouble Shooting	88
G	Usage of perf/perf-history	89
H	Usage of perf/with_*	91

List of Figures

2.1	COAST's architectural layers	9
4.1	GZIP file format	25
5.1	Massif output CoastSystemFunctions32	37
5.2	Massif output CoastSystemFunctions64	38
5.3	Massif output CoastBaseTest32	42
5.4	Massif output CoastBaseTest64	42

List of Tables

3.1	Data models in comparison	15
4.1	Unix millennium bug illustration	20
5.1	Valgrind Heap Summary	40

Listings

2.1	Example of an <code>Anything</code> configuration file	10
4.1	Excerpt from <code>REBitSetTest::GeneratePosixSet</code>	17
4.2	Test case to test memory limit using <code>int</code> array	21
4.3	Test case to test memory limit using an <code>Anything</code> vector	22
4.4	Test case <code>PoolAllocatorTest::ExcessTrackerEltGetSizeToPowerOf2Test</code> . .	24
4.5	Method <code>ul_long ExcessTrackerElt::GetSizeToPowerOf2(size_t)</code>	24
5.1	Example CSV file produced by <code>with_time</code> script	30
5.2	Diff: <code>time</code> -based performance measurement of <code>CoastFoundationPerfTest</code>	31
5.3	Word diff: <code>time</code> -based performance measurement of <code>CoastFoundationPerfTest</code> . .	32
5.4	Example output of <code>perf stat</code>	32
5.5	Simple <code>perf diff</code> of <code>CoastFoundationPerfTest</code>	35
5.6	Detailed <code>perf diff</code> of <code>CoastFoundationPerfTest</code>	35
5.7	<i>Boost</i> 's Small Object Optimization temporarily disabled	39
5.8	Allocation optimization of <code>String</code> in <code>coast/foundation/base/ITOString.cpp</code>	40
5.9	Snippet of <code>PoolAllocator</code> 's ctor	41
5.10	Cause for a spike in memory consumption in 64-bit	41
6.1	Example performance history result (CSV) (<code>perf-stat</code>)	50
7.1	Overview of <code>CoastSecurityTest</code> 's test suites and their status	53
7.2	COAST's <code>MD5Context</code> API	53
7.3	<i>OpenSSL</i> 's MD5 API	54
7.4	Fix for <code>coast/modules/Security/Blowfish.h</code>	54
7.5	Broken method in <code>coast/foundation/io/Resolver.cpp</code>	56
8.1	Preprocessor switch to decide between <code>std::unique_ptr</code> and <code>std::auto_ptr</code> . .	62
8.2	New file <code>coast/foundation/base/boost_or_std/type_traits.h</code>	62
8.3	New file <code>coast/foundation/base/boost_or_std/memory.h</code>	63
E.1	Diff: <code>perf stat</code> of <code>CoastEBCDICTest</code>	75
E.2	Diff: <code>perf stat</code> of <code>CoastEBCDICTest</code> (#2)	76
E.3	Diff: <code>perf stat</code> of <code>CoastFoundationAnythingOptionalTest</code>	76
E.4	Diff: <code>perf stat</code> of <code>CoastFoundationBaseTest</code>	77
E.5	Diff: <code>perf stat</code> of <code>CoastFoundationIOTest</code>	77
E.6	Diff: <code>perf stat</code> of <code>CoastFoundationMiscellaneousTest</code>	78
E.7	Diff: <code>perf stat</code> of <code>CoastFoundationMiscellaneousTest</code> (#2)	78
E.8	Diff: <code>perf stat</code> of <code>CoastFoundationPerfTest</code>	79
E.9	Diff: <code>perf stat</code> of <code>CoastFoundationPerfTest</code> (#2)	79
E.10	Diff: <code>perf stat</code> of <code>CoastFoundationTest</code>	80
E.11	Diff: <code>perf stat</code> of <code>CoastFoundationTimeTest</code>	80
E.12	Diff: <code>perf stat</code> of <code>CoastFoundationTimeTest</code> (#2)	81
E.13	Diff: <code>perf stat</code> of <code>CoastMTFoundationTest</code>	81
E.14	Diff: <code>perf stat</code> of <code>CoastRegexTest</code>	82
E.15	Diff: <code>perf stat</code> of <code>CoastRegexTest</code> (#2)	82
E.16	Diff: <code>perf stat</code> of <code>CoastStorageTest</code>	83
E.17	Diff: <code>perf stat</code> of <code>CoastSystemFunctionsTest</code>	83
G.1	Usage of <code>perf-history</code> script	89
H.1	Usage of <code>with_*</code> performance measurement scripts	91

Chapter 1

Scope

In brief, this term project consists of the migration of the COAST framework from 32-bit to 64-bit, measuring the difference in performance, as well as documenting the process.

The following two sections contain the mandatory and optional goals we formally agreed upon.

Goals

These are mandatory:

- The framework shall compile and work on both 32-bit and 64-bit platforms.
- Existing test suites¹ shall be examined regarding 32-bit data types and migrated accordingly.
 - At least the following test suites must compile and pass on both 32-bit and 64-bit:
 - * CoastFoundationAnythingOptionalTest
 - * CoastFoundationBaseTest
 - * CoastFoundationIOTest
 - * CoastFoundationMiscellaneousTest
 - * CoastFoundationPerfTest
 - * CoastFoundationTimeTest
 - * CoastMTFoundationTest
 - * CoastCompressTest
 - * CoastStorageTest
 - * CoastSystemFunctionsTest
 - * CoastRegexTest
- For the aforementioned test suites, the following properties of the migration from 32-bit to 64-bit shall be documented:
 - Differences in performance
 - Differences in memory usage

¹A list of existing test suites can be printed using `scons -u --showtargets`

- The methodology shall be documented properly.
- A howto will be written which documents how to setup the COAST framework on a common Linux environment and, e.g., how to run the CoastFoundationBaseTest test suite.

Optional Goals

At least one of the following has to be achieved:

- Access functions of the `Anything` class are based on the 32-bit `long int` data type. With the migration to 64-bit, the question arises whether to distinguish between 32/64-bit access functions as well as the internal data representation (`long int` \rightarrow `long long int`). Solutions as to how to deal with this situation shall be suggested and, if possible, the best one shall be implemented.
- The framework shall compile using the settings `boost` (default, `scons --use-lang-features=boost` meaning `-std=gnu++03`) and `-std=c++0x` (`scons --use-lang-features=c++0x`). However, it has to be ensured that it keeps working with the default setting `boost`. Any necessary changes need to be documented or implemented in the code using `#ifdef` blocks.
- It shall be possible for the performance measurement results to be:
 - archived as a history
 - created retrospectively

The performance measurement results used for this can, but don't have to come from `CoastFoundationPerfTest`.

- The following test suites are to be migrated in the order they appear:
 - *CoastWDBaseTest*
 - *CoastSecurityTest*
 - *CoastHTTPTest*
 - *CoastSSLTest*
 - *CoastPerfTest*
 - *CoastPerfTestTest*
 - *CoastFunctionalActionsTest*
 - *CoastWorkerPoolManagerTest*
 - *CoastQueueingTest*
 - *CoastNTLMAuthTest*
 - *CoastAccessControlTest*
 - *CoastHTMLRenderersTest*
 - *CoastActionsTest*
 - *CoastRendererTest*
 - *CoastAppLogTest*
 - *CoastStdDataAccessTest*
 - *CoastStringRenderersTest*
 - *CoastDataAccessTest*

Part I

Management Summary

1.1 Initial Situation

The IFS has been maintaining the COAST² framework since 2005. It is a flexible, highly configurable framework written in C++ and is still used by a business customer, where it's deployed as a frontend web server. The supported platforms include Linux and Solaris.

At the time being, it was implemented in the *gnu++98* dialect of C++, meaning some modern features known in *C++11* and newer were not used. Furthermore, the code base was written with only 32-bit hardware in mind, meaning some of the functions require the data types `int` and `long` to be 32-bit quantities.

It has an extensive test suite which can be used to verify correct functioning. This is how we knew for sure that COAST wouldn't work properly when compiled and ran on a 64-bit system.

1.2 Software Development Process

RUP³ is used to plan and manage this term project. It's an iterative process flexible enough for this kind of project. It's taught as part of the *Software Engineering* courses at HSR and is thus considered the primary candidate for the software development process used for this project.

Another candidate was Scrum, which we decided against as it's only useful for projects with developer teams of three to nine people.

1.3 Project Phases

RUP splits a project life-cycle into four phases, namely *Inception*, *Elaboration*, *Construction*, and *Transition*. In the next sections, we'll briefly elucidate what we did during each phase.

1.3.1 Inception

This phase mainly only consisted of the kickoff meeting, which took place on February 25th, 2016.

With great help from Marcel Huber, we found our bearings within the COAST framework's code base. We set up the basic environment and got the COAST framework to compile on our personal computers, which made us confident enough to proceed with it. This is a central part of RUP's *Inception* phase.

We also clarified a few things on the more administrative side like our availability hours, where the time tracking takes place, that we'll be using L^AT_EX to write the documentation.

After the meeting, we immediately started working on the documentation and on refining the term project goals, which already concludes this phase.

²<https://coast-project.org>

³Rational Unified Process

Marcel Huber ordered an instance of the *Software Engineering 2* VM for development purposes (like Redmine and Jenkins), which was delivered practically over night.

1.3.2 Elaboration

As suggested by RUP, we sketched out a rough schedule during this phase. This schedule consisted of five milestones and one week reserve:

1. Elaboration (due March 13th)
2. Construction: 64-bit Port (due April 3rd)
3. Construction: Performance Measurements (due April 24th)
4. Construction: Optional Goals (due May 15th)
5. Transition (due May 22nd)

As can be seen, we conveniently used some of RUP's terminology to name the mile stones of this term project.

By the end of this phase, we completed refining the term project goals and did most of the groundwork for the project, which included:

- preparing the VM
- setting up Redmine
- getting COAST to compile on the VM (32-bit only, of course)
- creating the 2 Git repositories needed, namely:
 - COAST with 64-bit support, hooked into Redmine, linked to the provided *Gerrit* installation
 - COAST-independent files like this documentation
- setting up the Jenkins build server
- outlining the parts, chapters, and sections of this document

During this phase, we ran into many problems around the VM, Redmine, and Jenkins. They're all documented in detail in the Problems chapter starting on page 72.

To prepare for the next phase, we also compiled COAST for 64-bit, run the extensive test suite, and recorded each of the visible issues on the Redmine platform.

So far the general architecture of COAST and our development setup proved to be stable, so we were all set to commence with the *Construction* phase. This insight and the resulting decision are, again, a central part of RUP's *Elaboration* phase.

1.3.3 Construction

As shown in the aforementioned milestones, we divided the *Construction* phase across three separate milestones, in each of which our engineering methodology of the more technical side will be required.

1.3.3.1 64-bit Port

For this milestone, we worked on the aforementioned issues created on the Redmine platform, one by one, to find the root cause, decide on an appropriate fix, implement it, and document it clearly.

Technically adept readers can find said documentation in the corresponding chapter of the Technical Report on page 16.

1.3.3.2 Performance Measurements

The first of the two main tasks of this milestone was to come up with an appropriate method for measuring the performance for each of the changed tests of the test suite, to find and analyze any differences in both speed and memory usage.

Once we had a reliable way of getting the numbers, we were able to measure the performance and document our findings. The detailed results of that can be found on page 26.

1.3.3.3 Optional Goals

All of the additional test suites of the optional goal have been migrated. The documentation can be found on page 52.

An infrastructure to create a history of performance measurements has been added. This can be done using any of the supported measurement methods, some simple, some more sophisticated. Typical values of interest are extracted from the performance measurement results and combined into a CSV for further processing and visualization, e.g. in a spreadsheet application. It's easily extensible to adapt it to other needs, such as extracting additional values from the results, or adding new methods to measure performance.

1.3.4 Transition

The last project phase was mainly planned to be documentation and administrative work, as well as holding the last meeting. As some of our last administrative duties, we wrote the abstract and designed an A0 poster.

1.4 Results

The results of this term project are:

- mandatory and optional COAST modules work correctly in 64-bit mode
- performance differences of fixed modules analyzed and explained
- script infrastructure to analyze performance differences of revisions over time
- COAST compiles on C++11 and C++14
- this document

Part II

Technical Report

Chapter 2

Context

2.1 Initial Situation

COAST¹, the C++ Open Application Server Toolkit, was developed in the late 1990s out of the need for a general purpose web application framework. Reasons for developing something new included Apache's poor API and poor performance back then. Also, reasons not to use Java were its lack of acceptance as well as its poor performance.²

It is based on earlier work done by André Weinand and Erich Gamma at Ubilab.

The goals for it were to

- have some general purpose application infrastructure,
- allow for simple introspection,
- provide a base for framework mechanisms, and to
- have minimal external dependencies.

Its strengths are:

- Simplicity
 - a simple design, no code and interface bloat
 - small server footprint
- Configurability
 - dynamic reconfigurability without stopping the server, even DLL module loading
- Multi-Threading for high-efficiency
 - Thread specific memory pools which allow for dynamic memory (no OS-level synchronization required)
 - low latency
- Unit tests

¹<https://coast-project.org>

²The information about both Apache and Java back then has been taken from the *Advanced Patterns and Frameworks* lesson about COAST.

It features client protocols such as HTTP, HTTPS, and FTP, can access data from databases like MySQL and LDAP servers, and even through SMTP. The supported output formats include HTML and XML.

It's supported on at least Linux and Sun Solaris, but it should be fairly easy to get it to run on any POSIX compliant system.

Originally developed in the *gnu++98* dialect of C++, modern features known from *C++11* such as *move semantics* were not used. The same goes for *templates* which, although available, were a little understood concept of C++ back then. Even compilers used to have lots of problems with *templates*.

At the time being, no one knew what 64-bit hardware was going to look like, and thus it was developed with only 32-bit in mind. This means that, although it would compile for 64-bit, much of its functionality depended on 32-bit specific data types. That it fails to work properly on 64-bit can be seen when running the test suite.

Since 2005 it's been maintained and further developed by the IFS.

2.2 Software Architecture

As shown in Figure 2.1, COAST features a layered architecture. This reduces the overhead for simple applications, increases flexibility, and makes it easy to extend.

Below is a brief overview about COAST's lower building blocks up to and including the multi-threading block. The higher levels are of less significance regarding this term project.

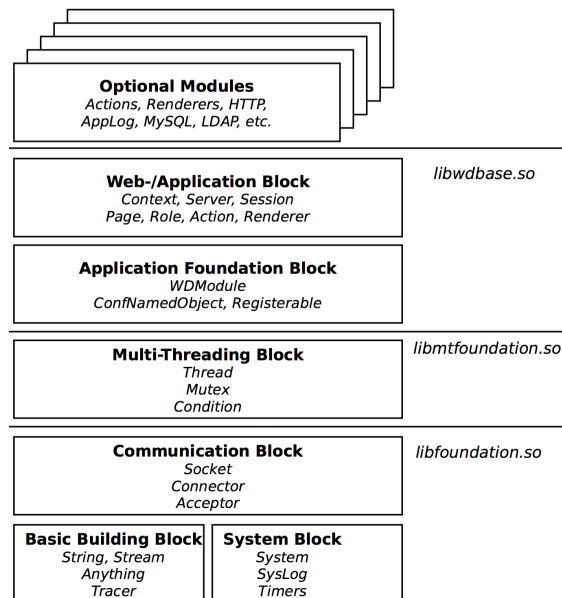


Figure 2.1: COAST's architectural layers

2.2.1 Basic Layer

2.2.1.1 Storage

For basic storage management, there is a static API similar to `malloc` / `free` known from C. It's implemented using an allocator, where `coast::storage::Global()` returns the global allocator instance and `coast::storage::Current()` the thread-specific one for the current thread.

The class `Allocator` implements a wrapper for the C API, whereas `PoolAllocator` implements an allocator for thread-specific storage using a bucket strategy.

2.2.1.2 String

`String` is a generic and safe character container. It's memory safe for text and binary data.

2.2.1.3 Streaming

As a basic I/O streaming mechanism, there's `StringStream` which uses a `String` as buffer, and `SocketStream`, which is used for network communication.

2.2.1.4 Anything

The universal, self-describing data container `Anything` is used throughout the COAST framework and features the ability to be read from and written to (configuration) files. It uses the *PImpl* idiom to implement concrete types such as `long`, `double`, `char*`, `String`. It also combines array and hashtable like behavior (associative access), and supports deep cloning and boasts various comparison methods.

It comes with its own memory management to allow for scalability on multi-processor and multi-core systems. It is not thread-safe due to its implementation being based on reference counting. It therefore must not be used across thread boundaries.

Its primary use is to represent possibly hierarchical configuration information, as well as being used as a flexible internal data structure for `TmpStore`, `RoleStore`, and `SessionStore` in higher levels.

Example of a `.any` file:

```
{
  /Text {
    /First Chapter {
      /Paragraph1 "As you can see this file has been stored as an Anything."
      /Paragraph2 "Now it is read into the sessionstore."
    }
    /Second Chapter {
      /Paragraph1 "Instead of this action you can test any existing action."
    }
  }
  /Data {
    /somedata { 12 43 56 90 34 }
    /anydata { 90 98 54 22 44 }
    /Port 55
  }
}
```

Listing 2.1: Example of an `Anything` configuration file

2.2.2 System Layer

OS wrappers for streams, filesystem operations, processes environment, date / time, timers, and Syslog support reside here.

2.2.3 Communication Layer

The `Socket` class provides access to socket level APIs, including polling mechanisms. Sockets connections also provide `iostream` operations.

2.2.3.1 Connector

The `Connector` class can be used to establish a connection to an endpoint, and thus represents the active side. The resulting socket stream can be read from and written to.

`Acceptor` is the passive counterpart to the `Connector` and listens for incoming connection requests. It uses an `AcceptorCallback` to act on said requests.

2.2.4 Multi-Threading Blocks

2.2.4.1 Thread

A `Thread` is an abstraction decoupled from the OS used for starting, running, and stopping threads. It features state semantics to reliably synchronize with itself and its clients. It supports thread-local storage and the use of pooled memory. Hook methods can be used for state transition handling.

2.2.4.2 Mutex

Next to basic locking and unlocking functionality, the `Mutex` class can be used as a scope guard and also allows recursive locking with a small bookkeeping overhead.

2.2.4.3 Condition

A `Condition` can unlock the associated mutex when waiting for an event to happen, and automatically lock said mutex when signaled. Timed waits are also possible.

Methods to signal a single or all (broadcast) waiters are available too.

2.2.4.4 ThreadPools

The `ThreadPools` class can be used to manage a pool of the same `Thread` objects to create, initialize, run, join, terminate, and delete them. Threads can either get dispatched for a small piece of work to be done, or get their workload on their own indefinitely.

2.3 Problem Description

The focus of this term project is the migration of the COAST framework from 32-bit to 64-bit, as opposed to being merely a feasibility analysis. Furthermore, the migration's effects on performance (speed-wise and memory usage-wise) shall be examined and documented. The necessary

methods to measure performance will first have to be conceived.

2.3.1 Non-Functional Requirements

The only NFRs that apply for this project are:

- it must still be possible to compile COAST on C++03 (setting *gnu++03*)
- it must support Linux
- test quality must not decrease
- the *CoastRecipes* app still has to work

Chapter 3

About 32/64-Bit

For the unfamiliar reader, below are a few brief sections on the history and comparison between 32-bit and 64-bit systems.

3.1 History

From the mid 1980s up until the early 2000s, 32-bit processors were the norm. This means those processors internally used 32-bit wide integer registers, 32-bit wide address buses and 32-bit wide data buses, which means only up to 4 GiB of physical memory could be directly addressed. Because of several other architectural limitations of mainstream computers such as 16-bit wide ALUs or some of the buses actually being narrower than 32 bits, and the development of Physical Address Extension, that memory limit did not require swift action.

Given the constant rise of memory needs and the falling prices, 32-bit addresses soon became the bottleneck, as a single process wasn't able to use more than 4 GiB of memory without some tricky hacks, even though the whole system was.

With the mainstream adoption of 64-bit processors in the early 2000s, that memory limit was lifted to a much higher amount of 16 EiB of theoretically addressable memory. In practice, it is still less, such as 48 bits or 52 bits effectively used to address virtual and physical memory, respectively [10, 4-Kbyte Page Translation, p. 132].

As of 2016, virtually all personal computer (stationary and mobile), as well as server processors use a 64-bit architecture, even extending to some smartphones. More precisely, they run applications as well as kernels in 64-bit mode, which wasn't usually the case for years.

As the ubiquitous x86-64 architecture was designed as an extension to the x86 instruction set, most 32-bit applications will run on such 64-bit hardware without any modifications. However, they won't be able to make use of the full set of features available on 64-bit hardware.

3.2 Advantages

Hardware capable of running 64-bit applications offers more than just a larger address space. Other features include:

- Some programs can greatly benefit from the wider registers (such as encoding, decoding, and encryption software).
- The x86-64 architecture supports more general-purpose registers, which means there's a higher chance that a program's data can fit and stay inside the processor's registers rather than being stored to and fetched from cache or even main memory later, which can result in significant performance improvements, especially when the program contains tight loops.
- The portions of address space reserved by some operating systems is much smaller in relativity to the whole address space.
- Memory mapped files over 4 GiB are much easier to implement on 64-bit hardware.

3.3 Disadvantages

The main disadvantage of 64-bit hardware is that the same amount of data will most likely take up more memory space due to longer data types such as pointers, and memory alignment padding. This is because of the different data models being used. This matter is further discussed below.

If a program can't benefit from an address space larger than 4 GiB, or use the larger data types and greater number of registers to its advantage, it's probably not worth the effort to port it (in case it had been designed specifically for 32-bit hardware).

That being said, it is, of course, still possible to develop a program that will compile on both 32 and 64-bit hardware by using data types such as `int`, `long` and fixed-size `int32_t`, `int64_t` where appropriate, and thus exploit the best of both worlds (danger of premature optimization).

3.4 Motivation: Why 64-bit?

The main motivation behind porting the COAST framework to 64-bit is to add proper support for modern 64-bit hardware. The primary intent of this port is not a performance improvement, as COAST's main purpose is not the encoding/decoding/encryption of huge amounts of data. A potential performance gain of COAST's crypto subsystem would be considered merely an added benefit.

3.5 Data Models

The C++ standard merely specifies in [5, Fundamental types, §3.9.1/2, p. 71] that `sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long) ≤ sizeof(long long)`. Due to this lax specification different data models are being used. Win32 and 32-bit Unix-like systems use the so-called *ILP32* data model. *ILP32* means that `int`, `long`, and pointers are 32-bit quantities.

On 64-bit hardware, Unix-like systems (including Linux, Solaris, OS X, BSD) use the *LP64* model, whereas Microsoft Windows uses the *LLP64* data model. *LLP64* denotes that only `long long` and pointers are 64 bit each, whereas within the *LP64* data model even `long` is a 64-bit quantity. See Table 3.1 for a comparison, which also shows the typical memory alignment requirements for each data type.

Data type	ILP32 size	ILP32 alignment	LP64 size	LP64 alignment
char	1 byte	1 byte	1 byte	1 byte
short	2 bytes	2 bytes	2 bytes	2 bytes
int	4 bytes	4 bytes	4 bytes	4 bytes
long	4 bytes	4 bytes	8 bytes	8 bytes
pointer	4 bytes	4 bytes	8 bytes	8 bytes
size_t	4 bytes	4 bytes	8 bytes	8 bytes
long long	8 bytes	4 bytes	8 bytes	8 bytes

Table 3.1: Data models in comparison

Chapter 4

64-Bit Port

4.1 Concept

COAST has a good set of unit tests to begin with and we are exploiting this fact. Without proper unit tests we would have to write our own tests to find and fix 64-bit issues. Our approach is to switch to `archbits=64` and run the tests to find issues and fix them. The following test suites worked out of the box in 64-bit mode:

- *CoastFoundationAnythingOptionalTest*
- *CoastFoundationIOTest*
- *CoastFoundationMiscellaneousTest*
- *CoastFoundationPerfTest*
- *CoastFoundationTest*
- *CoastFoundationTimeTest*
- *CoastMTFoundationTest*
- *CoastSystemFunctionsTest*

Of the tests that we're required to fix, the following used to fail on 64-bit:

- *CoastCompressionTest*
- *CoastFoundationBaseTest*
- *CoastRegexTest*
- *CoastStorageTest*

How we fixed these is documented under the section 4.4 Implementation. As mentioned in the Management Summary on page 4 already, we created an issue on the Redmine platform for each of the failing test suites during the *Elaboration* phase so we were all set to fix them during the *Construction* phase.

We also considered the online resources [16], [15], [19], [2] to get an overview of what to expect and back up our decisions. What follows is a list of what seem to be the most common issues

when porting source code from 32-bit to 64-bit, along with examples directly from the COAST code base, if possible.

4.2 Common Issues

Why can't we just tell the compiler to compile for 64-bit and be done with it? Here we'll explain some of the typical issues to expect when migrating (or *porting*) software from 32-bit to 64-bit.

4.2.1 Disabled Warnings

As mentioned on [16, Off-warnings], disabled warnings are a very common reason for portability issues. Actually this was the case in `CoastStorageTest`, as you can see later in subsection 4.4.2 on page 24.

The source line

```
return (1 << lMaxBit); //lint !e647
```

contains a portability issue (more on that later). Notice the comment. The documentation¹ of the static code analyzer *PC-Lint* states that this exact number causes truncation warnings to be suppressed, which confirms the fact that this is a very common issue.

4.2.2 Dynamic Size Types for Fixed Size Needs

Sometimes the source code specifies a certain type, probably to get a certain bit-size, but the type used actually doesn't have a fixed size. Like `unsigned long` instead of `uint32_t`.

Consider this example from COAST:

```
cppfile << std::endl;
cppfile << "const unsigned long _dummy_" << pcName << " [] = {" << std::endl;
    << endl;
cppfile << s << "};" << std::endl;
cppfile << "const REBitSet " << pcName << "(_dummy_" << pcName << ");" << endl;
    << std::endl;
```

Listing 4.1: Excerpt from `REBitSetTest::GeneratePosixSet`

The correct type in this case would have been `const uint32_t`.

C++ introduced the header `cstdint`, which defines fixed width integers like `uint32_t` with C++11, see [9]. COAST needs to work with older standards, so we had to use the C header `stdint.h`.

¹<http://www.gimpel-online.com/MsgRef.html>

4.2.3 Pointer Arithmetic

For both pointer arithmetic and array indexes, it used to be possible to use `int` on *ILP32*. But in the *LP64* data model, it's recommended to use `std::ptrdiff_t` instead. This way, destructive wrap arounds that could happen when looping through an array if the data type used for the index is too small (e.g. `unsigned short` or `unsigned int`), can be avoided.

However, none of the tests we fixed suffered from this issue.

4.2.4 Alignment

Problematic is code that makes assumptions about the actual location or size of objects in memory. As can be seen in Table 3.1, not only the data type sizes, but also the memory alignment restrictions differ between memory models.

Important is the following quote from the C++ standard [5, Class Members, §9.2/12, p. 219]:

“Nonstatic data members of a (non-union) class with the same access control (Clause 11) are allocated so that later members have higher addresses within a class object. The order of allocation of non-static data members with different access control is unspecified (Clause 11).”

This means that the compiler is generally not allowed to reorder the placement of data members in memory. However, it is still allowed to insert unnamed data members, so called *padding*. This is to ensure that the alignment requirements of the particular members can be met.

The performance of a data structure specifically crafted for high-performance on a system implementing the *ILP32* model might perform worse on *LP64* because of different alignment restrictions.

4.2.5 Numeric Constants

Numeric constants, sometimes called *magic numbers*, can cause porting issues, because they assume a certain type size. E.g. if on 32-bit originally all bits of an `unsigned long` would have been set using the numeric constant `0xFFFFFFFFUL`, only half of the bits would be set on 64-bit.

Instead, doing

```
unsigned int all_set = std::numeric_limits<unsigned long>::max();
```

or even

```
unsigned int all_set = -1UL;
```

would be more portable.

We found some instances in COAST, where numeric constants were interpreted as `int` or explicitly as `long` (e.g. `0L`) that caused some issues on 64-bit. See: section 4.4 and chapter 7 for more information.

4.2.6 Storing Integers in `double`

Apparently, some people used to do that, because a `double` (64-bit) uses 52 bit for the fraction, and only the remaining 12 bit for the exponent, which means that, in the *ILP32* data model, it's possible to fit a `long` (either `signed` or `unsigned`) into a `double`. However, because a `long` in the *LP64* data model is a 64-bit quantity, this is no longer possible.

We haven't found anything like this in COAST. But still, it's a common issue, apparently.

4.2.7 Storing Pointers in `int`

In the *ILP32* data model, it's possible to store a pointer into an `int`, both of which are 32-bit quantities. Of course, this isn't possible in the *LP64* data model where pointers are 64-bit quantities and `int` is still a 32-bit quantity. If, for some reason, one needs to treat pointers as integers, one should just use `intptr_t` or `uintptr_t`, respectively.

We haven't found anything like this in COAST.

4.2.8 Bitwise Shifting

Take the following code from the COAST codebase as an example:

```
return (1 << lMaxBit);
```

As long as the value of the variable `lMaxBit` is less than 32, everything goes well. But in case it's greater, like it happened to be in COAST's `CoastStorageTest` when compiled for 64-bit, it results in *undefined behavior*.

That's because `lMaxBit` is actually the exponent to calculate the next power of two greater than the value passed as a `long`, which can be up to 64 in the *LP64* data model, whereas the integer literal `1` is of type `signed int`, which is only a 32-bit quantity, even in the *LP64* data model. So when shifting to the left, which effectively means $1 * 2^{lMaxBit}$, and the resulting value can't be represented in said type (`signed int`), it's *undefined behavior*.

To cite the C++ standard on [5, Shift operators, p. 120]:

“The value of `E1 << E2` is $E1$ left-shifted $E2$ bit positions; vacated bits are zero-filled. If $E1$ has an unsigned type, the value of the result is $E1 * 2^{E2}$, reduced modulo one more than the maximum value representable in the result type. Otherwise, if $E1$ has a signed type and non-negative value, and $E1 * 2^{E2}$ is representable in the corresponding unsigned type of the result type, then that value, converted to the result type, is the resulting value; otherwise, the behavior is undefined.”

For more information about the `CoastStorageTest`, see subsection 4.4.2 on page 24.

4.2.9 Timestamps

On Unix-like operating systems, time is defined as the number of seconds since January 1st, 1970. To avoid hardware dependent assembler code in Unix, K&R invented the C program-

UTC Time	Decimal	Bytes
2038-01-19 03:14:07	+2147483647	0111'1111 1111'1111 1111'1111 1111'1111
2038-01-19 03:14:08	-2147483648	1000'0000 0000'0000 0000'0000 0000'0000

Table 4.1: Unix millennium bug illustration

ming language, which most of Unix' codebase was written in. To represent those timestamps, C uses the `time_t` type. According to the C standard [4, p. 388], the actual type of `time_t` is "implementation-defined". C++ inherited `time_t`, but C++11 introduced a type safe and more sophisticated alternative, the *chrono* library [8].

Linux and BSDs use a `signed long` to implement `time_t`. As shown in the Table 3.1, a `long` on 32-bit systems is only 32 bit wide. Table 4.1 illustrates a possible behavior (called *wrap around*, see the subsection 4.2.10 below) when the maximum value of the timestamp is reached, which will happen in January 2038.

Linux assumes that, as of 2038, most desktop and server systems will be 64-bit and thus the issue will no longer be of any significance. For embedded systems, they are going to add a new interface for 64-bit timestamps on 32-bit systems as explained on [17], which will be backwards compatible.

However, OpenBSD and FreeBSD [23] went the more radical path and broke backwards compatibility by implementing `time_t` as a `long long` type.

As a side note, in case COAST will be used on Linux 32-bit systems in 2038 and later, it might suffer from the Unix millennium bug.

4.2.10 Integer Overflow

The avid reader realizes that, very closely related to the issue in the previous section, an application might depend on 32-bit behavior when it comes to integer overflows. To quote the C++ standard regarding unsigned integer arithmetic on [5, Fundamental types, §3.9.1/4, p. 72]:

"Unsigned integers shall obey the laws of arithmetic *modulo* 2^n where n is the number of bits in the value representation of that particular size of integer.⁴⁸"

And the footnote:

"48) This implies that unsigned arithmetic does not overflow because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting unsigned integer type."

As for signed integer arithmetic, to quote the C++ standard again on [5, Expressions, §5/4, p. 84]:

"If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined."

Bottom line is, when porting a 32-bit application to 64-bit, one has to make sure it doesn't rely on *undefined behavior*, and also doesn't rely on 32-bit `unsigned` overflow behavior (*wrap around*), which, although *defined behavior*, behaves very differently with 64-bit integers.

4.3 Verification of 64-bit runtime

To make sure we're actually running in 64-bit mode, we created two tests. Both try to verify that allocating more than 4 GiB of virtual memory will succeed.

Both tests were performed on a headless Ubuntu Linux VM running on OSX. The command `free -m` reported a total of 7983 MiB main memory, and swap has been turned off for this test. See here:

```
$ free -m
```

	total	used	free	shared	buffers	cached
Mem:	7983	216	7767	8	24	
92						
-/+ buffers/cache:		99	7884			
Swap:	1019	0	1019			

4.3.1 Simple `int` Array

The newly added test looks like this:

```
#include <cstring> // for std::memset

void SystemBaseTest::VerifyAllocationTest() {
    int8_t* chunk = new int8_t[2147483647U]; // 2 GiB - 1 byte
    std::memset(chunk, 0xAA, sizeof(int8_t[2147483647U]));
    delete[] chunk;
}
```

Listing 4.2: Test case to test memory limit using `int` array

The number 2147483647U is exactly 2 GiB - 1 byte (so 0b01111111111111111111111111111111). This works in both 32-bit and 64-bit. On 64-bit, it also works when 4 GiB (4294967296 bytes) or more is used. However, using an array length of 2147483648U or more (≥ 2 GiB) on 32-bit results in the following error:

```
coast/foundation/base/Test/SystemBaseTest.cpp: In member function 'void ↵
↳ SystemBaseTest::VerifyAllocationTest()':
coast/foundation/base/Test/SystemBaseTest.cpp:32:51: error: overflow in array ↵
↳ dimension
    std::memset(chunk, 0xAA, sizeof(int8_t[2147483648U]));
                                   ^
coast/foundation/base/Test/SystemBaseTest.cpp:32:51: error: size of array 'type ↵
↳ name' is too large
```

4.3.1.1 Multiple Allocations

It should be noted that going back to the allocation that worked under 32-bit (2 GiB - 1 byte) and adding another allocation of 1.5 GiB — to arrive at a total of 3.5 GiB of allocated memory for this test — worked as well. But it's definitely impossible to pass 4 GiB on 32-bit.

4.3.2 Using Anything

A more realistic test was performed by creating as many `Anything` objects as possible, where each is initialized with a freshly allocated `String` of 1 MiB. Here's the test:

```
#include <iostream> // for std::cout
#include <vector>

void SystemBaseTest::VerifyAllocationWithAnythingTest() {
    std::vector<Anything*> v = std::vector<Anything*>();

    try {
        while(true) {
            const String* const s = new String(1024*1024);
            v.push_back(new Anything(s->cstr()));
            std::cerr << "Successful String allocations: " << v.size() << "\n";
        }
    } catch(std::bad_alloc e) {
        for (std::vector<Anything*>::iterator it = v.begin(); it != v.end(); ++it) {
            delete *it;
        }
        std::cerr << "All Strings deallocated.\n";
    }
}
```

Listing 4.3: Test case to test memory limit using an `Anything` vector

There was a catch to this: To be able to actually catch a `std::bad_alloc`, a `throw` statement had to be added in the `String::alloc(long)` function. Otherwise it would just log the error but not actually throw an exception or exit immediately.

4.3.2.1 Result on 32-bit

See the shortened output of the test:

```
[...]
Successful String allocations: 4057
Successful String allocations: 4058
Successful String allocations: 4059
FATAL: GlobalAllocator::Alloc malloc of sz:1048608 failed. I will crash :-(
ERROR: String::alloc: Memory allocation failed!
All Strings deallocated.
```

This shows that it fails to allocate any more `String` objects shortly before reaching 4 GiB (of course the process has allocated memory outside of this test as well).

Using `stderr` instead of `stdout` made sure we get, according to Unix conventions, unbuffered — and thus correctly ordered — output. When using `stdout`, it tends to look like it keeps allocating after the first failed allocation, which of course isn't the case.

4.3.2.2 Result on 64-bit

The following output proves that our test has been able to allocate more than 4 GiB of virtual memory in 64-bit. With $7658 * 1$ MiB allocations it came fairly close to the 7837 MiB of free main memory reported by `free -m`, definitely passing the 4 GiB mark.

```
[...]
```

```

Successful String allocations: 7656
Successful String allocations: 7657
Successful String allocations: 7658
/home/paddor/SA/coast/tests/CoastFoundationBaseTest/scripts/Linux_glibc_2.9-↵
↳ x86_64-64_debug/CoastFoundationBaseTest.sh: line 223: 2737 Killed ↵
↳ "$CMD" "$@"
scons: building terminated because of errors.

```

As can be seen, the process wouldn't even get a chance to cleanup. It gets killed by the Linux *Out Of Memory killer* (normally referred to as the *OOM killer*), as can be confirmed by taking a look at the syslog:

```

May  5 14:50:25 alpha SysLogTest[22956]: Message as error
May  5 14:50:31 alpha kernel: [130591.339369] CoastFoundation invoked oom-killer↵
↳ : gfp_mask=0x201da, order=0, oom_score_adj=0
May  5 14:50:31 alpha kernel: [130591.339373] CoastFoundation cpuset=session-14.↵
↳ scope mems_allowed=0
[...]
```

Of course the *OOM killer* acted correctly by identifying and killing the process responsible for the *out of memory* condition, given the test is literally designed to allocate all available memory.

The reason why the kernel entered the *out of memory* condition is this, quoted from [18]:

“By default, Linux follows an optimistic memory allocation strategy. This means that when `malloc()` returns non-NULL there is no guarantee that the memory really is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer.”

According to `man 5 proc` (see [22]), this so-called *heuristic overcommit* behavior can be disabled and instead a more pessimistic approach can be chosen by running `echo 2 /proc/sys/vm/overcommit_memory` as root. This will cause Linux to always check whether the requested memory is actually available and never overcommit. Using this setting, the test terminates gracefully on 64-bit as well:

```

Successful String allocations: 4609
Successful String allocations: 4610
Successful String allocations: 4611
FATAL: GlobalAllocator::Alloc malloc of sz:1048624 failed. I will crash :-(
ERROR: String::alloc: Memory allocation failed!
All Strings deallocated.

```

Strangely though, it happens much earlier, although still well north of the 4 GiB mark. The reasons to this are outside of the scope of this document.

4.4 Implementation

This section covers in detail how the broken tests were fixed.

4.4.1 CoastFoundationBaseTest

The first failing test was hard to find but easy to fix. In `AnythingParserTest::parseSimpleTypeLong` two assertions failed. The test tried to read the string of a long numeric value into an `Anything`, after which the behavior of reading this string into a `long` variable was tested. Whereas in *ILP32* the size of a `long` variable is 32 bits, this failed as expected. In contrast, with *LP64*'s

`long` being a 64-bit quantity, the test was able to store the value into a `long` variable without the desired overflow. We fixed this test by creating the numeric value as follows:

```
"1" + std::numeric_limits<long>::max();
```

This way we are even ready for more exotic platforms, where a `long` is neither 32 bit nor 64 bit wide, paving the way for the coming 128-bit port in a century.

The hard part was to search in the right place. After studying [19] and [15], we only looked for typical problems in porting to 64-bit, but forgot that the test might just be plain wrong. But it was a good lecture and opportunity to put our nose into *CoastFoundationBase* and the testing framework in general.

4.4.2 CoastStorageTest

Consider the following test:

```
void PoolAllocatorTest::ExcessTrackerEltGetSizeToPowerOf2Test()
{
    StartTrace(PoolAllocatorTest.ExcessTrackerEltGetSizeToPowerOf2Test);
    ExcessTrackerElt aDfltCtor;
    // [...]
    assertEquals(2147483648UL, aDfltCtor.GetSizeToPowerOf2(1073741825UL), "↵
    ↵ expected correct size");
}
```

Listing 4.4: Test case `PoolAllocatorTest::ExcessTrackerEltGetSizeToPowerOf2Test`

And the method being tested:

```
ul_long ExcessTrackerElt::GetSizeToPowerOf2(size_t ulWishSize)
{
    long lBitCnt = 0L, lMaxBit = 0L;
    while ( ulWishSize > 0 ) {
        ++lMaxBit;
        // count bits to see if the wish size is already an exact power ↵
        ↵ of 2
        lBitCnt += ( ulWishSize & 0x01 );
        ulWishSize >>= 1;
    }
    // adjust bitcount if wish size is already an exact power of 2
    if ( lBitCnt == 1 ) {
        --lMaxBit;
    }
    return (1 << lMaxBit); //lint !e647
}
```

Listing 4.5: Method `ul_long ExcessTrackerElt::GetSizeToPowerOf2(size_t)`

Here, the number of bits set in a number passed as a `size_t` parameter are counted. This is needed to deduce the next power of two greater than the given value `ulWishSize`. As you can see, at the end of the method `ExcessTrackerElt::GetSizeToPowerOf2`, the desired value is derived using a single bitwise shift operation.

Unfortunately, the result type in this case is `int` instead of `long`, which means shifting more than 31 bits will result in *undefined behavior* in the *ILP32* data model (since the result type is signed). That's because the integer literal `1` is of type `signed int`, which is a 32-bit quantity

in both the *ILP32* and the *LP64* data models. But `long` is a 64-bit quantity in *LP64*. For the relevant definitions in the C++ standard, see subsection 4.2.8 on page 19.

To fix the test, we changed the line to:

```
return (1L << lMaxBit); //lint !e647
```

By doing this, the integer conversion of the result and thus the *undefined behavior* is avoided, as the result type is changed to the desired `long`.

This slight type discrepancy existed in the 32-bit version of COAST, too, but was insignificant. The reason why is that in the *ILP32* data model, both `int` and `long` are 32-bit quantities.

4.4.3 CoastCompressionTest

This is how virtually all tests in this suite work: Feed the class with some sample data and compare the resulting stream with a binary string. We noticed there are lots of zeros in the actual stream causing the fail. Figure 4.1 shows the GZIP binary format. We had some issues with the gray fields.

MTIME (modification time) is a `time_t`, which is unspecified and *glibc* [25] uses `long`. Serializing a `long` on 64-bit systems results in a 64-bit timestamp, but GZIP **requires** a 32-bit timestamp. For more information about Unix timestamps, see subsection 4.2.9 on page 19.

The field CRC32 has the same problem. GZIP expects 32-bit but a `long` now is 64-bit on *LP64*. CRC16 is not affected because it's already a `short` which is 16 bit on *LP64* as well. See Table 3.1.

8-bit	8-bit	8-bit	8-bit	32-bit	8-bit	8-bit	8-bit	N-bit	8-bit	8-bit	16-bit	N-bit	32-bit	32-bit
ID1	ID2	CM	flags	MTIME	XFL	OS	XLEN	XLEN+	filename	filecomment	CRC16	comp	CRC32	ISIZE

Figure 4.1: GZIP file format

4.4.4 CoastRegexTest

CoastRegexTest suffered from the usual problem that `long` is a 64-bit quantity on in the *LP64* data model (see Table 3.1). The first bunch of tests were easy to fix by changing some `long` to `int32_t`. But then things got exciting.

All tests using regular expressions containing character classes such as `\S` and `\W` were failing. `RECompiler` and `REBitSet` seemed to work fine. A generated file named *REPosixBitSets.h* containing the bit representation of `isalnum`, `isalpha`, etc. stored them as `long[]` arrays, which had to be changed to `int32_t[]` as well. This was tricky to find, as we expected the issue to be in the `BitSet` or `Compiler`.

Chapter 5

Performance Measurements

Part of the mandatory goals is to analyze the performance for each of the modified test suites and illustrate the differences between the 32-bit and the 64-bit builds, as well as finding and explaining their causes.

For the scope of this project, we'll use the term *performance* as any unit allowing us to compare the execution time and memory footprint.

5.1 Legacy

This has been done in a way before, but the results are produced by COAST code itself, particularly the `CoastFoundationPerfTest` test suite. This means there exists a significant dependency on the code itself; only what the test suite has been programmed for will be measured.

Furthermore, this restriction also makes it impossible to use new or otherwise different methods of performance measurement to get performance characteristics of old code, let alone over time (which is one of the optional goals of this term project).

5.2 Concept

Since we'll need an independent way of measuring performance, both for speed and memory consumption, we'll look for simple as well as sophisticated software solutions available to sample an existing process and report statistics on it. The process in question will be any of COAST's existing test suites, as these are well known packages of functionality, which can be run in isolation as many times as needed.

Research on performance measurement methods concluded that there are multiple methods available which seem promising. We've discussed these with Marcel Huber and decided on which ones to give a try. How we integrated them into the COAST repository in the form of reusable scripts is elucidated in the next section.

Below we briefly explain the available measurement methods and our decisions.

5.2.1 `time`

This is a very basic way to get timing and memory statistics. The `time` command can show CPU time spent in user and system mode, real elapsed time (*wall-clock time*), as well as memory usage information such as maximum size of the resident data. The accuracy isn't perfect, as the manpage on Debian/Ubuntu states, because the elapsed time isn't collected atomically with the execution of the program. Furthermore, most of the information shown by `time` is based on the `wait(2)` system call.

The idea is to get a rough impression about any performance differences between the 32-bit and the 64-bit builds, and whether they're consistent and make sense at all.

5.2.2 `gprof`

*GNU Profiler*¹ looked promising but we rejected it because it can't handle *shared object* files. Since COAST consists of many shared objects, it renders `gprof` unusable for this case.

There is an analogous utility called `spprof` which claims to be able to use shared objects, but available documentation is scarce, which made us look for further alternatives.

5.2.3 `oprofile`

Marcel Huber pointed us to `oprofile`, which looked promising.

The problem was that it's an old Master thesis from 2001. While its documentation is considered to be comprehensive, it's also known to be unstable. `perf` emerged as its successor with support from the Linux kernel team itself. Thus, we immediately discarded `oprofile` in favor of `perf`.

5.2.4 `perf`

The `perf` utility for Linux is a modern tool collection used for performance analysis. It gets its information from the *performance counters*, which are a kernel-based subsystem that covers hardware features (like built into the CPU) as well as software features (like software counters and tracepoints).

Compared to software profilers, the hardware based approach allows low-overhead access to detailed performance information. Another great benefit is that no source code adaptation is needed.

`perf` consists of multiple sub tools which are invoked in a `git`-like manner, e.g. `perf stat ...` or `perf record ...` followed by a `perf report`

`perf` was developed and is still used by the Linux kernel [20]. Since then, we found talks from Netflix [13] and Google [7] employees using `perf` as a performance measurement tool. It seems to be good choice compared to the previous alternatives.

¹<https://sourceware.org/binutils/docs/gprof/>

5.2.5 Valgrind

*Valgrind*² is an essential, and truly valuable tool for any C/C++ programmer. *Valgrind* is known to be used as a memory leak detection tool.

From its manpage³:

“Valgrind is a flexible program for debugging and profiling Linux executables. It consists of a core, which provides a synthetic CPU in software, and a series of debugging and profiling tools. The architecture is modular, so that new tools can be created easily and without disturbing the existing structure”

Because of all this instrumentation, it is considered slow, but we will be using its *heap summary* to get an impression of the memory usage.

5.2.5.1 Memory Consumption

For further and detailed investigation, we plan to use `valgrind --tool=massif` which provides detailed information on which function uses how much memory.

Massif can be instructed to sample the process in intervals of milliseconds (wall-clock), number of instructions, or bytes allocated/deallocated. We plan to instruct *Massif* to use bytes as time-unit because it's a lot easier to compare in short running programs instead of ms or number of instructions. With ms or instructions the shape of the graphs do vary greatly.

We plan to generate graphs using `massif-visualizer`, an interactive visualization for *.massif* files.

5.3 Implementation

5.3.1 Architecture

Several shell scripts written to integrate the various performance measurement methods and to make the functionality easily accessible will be discussed in this chapter. To get a general idea what to expect, they are:

- *perf/with_massif*
- *perf/with_perf*
- *perf/with_perf_stat*
- *perf/with_time*
- *perf/with_valgrind*

As you can see, these shell scripts all reside in the newly created *perf* directory within COAST's repository. Furthermore, since they share common functionality, this shared functionality resides in the library script *perf/lib/measure_utils.sh*.

²<http://valgrind.org>

³<http://linux.die.net/man/1/valgrind>

5.3.1.1 Naming

The name of the scripts has been chosen look similar to plain Unix commands, meaning we didn't include a certain filename extension. This has the advantage that later on, in case the need arises, they can be transparently replaced with other solutions based on Python, C++ or any other language, while their usage (the interface to the user) can stay exactly the same.

5.3.1.2 A Note on Portability

On a side note, all the shell scripts created within the scope of this term project, and discussed in this chapter, are written for the Bourne shell. The Bourne shell is seen as the *lingua franca* of shells on Unix systems. This has the benefit that it'll take little to no effort to make the scripts run on any Unix-like system (e.g. Solaris).

5.3.1.3 Usage

The full usage text (59 lines) can be found in Appendix H. It's the same for all performance measurement scripts discussed in this chapter.

Here's the synopsis from `./with_perf --help`:

```
SYNOPSIS
-----
(1) ./with_perf
(2) ./with_perf TEST...
(3) ./with_perf --all-tests
(4) ./with_perf --diff=TEST
(5) ./with_perf --export [--all-tests | TEST...]
```

By default, any of them will build, run, and measure the set of core test suites for both 64-bit and 32-bit.

Different test suites can be specified as arguments, if necessary. If all possible test suites should be run, `--all-tests` can be passed, which makes the scripts use a `scons` command to dynamically deduce a complete list of all test targets.

Using `--diff=TEST`, a command will be printed. When executed, that command will show you the recorded performance differences between the 32-bit and 64-bit builds for the specified test suite.

The more speed related performance methods (as opposed to memory consumption) will run the test executable once before the actual measurement, as a warmup. By doing so, the next initialization of the test suite should take up as little time as possible, and also more accurately resembles a production environment.

5.3.2 First Script

We wrote a fairly simple shell script `perf/with_time` that will:

1. get list of all COAST tests
2. remember the current timestamp

3. for 32-bit and 64-bit builds, do:
 - (a) for each of the tests mentioned in the mandatory goal, do:
 - i. build test with `--build-cfg=optimized`
 - ii. create CSV with header line if it doesn't exist yet
 - iii. run test 20 times and measure timing using just one `time` command
 - iv. append the results to a CSV file

To be clear, most of this functionality is, of course, reusable and actually resides in the shared library file. Merely the specifics of the `time`-based approach are in the script *with_time*. Thus, only the implementation of this script is explained in detail. The others are virtually the same.

5.3.2.1 Details

The `optimized` build configuration has been chosen because debug information is not needed in the *time*-based approach and, more importantly, to more accurately simulate a production environment.

The relevant `scons` command is essentially:

```
scons --run-force --runparams="-x /usr/bin/time -x '-f' -x '$START_TIME,%U,%S,%e,%P,%M,%t,%D,%p,%W' -x '--output=$TIME_RESULT' -x '--append' -x $NTIMES -x $TIMES -- -all" $TESTNAME >$PERF_DIR/time/${TESTNAME}-${ARCHBITS}_${BUILD_CFG}.log
```

Here's an example of such a resulting CSV file:

```
$ cat perf/perf_results/time/CoastStorageTest-64_optimized.csv
start_time,times_run,user[s],system[s],real[s],cpu,maxres[KB],avgres[KB],
  avgunshared[KB],avgunsharedstack[KB],swaps
2016-04-21 10:47:57,20,2.41,0.21,3.21,98%,45236,0,0,0,0
2016-04-23 09:06:18,20,2.35,0.22,2.61,98%,72772,0,0,0,0
2016-04-23 14:28:01,20,2.30,0.25,2.61,97%,72772,0,0,0,0
2016-04-23 14:51:33,20,2.34,0.22,2.62,97%,72704,0,0,0,0
2016-04-24 00:55:07,20,4.91,0.39,5.40,98%,72720,0,0,0,0
```

Listing 5.1: Example CSV file produced by *with_time* script

This format turns out to be versatile enough to be imported into a statistical software or a spreadsheet application to create diagrams based on the data.

5.3.2.2 Challenges

Unfortunately it initially wasn't that easy to avoid end up measuring either Python's or `dash`'s statistics, as the `time` command ignores child processes created by the main command it is passed. The shell script generated by *SConsider* is needed to prepare the execution environment (change directory, set library related environment variables, ...) before starting the actual test executable, so we couldn't just circumvent that. We needed a way to pass a command just before the actual binary executable. Fortunately though, the option `-x` allows us to do just that.

After integrating the `time` command right before the invocation of the test binary itself, and thus circumventing *SCons*'s overhead, we realized that most of the tests are very quick. Quick, as in a typical test would finish in around 0.01 seconds. That's simply too short for `time` to measure appropriately. Its time granularity just isn't high enough.

Using the `--runparams="-x ... -x ..."` method, it's not straight-forward to run the test binary multiple times, because enclosing it in `for i in `seq 10`; do test_command; done` or using a pipeline like `seq 10 | xargs -n1 -I{} test_command` are both impossible.

We came up with a tiny shell script that does the trick. It's called `ntimes` and does just that: It takes as arguments a number and a command and just runs that command `n` times. It's unique feature is that it doesn't require any pipeline syntax, nor does it require any enclosing syntax. It just needs to be prepended to the command that should be executed multiple times. The variable `$NTIMES` shown above contains its resolved path, whereas `$TIMES` contains an integer like 20.

5.4 Results and Analysis

5.4.1 Test Environment

The test environment is a Lenovo X1 Carbon with 8 GiB RAM, 4xi7-4550U CPU @ 1.50GHz on Ubuntu 15.10 64-bit. To run the tests, we switched to `tty1`, killed X using `sudo systemctl stop lightdm.service` and ran the tests through the respective script.

5.4.2 time

In order to get a general idea, the `time`-based approach was a moderate success.

The results clearly show a consistent difference in memory usage between the 32-bit and 64-bit builds. But the timings are not precise enough for a speed comparison.

Here's an example output of `git diff --no-index4` for one of the tests, namely `CoastFoundationPerfTest`:

```
diff --git a/CoastFoundationPerfTest-32_optimized.csv b/CoastFoundationPerfTest-64_optimized.csv
index 6440020..e968611 100644
--- a/CoastFoundationPerfTest-32_optimized.csv
+++ b/CoastFoundationPerfTest-64_optimized.csv
@@ -1,5 +1,5 @@
  start_time,user[s],system[s],real[s],cpu,maxres[KB],avgres[KB],avgunshared[KB],avgunsharedstack[KB],swaps
-2016-04-23 00:06:18,9.84,3.86,14.25,96%,36616,0,0,0,0
-2016-04-23 00:28:01,10.27,3.92,14.84,95%,37788,0,0,0,0
-2016-04-23 00:51:33,8.99,3.76,13.47,94%,38720,0,0,0,0
-2016-04-23 00:55:07,20.76,7.96,30.42,94%,40640,0,0,0,0
+2016-04-23 00:06:18,6.68,1.49,8.41,97%,56612,0,0,0,0
+2016-04-23 00:28:01,6.68,1.53,9.06,90%,57256,0,0,0,0
+2016-04-23 00:51:33,6.73,1.54,8.68,95%,58748,0,0,0,0
+2016-04-23 00:55:07,14.57,2.92,19.77,88%,61176,0,0,0,0
```

Listing 5.2: Diff: `time`-based performance measurement of `CoastFoundationPerfTest`

⁴`--no-index` allows Git to compare files that aren't known to Git, which is the case for these result files

As you can see, Git recognized all lines as changed and thus there's a bunch of green lines followed by a bunch of red lines. To avoid this behavior and improve readability, a more sophisticated diff can be performed with

```
git diff --no-index --word-diff-regex="[^\s,|,]" \
perf/perf_results/time/CoastFoundationPerfTest-*_optimized.csv
```

which will tell Git what a word in this case consists of, instead of just runs of non-whitespace characters. The output look like this, which would be fairly readable with more space available:

```
diff --git a/CoastFoundationPerfTest-32_optimized.csv b/CoastFoundationPerfTest-64_optimized.csv
index 6440020..e968611 100644
--- a/CoastFoundationPerfTest-32_optimized.csv
+++ b/CoastFoundationPerfTest-64_optimized.csv
@@ -1,5 +1,5 @@
start_time,user[s],system[s],real[s],cpu,maxres[KB],avgres[KB],avgunshared[KB],←
└─ avgunsharedstack[KB],swaps
2016-04-23 00:06:18,[-9.84-]{+6.68+},[-3.86-]{+1.49+},[-14.25-]{+8.41+},[-96%]{+97%+},[-←
└─ 36616-]{+56612+},0,0,0,0
2016-04-23 00:28:01,[-10.27-]{+6.68+},[-3.92-]{+1.53+},[-14.84-]{+9.06+},[-95%]{+90%+},[-←
└─ 37788-]{+57256+},0,0,0,0
2016-04-23 00:51:33,[-8.99-]{+6.73+},[-3.76-]{+1.54+},[-13.47-]{+8.68+},[-94%]{+95%+},[-←
└─ 38720-]{+58748+},0,0,0,0
2016-04-23 00:55:07,[-20.76-]{+14.57+},[-7.96-]{+2.92+},[-30.42-]{+19.77+},[-94%]{+88%+},[-←
└─ 40640-]{+61176+},0,0,0,0
```

Listing 5.3: Word diff: time-based performance measurement of CoastFoundationPerfTest

To be precise, adding the option `--word-diff=color` would make the output even more readable by leaving out the extra enclosing characters and instead just colorize the differing values. But doing so would have made colorizing this listing impossible.

As this was just a way to get a general idea, and the time-based approach is rather limited, we kept looking for more sophisticated tools. Read on.

5.4.3 *perf*

To get an overview of the performance differences between the 32-bit and 64-bit builds, we used `perf stat` as shown in Appendix E. The command internally used is:

```
scons --ignore-missing --with-src-boost=3rdparty/boost \
--with-src-zlib=3rdparty/zlib --with-bin-openssl=3rdparty/openssl \
--build-cfg=optimized --archbits=64 --run-force \
--runparams="-x perf -x stat -x --debug -x '--output \${RESULT}' \
-x '--repeat 20' -- -all" \
$TESTNAME
```

The option `--detailed` stands for detailed output, `--output $RESULT` denotes the result file, and `--repeat 20` causes the test to be run 20 times by *perf*.

Example output:

81.437651	task-clock (msec)	#	0.350 CPUs utilized	
135	context-switches	#	0.002 M/sec	
1	cpu-migrations	#	0.012 K/sec	
1303	page-faults	#	0.015 M/sec	
191305663	cycles	#	2.258 GHz	(50.01%)
<not supported>	stalled-cycles-frontend			
<not supported>	stalled-cycles-backend			
457793461	instructions	#	2.07 insns per cycle	(62.97%)
123582091	branches	#	1458.496 M/sec	(62.91%)

344673	branch-misses	#	0.28% of all branches	(65.30%)
177554196	L1-dcache-loads	#	2095.465 M/sec	(56.92%)
1444585	L1-dcache-load-misses	#	0.90% of all L1-dcache hits	(25.88%)
286722	LLC-loads	#	3.384 M/sec	(24.72%)
5160	LLC-load-misses	#	3.36% of all LL-cache hits	(36.07%)
0.232891130 seconds time elapsed		(+- 1.87%)		

Listing 5.4: Example output of `perf stat`

To understand the meaning of the output, it's necessary to know what each of the counters mean. A legend for those, including a general introduction to *perf stat* can be taken from [13, Perf Examples] and [21, Perf Stat].

Here's a quick overview for the counters and their meanings:

task-clock (msec)

Time in msec spent on profiled task (like CPU time, but measured with dedicated hardware support), where less is better, and how parallel (how many CPUs the task's load was distributed to), where more is better.

context-switches

How often the kernel switched from one process/thread to another one. Less is better.

cpu-migrations

Kernel moves a thread from one CPU to another CPU. Less is better.

page-faults

Accesses to memory pages which are in virtual address space but not loaded in main memory (traps). Less is better.

cycles

Total cycles spent. Less is better, but can be irrelevant for performance (in case of other bottlenecks).

instructions

Instructions per cycle. Generally, more is better, but beware, e.g. spin locks⁵ could lead to high values too.

branches

How many jumps/loops in code. Not directly relevant for performance.

branch-misses

Incorrectly predicted branches. Less is better.

L1-dcache-loads

L1 data cache loads. More is better.

L1-dcache-load-misses

L1 missed data cache loads. Less is better.

LLC-loads

Last level cache loads. More is better.

LLC-load-misses

Missed last level cache loads. Less is better.

seconds time elapsed

Wall clock of time elapsed while running the given task.

⁵<https://en.wikipedia.org/wiki/Spinlock>

Most of the `perf stat` results (detailed diffs in Appendix E) of the COAST test suites are quite similar. It looks like the switch to 64-bit does not have a big impact on performance. Generally there are more cache misses and instructions on 64-bit but also higher instructions per seconds and better branch prediction, which evens it out.

The differences for the following tests are significant enough to give them a closer look. We decided to rerun them and then analyze them with `perf record` and `perf report` if deemed necessary. Overall, the test system seemed to have less load during the second run. But the numbers stayed stable which allowed a comparison. Here's a short summary of the outliers in the 64-bit build:

- `CoastFoundationMiscellaneousTest` (details in Listing E.6)
 - High standard deviation
 - More page faults
 - Same seconds time elapsed
 - Kind of same in the second `perf stat` run (see Listing E.7). Further investigation hasn't been successful, as test is too short and consists mainly of memory allocations.
- `CoastEBCDICTest` (details in Listing E.1)
 - High standard deviation
 - 1.35× slower
 - Looks a lot better after the second `perf stat` run (see Listing E.2), and thus there's no need to further analyze the results.
- `CoastFoundationPerfTest` (details in Listing E.8)
 - High standard deviation
 - 1.9× faster
 - Less page-faults
 - More context switches
 - More L1 dcache misses
 - Less LLC cache misses
 - Kind of same in the second `perf stat` run (see Listing E.9). See the further investigation in subsection 5.4.3.1
- `CoastRegexTest` (details in Listing E.14)
 - Some deviation
 - Looks better in the second `perf stat` run (see Listing E.15).
- `CoastFoundationTimeTest` (details in Listing E.11)
 - Same time
 - More L1 dcache misses (`time_t` now 64-bit?)
 - Looks better in the second `perf stat` run (see Listing E.12).

5.4.3.1 Detailed Analysis: `CoastFoundationPerfTest`

The `perf diff` output in Listing 5.5 tells us that the test spent 17.93% less time in the kernel in 64-bit mode. The handling of page faults in Listing 5.6 in 32-bit mode seems to have the biggest impact. `CoastFoundationPerfTest` is mainly testing memory handling of the [Anything](#)

and `String` classes, particularly the `Anything::IFAHHash` function.

Some ideas as to why much less time is spent in the kernel on 64-bit include:

1. larger buffers
2. better implementation
3. Given the fact that the test has been run on a 64-bit system:
 - (a) expensive syscall conversions on 32-bit
 - (b) 64-bit hardware, although just an extension to x86, might be optimized specifically for 64-bit code

These are just ideas and we have no proof for any of them. Attempts to contact a professor at HSR whose work involves Linux kernel hacking remained without success.

```
$ sudo perf diff -f --sort comm,dso CoastFoundationPerfTest-32_optimized.perf ←
    ↳ CoastFoundationPerfTest-64_optimized.perf
# Event 'cycles'
#
# Baseline      Delta      Command      Shared Object
# .....
#
# 44.40%    -2.83%  CoastFoundation  libCoastFoundationBase.so
# 23.22%   -17.93%  CoastFoundation  [kernel]
# 20.38%    +4.58%  CoastFoundation  libc-2.21.so
# 8.27%    +7.23%  CoastFoundation  libstdc++.so.6.0.21
# 3.73%    +7.22%  CoastFoundation  libCoastFoundationMiscellaneous.so
#                               +1.73%  CoastFoundation  CoastFoundationPerfTest
```

Listing 5.5: Simple `perf diff` of `CoastFoundationPerfTest`

```
$ sudo perf diff -f CoastFoundationPerfTest-32_optimized.perf CoastFoundationPerfTest-64←
    ↳ _optimized.perf | c++filt
# Event 'cycles'
#
# Baseline      Delta      Shared Object      Symbol
# .....
# 18.65%    +3.44%  libc-2.21.so        [.] 0x0000000000006ff61
# 8.25%    -5.87%  [kernel]           [k] page_fault
# 3.92%           [kernel]           [k] clear_page_c_e
# 3.83%    -0.38%  libCoastFoundationBase.so [.] IFAHHash(char const*, long&, ←
    ↳ char, char)
# 3.43%           libCoastFoundationBase.so [.] __x86.get_pc_thunk.bx
# 2.78%    -0.43%  libCoastFoundationBase.so [.] String::Set(long, char const*, ←
    ↳ long)
# 2.42%           libCoastFoundationBase.so [.] AnyIndTable::At(long)
# 2.08%    +2.52%  libCoastFoundationBase.so [.] AnyKeyTable::DoHash(char const←
    ↳ *, bool, long, unsigned long) const
# 2.05%           libCoastFoundationBase.so [.] Allocator::Calloc(int, unsigned←
    ↳ int)
# 1.76%    +1.12%  libstdc++.so.6.0.21 [.] __dynamic_cast
# 1.38%           libCoastFoundationBase.so [.] Allocator::Malloc(unsigned int)
# 1.37%           libCoastFoundationBase.so [.] Anything::SetAllocator(←
    ↳ Allocator*)
# 1.37%           libstdc++.so.6.0.21 [.] std::basic_ostream<char, std::←
    ↳ char_traits<char> >& std::_ostream_insert<char, std::char_traits<char> >(std::←
    ↳ basic_ostream<char, std::char_traits<char> >&, char const*, int)
# 1.26%    +0.37%  libCoastFoundationBase.so [.] String::IntReadFrom(std::←
    ↳ basic_istream<char, std::char_traits<char> >&, char)
# 1.07%           [kernel]           [k] get_mem_cgroup_from_mm
# 1.04%    -0.46%  libCoastFoundationBase.so [.] coast::storage::Current()
# 1.03%           libCoastFoundationBase.so [.] Allocator::RealMemStart(void*)
# 1.03%           libstdc++.so.6.0.21 [.] std::basic_ostream<char, std::←
    ↳ char_traits<char> >::sentry::sentry(std::basic_ostream<char, std::char_traits<←
    ↳ char> >&)
# 1.03%           libCoastFoundationMiscellaneous.so [.] PoolAllocator::Alloc(unsigned ←
    ↳ int)
# 1.03%    +2.98%  libstdc++.so.6.0.21 [.] std::basic_ostream<char, std::←
    ↳ char_traits<char> >::put(char)
```

```

1.02%   +0.09%  libCoastFoundationBase.so      [.] AnyArrayImpl::~AnyArrayImpl()
1.02%                               libCoastFoundationBase.so      [.] String::~String()
1.02%   -1.00%  libCoastFoundationBase.so      [.] AnythingToken::isNameDelimiter(←
    ↘ char)
1.02%                               libCoastFoundationMiscellaneous.so [.] PoolAllocator::FindBucketBySize←
    ↘ (unsigned int)
0.79%   +0.36%  libCoastFoundationBase.so      [.] Anything::operator=(Anything ←
    ↘ const&)
[...]
```

Listing 5.6: Detailed `perf diff` of `CoastFoundationPerfTest`

5.4.4 Valgrind

Based on the *Valgrind Heap Summary* in Table 5.1, memory consumption usually seems to be $1 - 1.6\times$ higher in 64-bit. Since `long` often was used as a substitute for `int` in the COAST framework, that’s surprising. We expected a much higher memory usage instead. We decided to take a closer look at some outliers, namely `CoastFoundationMiscellaneousTest` and `CoastSystemFunctionsTest`. All other tests with a ~ 1.5 factor do not have a specific reason for the increased memory usage — it’s just general higher memory consumption.

Both `CoastFoundationMiscellaneousTest` and `CoastSystemFunctionsTest` show the same graph, as can be seen in Figure 5.1 and Figure 5.2. The initial spike, which is $\sim 2.7 - 3.5\times$ higher on 64-bit, is not caused by the COAST framework directly. Instead, its root cause is the `call_init()` function of *glibc-2.21*. See subsection 5.4.4.1 for our further investigations on this.

Another try was to examine what other shared object files (*.so*) were loaded and whether there is a difference between the tests: `cat /proc/PID/maps`, interesting is the increased size of *libm.so* from 302 KiB in 32-bit to 1.1 MiB on 64-bit. But other tests load this *.so* too, so it can’t be the culprit. *SCons* also uses `-fPIC` in order to avoid loading those libraries into the process memory.

5.4.4.1 About `call_init()`

The *GNU C Library*⁶’s function `call_init()` (defined in *dl-init.c*) is what caused the initial spike in memory usage.

Unfortunately, we were unable to find any useful documentation about this function or other helpful information on this specific effect. Furthermore, COAST loads *shared objects* using the class `AppBooter`, but neither of the two affected tests make use of it.

So we looked into alternative implementations of the C standard library, namely *musl libc*⁷ and *μClibc*⁸, to find any clues as to the purpose of this function. Unfortunately neither of them have this function.

Even though *musl libc* provides a `gcc` command, to transparently use *musl libc*, unfortunately it’s only for C. The other library doesn’t provide anything like this. So integrating either of the two turned out to be unfeasible within a reasonable time frame.

⁶<https://www.gnu.org/software/libc/>

⁷<http://www.musl-libc.org>

⁸<https://uclibc.org>

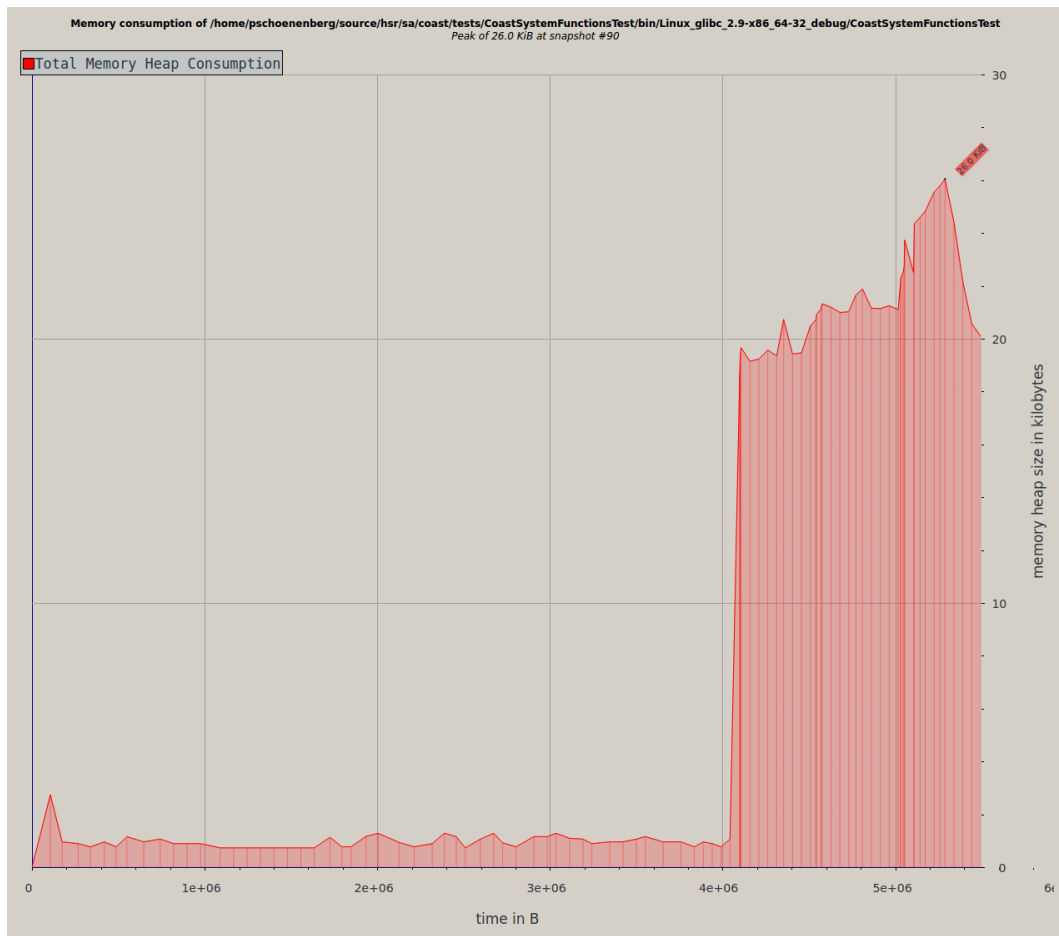


Figure 5.1: Massif output CoastSystemFunctions32

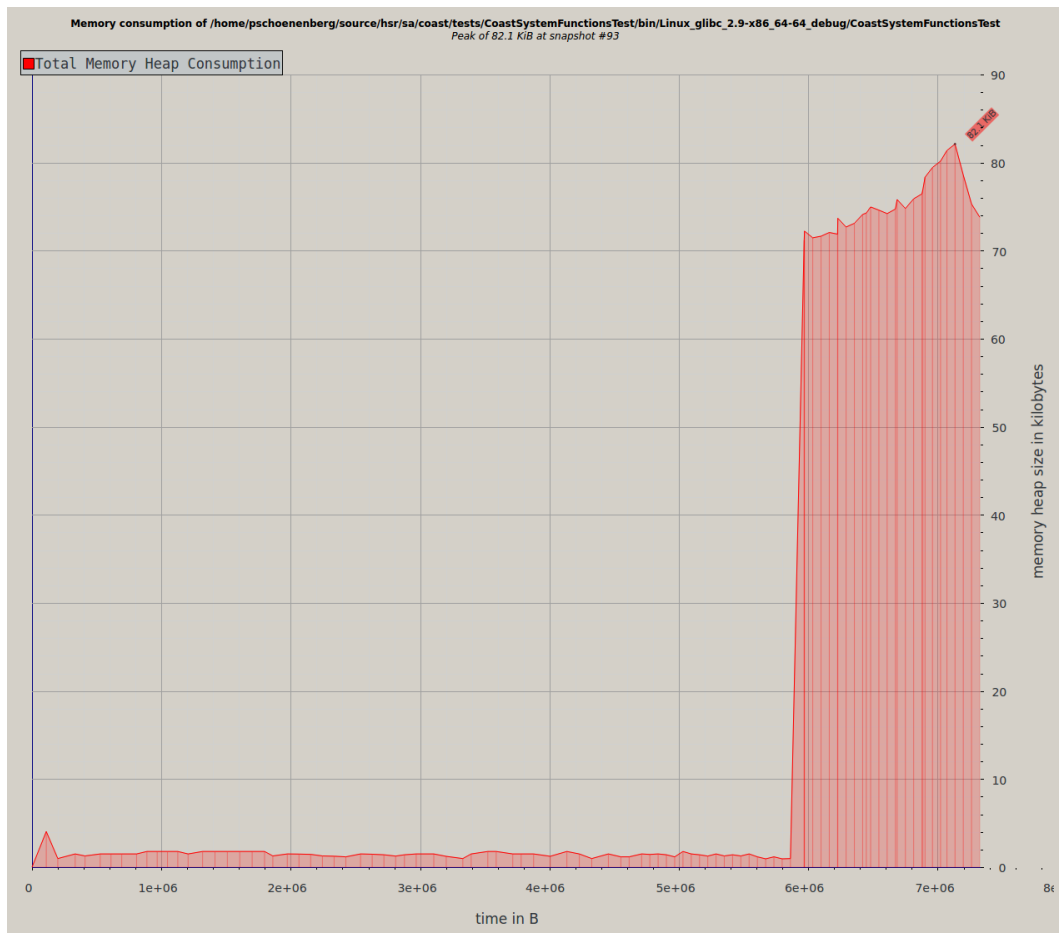


Figure 5.2: Massif output CoastSystemFunctions64

5.4.4.2 SBO/SSO

Interesting is that some tests (including EBCDIC, FoundationBase, FoundationMiscellaneous, and others) result in **less** total allocations. One explanation we figured out in meeting #9, is the *small buffer optimization (SBO)* and *short string optimization (SSO)*. `ITOString` uses something similar to this technique as shown in Listing 5.8. Both `long` members in `StringImpl` doubled in size in 64-bit, which means every allocated string may have 8+ bytes more. Because of `capacity = GetAllocator()->SizeHint(capacity)`; the actual allocation size might differ, because sometimes capacity in 64-bit will allocate the next bigger bucket and sometimes it will still fit in the same bucket as with 32-bit. Our test, where we changed both `long` members to `int`, shows a bit lower memory consumption but no difference in the number of allocations.

Another idea is `boost::function`. We temporarily got rid of the *Small Object Optimization* as shown in Listing 5.7. This change led to more allocations. `CoastFoundationBaseTest` used to result in 3'479'836 as opposed to 3'479'803 allocations with this change. As another test, we added a 4-byte `int` member to the `struct function_buffer` which led to 3'479'833 allocations. It looks like we found one of the culprits.

```
template<typename F>
struct function_allows_small_object_optimization
{
    BOOST_STATIC_CONSTANT
    (bool,
-     value = ((sizeof(F) <= sizeof(function_buffer) &&
-               (alignment_of<function_buffer>::value
-               % alignment_of<F>::value == 0)))));
+     value = false);
};
```

Listing 5.7: *Boost*'s Small Object Optimization temporarily disabled

Explanation for the columns in Table 5.1:

B in use at exit: Bytes in use after exit of program (usually pools)

#allocs: Total of allocations

#frees: Total of frees

allocated: Total allocated bytes in runtime

checked: Total scanned bytes

COAST Testsuite	B in use at exit	#allocs	#frees	B allocated	checked
EBCDIC 32	70'208	7'896	7'841	545'150	144'724
EBCDIC 64	126'224	7'894	7'839	704'725	186'032
Difference	56'016	-2	-2	159'575	41'308
Factor	1.798	1	1	1.293	1.285
Foundation 32	18'944	62'214	62'213	3'091'841	116'164
Foundation 64	72'704	62'214	62'213	4'822'353	157'472
Difference	53'760	0	0	1'730'512	41'308
Factor	3.838	1	1	1.56	1.356
FoundationAnythingOptional 32	18'944	50'357	50'356	2'640'173	99'812
FoundationAnythingOptional 64	72'704	50'357	50'356	4'098'891	128'816
Difference	53'760	0	0	1'458'718	29'004
Factor	3.838	1	1	1.553	1.291
FoundationBase 32	18'944	3'479'808	3'479'807	395'955'577	128'516
FoundationBase 64	72'704	3'479'803	3'479'802	418'179'124	169'824
Difference	53'760	-5	-5	22'223'547	41'308
Factor	3.838	1	1	1.056	1.321
FoundationIO 32	18'944	74'255	74'254	21'220'485	132'532
FoundationIO 64	72'704	74'257	74'256	22'865'257	173'840
Difference	53'760	2	2	1'644'772	41'308
Factor	3.838	1	1	1.078	1.312
FoundationMiscellaneous 32	18'944	1'059	1'058	108'488	103'940
FoundationMiscellaneous 64	72'704	1'051	1'050	185'704	132'944
Difference	53'760	-8	-8	77'216	29'004
Factor	3.838	0.992	0.992	1.712	1.279
FoundationPerf 32	18'944	1'856'644	1'856'643	2'878'515'070	103'908
FoundationPerf 64	72'704	1'842'175	1'842'174	2'927'396'552	132'928
Difference	53'760	-14'469	-14'469	48'881'482	29'020
Factor	3.838	0.992	0.992	1.017	1.279
FoundationTime 32	18'944	62'214	62'213	3'091'841	116'164
FoundationTime 64	72'704	62'214	62'213	4'822'353	157'472
Difference	53'760	0	0	1'730'512	41'308
Factor	3.838	1	1	1.56	1.356
MTFoundation 32	23'950	166'317	166'210	31'424'965	137'324
MTFoundation 64	81'122	166'234	166'127	35'786'710	183'256
Difference	57'172	-83	-83	4'361'745	45'932
Factor	3.387	1	1	1.139	1.334
Regex 32	19'616	728'601	728'597	44'530'017	120'148
Regex 64	73'952	728'601	728'597	59'841'461	169'632
Difference	54'336	0	0	15'311'444	49'484
Factor	3.77	1	1	1.344	1.412
Storage 32	20'128	2'450'204	2'450'199	124'924'687	137'596
Storage 64	74'560	2'450'205	2'450'200	194'316'177	183'440
Difference	54'432	1	1	69'391'490	45'844
Factor	3.704	1	1	1.555	1.333
SystemFunctions 32	18'944	207	206	29'990	99'732
SystemFunctions 64	72'704	207	206	88'038	128'736
Difference	53'760	0	0	58'048	29'004
Factor	3.838	1	1	2.936	1.291
Total: 32	285'454	8'939'776	8'939'597	3'506'078'284	1'440'560
Total: 64	937'490	8'925'212	8'925'033	3'673'107'345	1'904'392
Difference	652'036	-14'564	-14'564	167'029'061	463'832
Factor	3.284	0.998	0.998	1.048	1.322

Table 5.1: Valgrind Heap Summary

```

void String::alloc(long capacity)
{
    // this method is extremely performance sensitive
    // beware of overhead
    // make initial capacity some power of 2 for saving allocs with short
    // strings, only if we have fixed size initializers string buffers
    // smaller than cStrAllocMinimum are created
    if ( capacity <= 0 ) {
        capacity = cStrAllocMinimum;
    }
    capacity += sizeof(*fStringImpl); // add tara

```

```

        capacity = GetAllocator()->SizeHint(capacity);
        fStringImpl = static_cast<StringImpl *>(fAllocator->Calloc(capacity, ←
            sizeof( char )));

        if (!fStringImpl) {
            //--- allocation failed
            SystemLog::Error("String::alloc: Memory allocation failed!");
        } else {
            // subtract to get net capacity again.
            fStringImpl->fCapacity = capacity - sizeof(*fStringImpl);
            // length is 0 by using Calloc
        }
    }

    ///! struct that uses a space efficient trick to implement string
    struct StringImpl {
        ///! the size of the allocated buffer
        long fCapacity;
        ///! the length of the String as perceived by the user of this class
        long fLength;
        ///! the allocated characters follow just after this so the start of the ←
        // characters is (char *) (fStringImpl+1);
        char *Content() {
            return reinterpret_cast<char *>(this + 1);
        }
        ///! the allocated characters follow just after this so the start of the ←
        // characters is (char *) (fStringImpl+1);
        const char *Content() const {
            return reinterpret_cast<char const *>(this + 1);
        }
    }
} *fStringImpl;

```

Listing 5.8: Allocation optimization of `String` in `coast/foundation/base/ITOString.cpp`

Massif shows 64-bit consumption is similarly shaped, but generally higher, as can be seen in Figure 5.3 and Figure 5.4.

Figure 5.4 (64-bit) shows an interesting extra spike. To analyze this spike, we reran *massif* with `--detailed-freq=1` to get more details about where this spike comes from. The following listing shows the respective source code from *PoolAllocator.cpp*:

```

// PoolAllocator::PoolAllocator(long, unsigned long, unsigned long) on line 256
fPoolMemory = ::calloc(fAllocSz, 1);
PoolBuckets = (PoolBucket *)::calloc(fNumOfPoolBucketSizes + 1,
    sizeof(PoolBucket));

```

Listing 5.9: Snippet of `PoolAllocator`’s ctor

...which has been called by the test case `AnythingDeepCloneTest::DeepCloneBug232Test()` (`AnythingDeepCloneTest.cpp:527`) by the line `PoolAllocator p(1, 16384, 20);`.

Since `PoolBucket` is a struct (Listing 5.10) containing two `size_t`, one void pointer and two pointers for `boost::shared_ptr` [3], and all of these types are bigger quantities in *LP64*, this struct now has 5 times the size. This is one possible cause for the spike.

```

struct PoolBucket {
    size_t fSize;
    size_t fUsableSize;
    void *fFirstFree;
    Allocator::MemTrackerPtr fBucketTracker;
};

```

Listing 5.10: Cause for a spike in memory consumption in 64-bit

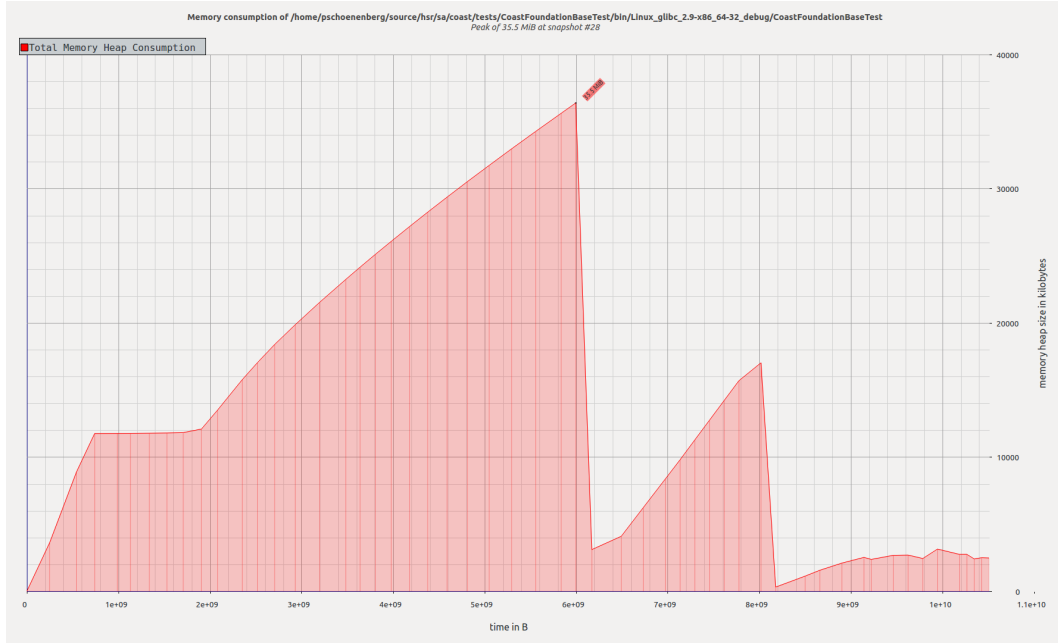


Figure 5.3: Massif output CoastBaseTest32

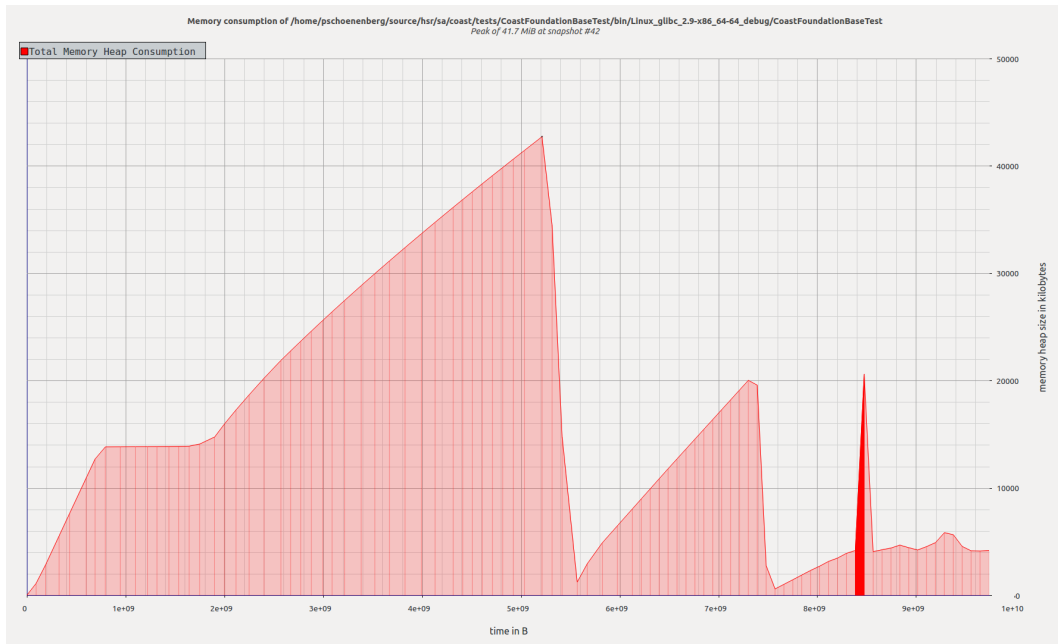


Figure 5.4: Massif output CoastBaseTest64

5.5 Conclusion

As we have seen in subsection 5.4.4, most tests use 1 – 1.5× more memory in 64-bit, with the exception of the two outliers discussed earlier. That’s clearly less than we expected, since COAST used `long` as a replacement for `int`, which doubles in size in the *LP64* data model.

Performance-wise in terms of speed, a port to 64-bit is negligible. Some system calls are faster

on 64-bit but we've been unable to find explicit proof or documentation about it. Nor have we been able to get professional insights about it, as mentioned before.

Both performance and memory consumption are highly sensitive to numerous factors and there is no easy answer. If performance matters, you have to measure, optimize and tune your application for your system and target machine.

Chapter 6

Optional Goal: Performance History

This goal consists of working out a way to get and store the history of performance measurements over time, for example to allow visualizing them later on. This could prove helpful when trying to track down a commit which had a negative impact on performance.

Below are three more specific use cases which describe when the performance history could be useful.

6.1 Use Case #1: Analyzing a Range of Revisions

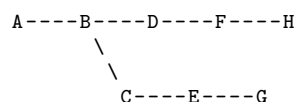
Given a range of revisions, or in Git jargon called a *commit range*, one wants to analyze performance difference imposed by the commits. This could help track down a certain commit as described at the beginning of this chapter.

One example for a commit range could be the last 50 commits on the master branch. Based on this range, the performance effect of said commits could be computed iteratively.

6.1.1 Specifying Commit Ranges

It turns out that Git has very sophisticated commit selection capabilities, including the selection of a range of commits or a certain subset thereof (more on this later). A commit range such as `old..new` selects all commits reachable (“ancestors of”) from `new`, excluding the ones reachable from `old`.

Consider the following commit history:



Every letter denotes a Git commit. The time line is from left to right. Commit H is the tip of the mainline branch, usually called `master`. Commits C, E, and G make up a branch (e.g. `featureX`)

which has descended from commit B. Commit G represents the tip of that branch.

Selecting the range `B..H`, would return the commit set (D, F, H) . This is because it's really just shorthand for `H ^B`, which translates to "all commits reachable from H without the commits reachable from B".

It goes quite a bit further than that¹. For example, specifying `G...H` (three dots) only selects commits that are reachable by either G or H, but not both. This effectively lists all divergent commits between the two branches (G, E, C, H, F, D) .

6.1.2 Interpolating Results

The analysis described above could either be done for all commits of a given commit range, which can take **a long time**, or it could be done just for a subset thereof to save (build) time. In the latter case, the intermediate measurement results can be interpolated, for example using a trend line in the diagram of a spreadsheet application.

Imagine one wants to create performance history data for the time span of a whole year (assuming that the selected test suites haven't changed significantly within that time). Does one really need to build and run the test suite(s) for every commit done in this time span? Not to get a general idea of the performance characteristics, like whether the memory consumption or execution time went generally up or down. A subset of the commit range would suffice. For this example, a subset of maybe 32 data points would be by far enough for interpolation to notice a general trend.

Git has the answer: Passing the option `--bisect-all` to `git-rev-list` will list the commits in bisect order, which is effectively binary search order, where it lists the commits with the largest distance to either boundary first, then the ones whose distance is half as large, and so on. By limiting the number of selected outputs using `--max-count`, it essentially lets us select an appropriate subset of commits to be able to create an interpolated graph subsequently.

Ideally, one might want to choose the number of commits to be a power of two. This would ensure a fairly equal (topological) distance between each pair of commits, if so desired.

6.2 Use Case #2: Analyzing a List of Revisions

Given a list of commits (not a range), the performance of the software at exactly those commits shall be analyzed. This can be useful to show the effects of a single commit, compared to its parent. For this case, the commit and its parent have to be specified.

It could also be useful for an arbitrary set of commits, possibly related to each other in a more complex way. E.g. one needs to take advantage of the full power of `git-rev-list`, and not only its ability to enumerate over a range of commits.

¹<https://www.kernel.org/pub/software/scm/git/docs/gitrevisions.html>

6.3 Use Case #3: Archiving the Results

Properly archiving the results of this script can be necessary to process it in another application, or simply to have it outside of the repository, respectively the *master* branch.

6.4 Non-Functional Requirements

Marcel Huber requested that the solution shouldn't be based on Perl, rather preferring Python or a pure shell script.

6.5 Concept

For the same reasons as described in section 5.1, independent measurement methods will be used. COAST's existing performance related test suites won't be treated in any special way. Instead, the newly created performance measurement scripts will be reused. Of course, they can also be used to analyze the performance related test suites.

The idea is to write an additional shell script `perf-history` that will:

1. process the options and arguments passed
2. deduce a list of desired commits
3. create a new directory (all information related to this run)
4. get confirmation from the user to proceed
5. iterate through each commit:
 - (a) create a result directory for the commit
 - (b) run the chosen measurement script, which will:
 - i. build the necessary executables
 - ii. run the appropriate test suites and measure performance
 - iii. write the results to a file within the newly created directory
 - iv. extract desired measurement values
 - v. append to an accumulated CSV file

Needless to say, the functionality to build, run, and measure performance of certain test suites can be reused from the existing script solutions described in chapter 5.

6.5.1 File Format

Again, CSV is a very versatile format, even more than something like JSON/YAML, as even these seemingly simple formats require specialized commands to be processed on the command line. As one of the Unix philosophies recommends the use of [14, The Unix Philosophy, p. 76]

“common underlying format—the line-oriented, plain text file”

It can still be converted to anything else later if necessary.

6.5.2 Applicability

Given the current set of lower-level performance measurement scripts, this script only makes sense in combination with:

- `time`
- `perf-stat`
- `valgrind`

It does not make sense in conjunction with either *perf* itself or *Valgrind*'s memory consumption analyzing tool *Massif*. This is because their results are not suitable to be exported to CSV.

6.6 Implementation

Of course, the new script has been written to be Bourne shell compatible to ensure the best possible portability across Unix systems. In addition to that, only standard versions of *sed* and *grep* were used. We deliberately avoided using GNU extensions of said utilities (as *GNU is Not Unix*).

6.6.1 Using Current Script Revisions for Old Code

Because of the very nature of this utility, namely jumping back and forth between revisions, we had to find a way to ensure that only the newest version of the performance measurement scripts are used. We had two options:

1. *Git Sparse Checkouts*

This is a feature of Git [12, Sparse checkout] which lets you do sparse checkouts. It means only files that have been deemed interesting by configuration is actually checked out when checking out a specific commit. This turns out to be a very complex feature, and once enabled, it's not straight forward to just disable it. Fully repopulating the working directory would require:

- (a) changing the configuration: making all files interesting again
- (b) checking out the working directory
- (c) disabling the sparse checkout feature

So this feature is not only complex, but also impairs the agility when working with Git.

2. Simply freezing the scripts

By simply copying the existing scripts to the newly created directory, we can guarantee that the current version is used for all commits checked out during the process of creating the performance analysis. It might also make later diagnostics and debugging easier. This is because there could be a dated, accumulated CSV file made up of old columns, which is preferably extended by scripts of the corresponding version.

This option is simple and pragmatic.

Needless to say, we decided to implement the second option.

6.7 Usage

Here's the synopsis from `perf/perf-history --help`:

SYNOPSIS

```
(1) ./perf-history [--method=METHOD] [--max-count=N] <commit>...
(2) ./perf-history [--method=METHOD] --stdin
```

The full usage text (116 lines) can be found in Appendix G.

The two forms listed directly correspond to the use cases #1 and #2.

6.7.1 Selecting the Measurement Method

As shown in the previous section, using the option `--method` the performance measurement method can be specified. By default, it's `time`. All currently supported methods are:

- `--method=time`
- `--method=perf_stat` (for `perf-stat`)
- `--method=valgrind`

6.7.1.1 Extensibility

As you can see, these values exactly correspond to the previously added, lower-level performance measurement scripts. That's because that's exactly how they're selected: The method name given is taken into account when looking up the lower-level script. It's expected that it (or a symlink to it) exists next to the `perf-history` script itself, starting with `with_`. For example, running `perf/perf-history --method=foo` will make it look for a script called `perf/with_foo`.

This makes extending this infrastructure trivial. An additional method really only needs to be able to do two things:

1. hook into a `scons` command and measure a given test suite
2. export the result as CSV on STDOUT (including a header) if the `--export` option was given

Common functionality like deducing a list of desired test suites, iterating over the desired processor architectures, building the executables, performing a warmup round, printing progress information, etc. can simply be loaded and reused.

6.7.1.2 Example Extension

As a matter of fact, it should be fairly trivial to define an additional lower-level script which merely acts as a wrapper around COAST's test suites, but provides the ability to export the already printed run durations as CSV. It could be called `with_self`, and use something like the following `sed` script to filter and transform the relevant lines from the test suite output:

```
sed -n 's/^--\(\w\+\)\.\(\w\+\)--s*(\([^[:digit:]]+\)ms).*\/\1,\2,\3/p'
```

Consider the following input:

```
Running RegexTest
--RegexTest.MatchLiteral-- (0ms)
--RegexTest.MatchDot-- (0ms)
--RegexTest.MatchDotDotDot-- (0ms)
--RegexTest.MatchDotStar-- (0ms)
--RegexTest.MatchAStar-- (0ms)
--RegexTest.LargeLiteralTest-- (10ms)
--RegexTest.BackRefTest-- (0ms)
--RegexTest.ShortLiteralTest-- (20ms)
--RegexTest.LargeDotStarTest-- (10ms)
--RegexTest.MatchConfig-- (0ms)
--RegexTest.MatchFlagsTest-- (0ms)
--RegexTest.SplitTest-- (0ms)
--RegexTest.SubstTest-- (0ms)
--RegexTest.GrepTest-- (0ms)
--RegexTest.GrepSlotNamesTest-- (0ms)
--RegexTest.GetMatchTest-- (0ms)
OK (16 tests and 456 assertions in 40 ms)
```

The input would be transformed into the following CSV records:

```
RegexTest,MatchLiteral,0
RegexTest,MatchDot,0
RegexTest,MatchDotDotDot,0
RegexTest,MatchDotStar,0
RegexTest,MatchAStar,0
RegexTest,LargeLiteralTest,10
RegexTest,BackRefTest,0
RegexTest,ShortLiteralTest,20
RegexTest,LargeDotStarTest,10
RegexTest,MatchConfig,0
RegexTest,MatchFlagsTest,0
RegexTest,SplitTest,0
RegexTest,SubstTest,0
RegexTest,GrepTest,0
RegexTest,GrepSlotNamesTest,0
RegexTest,GetMatchTest,0
```

Notice that the superfluous lines at the top and bottom have been skipped.

6.7.2 Selecting the Test Suites

By default, all tests from the mandatory goal are taken. They're called *core tests* in the usages. They're simply the list of test suite names stored in *perf/lib/core_tests.txt*.

If one wants to run the performance analysis for a certain set of test suites, the environment variable `TEST_NAMES` can be set to point to a file containing the desired list.

6.7.2.1 Example

To run all test suites, the appropriate file containing all tests can easily be created in advance like this:

```
scons -u --showtargets | grep '^ - .*Test$' | cut -d' ' -f3 > all_tests.txt
```

Afterwards, the performance history script can be started like this:

```
TEST_NAMES=all_tests.txt ./perf-history [<option>...] [<commit>...]
```

6.7.3 Selecting the Processor Architecture

By default, executables are only built for the current processor architecture (unlike the performance measurement scripts used standalone, which will build for 64-bit and 32-bit by default).

To change the default, set the environment variable `ALL_ARCHBITS`.

6.7.3.1 Example

To build and measure every selected test suite for both 32-bit and 64-bit, use the following command structure:

```
ALL_ARCHBITS="32 64" ./perf-history [<option>...] [<commit>...]
```

6.7.4 Example of Use Case #1

Assuming one wants to create a performance analysis using *perf* over the range of commits `B..H` (as described above), the following command can be used:

```
./perf-history --method=perf B..H
```

This will check out and measure up to 16 commits by default.

The values from the first column of Listing 5.4 will be extracted and combined into the file *perf/history-YYYYMMDD-XXXXXX/result.csv*, where `YYYYMMDD` is the current date and `XXXXXX` is a random sequence of characters (the effect of `mktemp -d`).

Here's an example of the resulting CSV:

```
commit, testname, archbits, buildcfg, method, times_run, elapsed_time[s], elapsed_time_derivation←
↳ [%], task_clock[msec], context_switches, cpu_migrations, page_faults, cycles, ←
↳ stalled_cycles_frontend, stalled_cycles_backend, instructions, branches, branch_misses, ←
↳ ll_dcache_loads, ll_dcache_load_misses, llc_loads, llc_load_misses
5709245, CoastRegexTest, 64, optimized, perf_stat←
↳ , 20, 0.095087504, 0.49, 69.906876, 143, 0, 1303, , , , , , ,
4162373, CoastRegexTest, 64, optimized, perf_stat←
↳ , 20, 0.098277338, 0.83, 69.808484, 135, 0, 1299, , , , , , ,
ad5e2b8, CoastRegexTest, 64, optimized, perf_stat←
↳ , 20, 0.097705799, 0.61, 75.112279, 136, 0, 1303, , , , , , ,
62c0932, CoastRegexTest, 64, optimized, perf_stat←
↳ , 20, 0.096450159, 0.81, 69.602125, 412, 0, 1300, , , , , , ,
74c415d, CoastRegexTest, 64, optimized, perf_stat←
↳ , 20, 0.097863627, 0.90, 70.066295, 135, 0, 1303, , , , , , ,
f4c8ee9, CoastRegexTest, 64, optimized, perf_stat←
↳ , 20, 0.098398006, 1.03, 77.321331, 399, 0, 1299, , , , , , ,
5b36c3b, CoastRegexTest, 64, optimized, perf_stat←
↳ , 20, 0.094257277, 1.06, 72.752429, 135, 0, 1300, , , , , , ,
acd9d99, CoastRegexTest, 64, optimized, perf_stat←
↳ , 20, 0.096416857, 0.68, 70.069705, 372, 0, 1303, , , , , , ,
66ae973, CoastRegexTest, 64, optimized, perf_stat←
↳ , 20, 0.098175875, 1.08, 70.042155, 135, 0, 1300, , , , , , ,
```

Listing 6.1: Example performance history result (CSV) (*perf-stat*)

Of course this is just an example, limited to one single test suite.

As you can see, some of the columns were left empty. That's because **perf-stat** reported **<not supported>** for those fields. That's probably because it was run on a VM which doesn't have proper access to the lower level performance counter system. But the extraction logic is there and would extract the values accordingly, if they were reported by **perf-stat**.

Using a spreadsheet application, one could now filter/sort by any of the columns, including but not limited to:

commit the commit SHA1

testname the test suite name

archbits processor architecture such as 32 or 64

archbits build configuration such as **debug** or **optimized**

times_run how many times the test suite executable has been run

elapsed_time[s] elapsed time (wall-clock time) in seconds

6.7.5 Example of Use Case #2

Assuming has a certain set of commits ready in a file *my_commits.txt* and wants to use *Valgrind* to analyze the performance of each of the commits, the following command will be helpful:

```
./perf-history --method=valgrind --stdin < my_commits.txt
```

This will extract *Valgrind*'s **LEAK SUMMARY** and **HEAP SUMMARY** values and combine them into the resulting CSV file *perf/history-YYYYMMDD-XXXXXX/result.csv*.

6.7.6 About Use Case #3

Here, it's really up to the user what to do with the resulting CSV file. They might decide to either:

- just leave it where it was created (maybe add its directory to *.gitignore*),
- fill it into a DBMS, or
- put it into its own branch dedicated to archived performance measurements.

The third option could be done using the following command:

```
git checkout --orphan perf_archive
```

This will create a completely independent branch (not descending from a given commit) called *perf_archive* which could be used to keep the data in a safe, rather isolated place.

Chapter 7

Optional Goal: Migrating Further Tests

7.1 Concept

The same concept as in chapter 4 was used to migrate the remaining tests.

The following test suites worked out of the box in 64-bit mode:

- *CoastWDBaseTest*
- *CoastHTTPTest*
- *CoastSSLTest*
- *CoastPerfTest*
- *CoastPerfTestTest*
- *CoastFunctionalActionsTest*
- *CoastWorkerPoolManagerTest*
- *CoastNTLMAuthTest*
- *CoastAccessControlTest*
- *CoastHTMLRenderersTest*
- *CoastActionsTest*
- *CoastAppLogTest*
- *CoastStdDataAccessTest*
- *CoastStringRenderersTest*
- *CoastDataAccessTest*

The next section covers in detail how the broken tests were fixed.

7.2 Implementation

7.2.1 CoastSecurityTest

The following listing shows the initial situation when compiled with 64-bit. Some tests pass, some merely fail and others core dump with an segmentation fault.

```
void setupRunner(TestRunner &runner)
{
    // ok
    ADD_SUITE(runner, Base64Test);
    // core dump
    ADD_SUITE(runner, MD5Test);
    // failures
    ADD_SUITE(runner, BlowfishTest);
    // core dump
    ADD_SUITE(runner, ScrambleStateTest);
    // ok
    ADD_SUITE(runner, TableCompressorTest);
    // core dump
    ADD_SUITE(runner, NewRendererTest);
    // core dump
    ADD_SUITE(runner, UniqueIdGenTest);
} // setupRunner
```

Listing 7.1: Overview of CoastSecurityTest’s test suites and their status

7.2.1.1 MD5

Some test suites in the CoastSecurityTest simply failed and others even crashed with a core dump. Both *md5* and *blowfish* core dumped which was unacceptable when other modules depend on them. Since we are by far no security experts and *COAST* already depends on the *OpenSSL* library, we asked our supervisor Marcel Huber if we could just replace the algorithms with the *OpenSSL* implementation. He agreed under the condition that we must not introduce a new dependency.

Because *COAST*’s and *OpenSSL*’s interfaces are so similar (see Listing 7.2 and Listing 7.3), this switch was quite easy. There is one discrepancy though: The function `MD5_Final()` erases the context. This means the `MD5Context` object is not usable after a call to `MD5_Final()`. We could fix this by either explicitly requiring a call to `MD5Context::Init()` and make this invariant explicit, or call `MD5Context::Init()` in `MD5Context::Final()` after `MD5_Final()` to reinitialize the invariant. The first solution would require some changes and the second solution imposes an unnecessary performance hit. Since this class is already being used correctly and merely an implementation detail in *MD5.cpp*, we decided to write a comment and live with this discrepancy.

```
class MD5Context {
    MD5Context(const MD5Context &);
    MD5Context &operator=(const MD5Context &);
public:
    MD5Context();

    void Init();
    void Update(const unsigned char *buf, unsigned len);
    void Final(unsigned char digest[16]);
    void Transform(uint32 buf[4], const uint32 in[16]);

protected:
```

```

uint32 fBuf[4];
uint32 fBits[2];
unsigned char fIn[64];
};

```

Listing 7.2: COAST's `MD5Context` API

```

int MD5_Init(MD5_CTX *c);
int MD5_Update(MD5_CTX *c, const void *data, unsigned long len);
int MD5_Final(unsigned char *md, MD5_CTX *c);

```

Listing 7.3: *OpenSSL*'s MD5 API

The structure of `MD5Context` looks quite similar to Colin Plumb's MD5 implementation [1]. This class also has the same disadvantages; that an 32-bit integer data type and compile-time endianness configuration is required. *OpenSSL*'s implementation fixed both issues.

To be able to use *OpenSSL*'s crypto library, we had to add `-lcrypto`. Since it's already a dependency and known to *SConsider*, this adjustment in *CoastSecurity.sconsider* was simple: `'linkDependencies': ['CoastWDBase', 'openssl']`

After the switch to *openssl*'s implementation of MD5, all tests failing with core dumps are now "just failing" or in case of `MD5Test` and `UniqueIdGenTest` even fixed.

7.2.1.2 Blowfish

Similar to the MD5 test suite, other test suites, classes and modules in *CoastSecurity* make use of the `BlowfishScrambler`. So it's a natural choice to fix this module in case of dependent errors. We found out, that only a minor fix was required. See here:

```

diff --git a/coast/modules/Security/Blowfish.h b/coast/modules/Security/Blowfish.h
--- a/coast/modules/Security/Blowfish.h
+++ b/coast/modules/Security/Blowfish.h
@@ -11,6 +11,8 @@

#include "SecurityModule.h"

+#include <stdint.h>
+
//---- BlowfishScrambler -----
class BlowfishScrambler : public Scrambler
{
@@ -28,7 +30,7 @@ public:
#define BF_ROUNDS      16
#define BF_BLOCK       8
#if !defined(BF_LONG)
-#define BF_LONG unsigned long
+#define BF_LONG uint32_t
#endif
    struct BlowfishKey {
        BF_LONG P[BF_ROUNDS+2];

```

Listing 7.4: Fix for *coast/modules/Security/Blowfish.h*

After that fix, all remaining test suites in *CoastSecurityTest* are executed successfully.

7.2.1.3 Switch to *OpenSSL*

Since *OpenSSL* is already a dependency of COAST and we also introduced this dependency in the *CoastSecurityModule* with the MD5 fix, we tried to switch the inhouse *Blowfish* implementation to *OpenSSL*'s implementation. The main motivation was that *OpenSSL* is actively developed

and should be more secure with less maintenance overhead. Using our `perf-history` script described in chapter 6, we even could have compared both implementations under performance aspects.

The port to *OpenSSL* of the *Blowfish ECB mode* was quite trivial; change some function calls, remove old stuff and everything worked out of the box. But the *Blowfish CBC mode* in the COAST implementation seems to have a different padding and endianness handling than *OpenSSL*'s implementation. This caused 408 failures in the `CoastSecurityTest`. Fixing these tests without breaking backwards compatibility and introducing new security issues was too risky, so we decided to keep the COAST's handcrafted implementation.

7.2.1.4 Suggestion

We also believe that the effort would better be spent in switching to a more modern encryption algorithm like *AES*. To quote Bruce Schneier in 2007 [6, Bruce Schneier interview]

“At this point, though, I’m amazed it’s still being used. If people ask, I recommend Twofish instead.”

In addition to that, given the severe security flaws of the ECB (Electronic Code Book) mode, we advise to remove it completely.

7.2.2 CoastQueueingTest

The tests in this suite failed in nondeterministic manner, e.g. seg faults, random fails etc. We decided to use the testcase `void QueueWorkingModuleTest::GetAndPutbackTest()` to track down the cause, because this was the first test we found which simply failed without core dumps. Instrumenting this test and using the COAST tracing framework, we managed to track down the error to the sema macros in *SystemAPI.h*. The semaphores are initialized by `int sem_init(sem_t *sem, int pshared, unsigned int value)` [11], the `unsigned int value` parameter in particular. COAST's `class QueueBase` used to have a `long` as `size_type`.

Since an `unsigned int` overflow is well defined behavior, as discussed in subsection 4.2.10, the semaphores used this value, but the entire synchronization logic was undermined, resulting in all sorts of weird threading issues (race condition, dead locks, etc).

We fixed this by changing `size_type` to `int`. `unsigned int` can't be used, because `-1` is used as a sentinel value indicating “queue has shut down already”.

7.2.3 CoastRendererTest

This test failed with three failures, all of them concerning domain names. At a first glance, it looked like a configuration problem, but strangely it only happened in 64-bit, so that possibility was been ruled out. See this listing:

```
!!!FAILURES!!!
Test Results:
Run: 1   Failures: 3   Errors: 0
(204 assertions ran successfully in 180 ms)
There were 3 failures:
1) NewRendererTest.TestCases: coast/wdtest/bases/NewRendererTest.cpp:67:
Difference at position 1
```



```

expected                                     | differences
5B 64 2D 70 68 73 2E 64 61 74 61 6C 61 6E 2E 63 [d-phs.datalan.c | .. 6C 6F 63 ←
    ↳ 61 6C 68 6F 73 74 5D                               localhost]
68 5D                                     h] |
; NewRendererTestConfig.any:0 at TestCases.GetThisHostNameRendererFullTest
2) NewRendererTest.TestCases: coast/wdtest/bases/NewRendererTest.cpp:67:
Difference at position 1
expected                                     | differences
5B 64 2D 70 68 73 5D                               [d-phs] | .. 6C 6F 63 ←
    ↳ 61 6C 68 6F 73 74 5D                               localhost]
; NewRendererTestConfig.any:0 at TestCases.GetThisHostNameRendererHostTest
3) NewRendererTest.TestCases: coast/wdtest/bases/NewRendererTest.cpp:67:
Difference at position 1
expected                                     | differences
5B 64 61 74 61 6C 61 6E 2E 63 68 5D                               [datalan.ch] | .. 5D ←
    ↳                                     ]
; NewRendererTestConfig.any:0 at TestCases.GetThisHostNameRendererDNSTest

```

The affected function of COAST is:

```

bool LinuxResolver::IP2DNS(const String &ipAddress, unsigned long addr)
{
    StartTrace1(Resolver.IP2DNS, "<linux> ip [" << ipAddress << "]);
    struct hostent he;
    struct hostent *res = 0;
    int err = 0;

    const int bufSz = 8192;
    char buf[bufSz];

    int result = gethostbyaddr_r((char *)&addr, sizeof(addr), AF_INET, &he, ←
        ↳ buf, bufSz, &res, &err);
    if ( result == 0 && err == NETDB_SUCCESS) {
        extractFromHostent(*res);
        return true;
    }
    return false;
}

```

Listing 7.5: Broken method in *coast/foundation/io/Resolver.cpp*

According to [11], the function `int gethostbyaddr_r(const void *addr, ...)` takes a `const void *addr` as the first parameter. COAST used to pass an `unsigned long *` as that parameter, which used to work on 32-bit, since an `unsigned long` in the *ILP32* data model is a 32-bit quantity, just like an IPv4 address.

However, on 64-bit, this makes the function return the error 2, which translates to the unhelpful error message

“TRY_AGAIN - A temporary error occurred on an authoritative name server. Try again later.”

Changing the `unsigned long *` to `uint32_t`, which is appropriate for IPv4 addresses no matter the memory model actually being used, fixes the problem.

On a side note: Limiting support to only IPv4 addresses is appropriate in this case, as the third parameter (hard-coded to `AF_INET`) explicitly denotes the IPv4 address family.

7.2.3.1 Discovery of a COAST Bug

Unrelated to any porting issues, we've discovered a new bug. It's reproducible on both 32-bit and 64-bit and makes this test crash with a segmentation fault when run with `--runparams="-d -d -- "`, which causes the test binary to be executed with *GDB*:

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000 in ?? ()
20160526152440: ===== GDB backtrace =====
#0 0x00000000 in ?? ()
No symbol table info available.
#1 0xf7dcd956 in coast::memory::safeFree (a=0x80c9a10, ptr=0x80c9ca0) at coast/↵
↳ foundation/base/ITOSTorage.cpp:273
    __PRETTY_FUNCTION__ = "void coast::memory::safeFree(Allocator*, void*)"
#2 0xf7dbb7a6 in coast::SegStorAllocatorNewDelete<AnyKeyAssoc>::operator delete↵
↳ [] (ptr=0x80c9ca4) at coast/foundation/base/SegStorAllocatorNewDelete.h:85
    __PRETTY_FUNCTION__ = "static void coast::SegStorAllocatorNewDelete<T>::↵
↳ operator delete [](void*) [with T = AnyKeyAssoc]"
    realPtr = 0x80c9ca0
    a = 0x80c9a10
#3 0xf7db872b in AnyArrayImpl::~~AnyArrayImpl (this=0x80d1c30, __in_chrg=↵
↳ optimized out) at coast/foundation/base/AnyImpls.cpp:600
    j = 0
#4 0xf7db884f in AnyArrayImpl::~~AnyArrayImpl (this=0x80d1c30, __in_chrg=↵
↳ optimized out) at coast/foundation/base/AnyImpls.cpp:619
No locals.
#5 0xf7dc7c9c in AnyImpl::Unref (this=0x80d1c30) at coast/foundation/base/↵
↳ AnyImpls.h:67
No locals.
#6 0xf7dc0114 in Anything::operator= (this=0x80c4adc <coast::utility::↵
↳ singleton_default<CacheHandlerImpl>::instance()::obj+28>, a=...) at coast/↵
↳ foundation/base/Anything.cpp:1155
    al = 0x80c9a10
    oldImpl = 0x80d1c30
#7 0xf7dc110d in Anything::clear (this=0x80c4adc <coast::utility::↵
↳ singleton_default<CacheHandlerImpl>::instance()::obj+28>) at coast/↵
↳ foundation/base/Anything.cpp:1566
No locals.
#8 0xf7cfc91c in CacheHandlerImpl::~~CacheHandlerImpl (this=0x80c4ac0 <coast::↵
↳ utility::singleton_default<CacheHandlerImpl>::instance()::obj>, __in_chrg↵
↳ =optimized out) at coast/wdbase/CacheHandler.cpp:36
No locals.
#9 0xf7e29c63 in ?? () from /coast/lib/Linux_glibc_2.9-x86_64-32_debug_trace/↵
↳ libc.so.6
No symbol table info available.
#10 0xf7e29cc1 in exit () from /coast/lib/Linux_glibc_2.9-x86_64-32_debug_trace/↵
↳ libc.so.6
No symbol table info available.
#11 0xf7e1374a in __libc_start_main () from /coast/lib/Linux_glibc_2.9-x86_64-32↵
↳ _debug_trace/libc.so.6
No symbol table info available.
#12 0x08053921 in _start ()
No symbol table info available.
eax                0x0          0
ecx                0x80c9ca8      135044264
edx                0x0          0
ebx                0xf7df90ac     -136343380
esp                0xffffd96c     0xffffd96c
ebp                0xffffd988     0xffffd988
esi                0x80c9ca8      135044264
edi                0xf7fb241c     -134536164
eip                0x0          0x0
eflags             0x10296    [ PF AF SF IF RF ]
cs                 0x23         35
ss                 0x2b         43
ds                 0x2b         43
es                 0x2b         43
fs                 0x0          0
gs                 0x63         99
```

```
=> 0x0: /tmp/CoastRendererTest.sh_23292:35: Error in sourced command file:  
Cannot access memory at address 0x0  
scons: done building targets.
```

There might be a latent bug lurking, but time is running out, which renders an in-depth analysis impossible.

Chapter 8

Optional Goal: C++11/14 Support

8.1 Analysis

We compiled COAST with `scons --use-lang-features=c++0x` (where `c++0x` is equivalent to `c++11`) and it did compile without changes, although with quite a few warnings about the now deprecated `std::auto_ptr`. However, all of these warnings came from the *Boost* library, particularly its file `boost/smart_ptr/shared_ptr.hpp`.

Strictly speaking, no changes were necessary to make the code conform to C++11. However, we made a few changes to improve on the warning situation and reduce code complexity, as documented below in section 8.2. We also removed obsolete keywords and magic comments, as described in section 8.4.

With these changes implemented, we also compiled the COAST framework for C++14 (`scons --use-lang-features=c++14`), which behaved the same as on C++11. So COAST now conforms to C++11 as well as C++14.

8.2 About `std::auto_ptr`

All occurrences of `std::auto_ptr` within COAST used to be wrapped in `#if`s, which has been discussed before and shown in Listing 8.1. This has been the case since the upstream COAST commit `cd7f51d`¹.

`std::auto_ptr` (declared in the `<memory>` header) has been deprecated since C++11 and will be removed in C++17. It's a *smart pointer* that used to provide the means to perform automatic object destruction and a way to enforce unique object ownership before C++ had *move semantics*. However, the safer alternative `std::unique_ptr`, introduced in C++11, is the preferred choice to achieve the same in modern C++.

`std::unique_ptr` is considered superior because its API leaves no ambiguity — ownership has to be transferred explicitly using `std::move()` as opposed to implicitly using a `copy assignment`. Furthermore, it supports arrays which have to be deallocated using `delete[]`, whereas `std::auto_ptr` would always attempt to use `delete`.

¹<https://gerrit.coast-project.org/gitweb?p=coast.git;a=commit;h=cd7f51d1d3e41b7678238f3502773c8053e05ac2>

8.2.1 Migrating to `std::unique_ptr`

Generally, no issues should occur when migrating from `std::auto_ptr` to the safer alternative. However, there is one caveat: `std::auto_ptr` treated copies as moves by modifying (nullifying) the *rhs* during a copy assignment and during copy construction. This was a way to emulate the now standardized *move semantics* to enforce unique ownership of an object.

Any legacy code that relies on this nullifying behavior won't compile anymore, which is a good thing since it doesn't just silently break functionality. The affected code will have to be changed to use `std::move()` to explicitly nullify the *rhs*. So the following code:

```
std::auto_ptr<T> p(new T);
std::auto_ptr<T> p2 = p;
```

...would have to be changed to:

```
std::unique_ptr<T> p(new T);
std::unique_ptr<T> p2 = std::move(p);
```

8.2.2 Reducing Verbosity

To disable the warnings about *Boost*'s use of the deprecated `std::auto_ptr` when compiling, we had the following options:

1. Setting the macro definition `BOOST_NO_AUTO_PTR` to make *Boost* avoid using `std::auto_ptr` in C++11 and on.

The build system would make it easy to set this definition in the build environment using the `CPPDEFINES`² key.

Unfortunately it's unfeasible to do this consistently, as the user is given the choice to use COAST's own 3rd party repository of *Boost*, or use the one provided by the system. If only *Boost*'s header files would be used, it might have been debatable. But COAST also makes use of *Boost*'s *Regex* and *System* libraries, which could be compiled externally where *SConsider* isn't used and thus can't ensure the same build environment. We cannot risk incompatibilities like this.

2. Passing `-Wno-deprecated-declarations` to *GCC*.

This option³ disables all warnings about deprecated declarations including functions, variables, and types. By default, these warnings are enabled.

However, the indirection introduced by the build system makes this unmanageable without changing *SConsider* itself.

It also entails the significant drawback that these kinds of warnings would most likely be disabled for more files than actually needed, which could lead to issues later on as discussed

²<https://coast-project.org/projects/sconsider/wiki>

³<https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

before in subsection 4.2.1 on page 17.

3. Selectively disabling these warnings.

It turns out there's a way to selectively pass certain compiler options to just parts of the code base using the `#pragma` directive.

Fortunately, this option does not entail the previous options' drawback because it can be used very selectively, effectively ignoring `-Wdeprecated-declarations` in only the affected source code sections of *Boost* where `std::auto_ptr` is used.

Of course, we agreed upon the third option which solves the problem in a reasonably pragmatic, yet responsible way.

The following snippet (from *boost/shared_ptr.hpp*) shows how the option `-Wdeprecated-declarations` is ignored and then unignored after the affected source file has been included:

```

// BEGIN ignore deprecation warnings about std::auto_ptr
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wdeprecated-declarations"
+
#include <boost/smart_ptr/shared_ptr.hpp>
+
// END ignore deprecation warnings about std::auto_ptr
#pragma GCC diagnostic pop

```

This patch has been handed over to Marcel Huber to be applied in COAST's 3rd party *Boost* repository.

8.3 A More Transparent Alternative to Preprocessor Switches

The COAST framework was already making use of an intelligently defined namespace called `boost_or_tr1`, defined in *coast/foundation/base/ITOTypeTraits.h*, as can be seen here:

```

if defined(USE_TR1)
#include <tr1/type_traits>
namespace boost_or_tr1 = std::tr1;
#elif defined(USE_STD0X) || defined(USE_STD11) || defined(USE_STD14) || \
      defined(USE_STD17) || defined(USE_STD1y) || defined(USE_STD1z)
#include <type_traits>
namespace boost_or_tr1 = std;
#else // USE_STD03
#include <boost/type_traits.hpp>
namespace boost_or_tr1 = boost;
#endif

```

Depending on the correct definitions of `USE_STD03`, `USE_TR1`, `USE_STD11` and friends, this acts as an elegant compile-time switch to be able to transparently use more recent features from the `<type_traits>` header. The actual implementation would either be provided by the standard library itself, in case it's compiled on C++11 or newer, by the *Technical Report 1* C++ library extensions, if available, or by *Boost* libraries.

However, considering that TR1 was only a proposal and mainly existed to ease the transition to C++11, this namespace was misnamed and thus has been renamed to `boost_or_std`.

Furthermore, and in a more serious matter, it can't be used for `std::unique_ptr` which is because neither *Boost*, nor *TR 1* can provide it on C++ versions older than C++11. Consider this code snippet from *coast/modules/Queueing/QueueWorkingModule.h*, which is just one of many occurrences (11 of this kind across 9 files):

```
#if __cplusplus >= 201103L
    typedef std::unique_ptr<QueueType> QueueTypePtr;
    typedef std::unique_ptr<Context> ContextPtr;
#else
    typedef std::auto_ptr<QueueType> QueueTypePtr;
    typedef std::auto_ptr<Context> ContextPtr;
#endif
```

Listing 8.1: Preprocessor switch to decide between `std::unique_ptr` and `std::auto_ptr`

This increases code complexity unnecessarily and is error prone. So we wanted to work out a solution to reduce code complexity in a similar fashion as above. Using the solution, which is described in the next section, this has been refactored to:

```
typedef boost_or_std::auto_ptr<QueueType> QueueTypePtr;
typedef boost_or_std::auto_ptr<Context> ContextPtr;
```

Of course this refactoring has been applied to all other occurrences of this kind of code smell as well.

8.3.1 Detailed Solution

Two new header files have been created. One is for the intelligent definition of `<type_traits>` features, the other one for the intelligent definition of `<memory>` features.

The original preprocessor switch has been extracted to its own file and slightly changed to this:

```
#ifndef BOOST_OR_STD_TYPE_TRAITS_H
#define BOOST_OR_STD_TYPE_TRAITS_H

#if defined(USE_TR1)
#include <tr1/type_traits>
namespace boost_or_std {
    using namespace std::tr1;
}
#elif defined(USE_STD0X) || defined(USE_STD11) || defined(USE_STD14) || \
defined(USE_STD17) || defined(USE_STD1y) || defined(USE_STD1z)
#include <type_traits>
namespace boost_or_std {
    using namespace std;
}
#else // USE_STD03
#include <boost/type_traits.hpp>
namespace boost_or_std {
    using namespace boost;
}
#endif

#endif // BOOST_OR_STD_TYPE_TRAITS_H
```

Listing 8.2: New file *coast/foundation/base/boost_or_std/type_traits.h*

The header file for `<memory>` features is:

```
#ifndef BOOST_OR_STD_MEMORY_H
#define BOOST_OR_STD_MEMORY_H

#if defined(USE_TR1)
#include <tr1/memory>
namespace boost_or_std {
    using std::auto_ptr;
    using std::tr1::shared_ptr;
}
#elif defined(USE_STD0X) || defined(USE_STD11) || defined(USE_STD14) ||\
defined(USE_STD17) || defined(USE_STD1y) || defined(USE_STD1z)
#include <memory>
namespace boost_or_std {
    template <class T, class Deleter = std::default_delete<T> >
        using auto_ptr = std::unique_ptr<T, Deleter>;

    using shared_ptr = std::shared_ptr;
}
#else // USE_STD03
#include <boost/shared_ptr.hpp>
#include <memory>
namespace boost_or_std {
    using std::auto_ptr;
    using boost::shared_ptr;
}
#endif

#endif // BOOST_OR_STD_MEMORY_H
```

Listing 8.3: New file *coast/foundation/base/boost_or_std/memory.h*

Notice how neither of them declares the `boost_or_std` namespace as an alias for another namespace anymore, but instead declares a fresh one and uses `using other_namespace;` in it. This is to ensure that both new headers can be included together. More importantly, it also avoids polluting well-known namespaces, which is not allowed for `std`.

As you can see, both `boost_or_std::auto_ptr` and `boost_or_std::shared_ptr` are declared intelligently in a similar fashion, based on the current build environment. They're either aliases for their implementations in the standard library/*TR 1* and *Boost* (in legacy compiler versions), or as *alias templates*⁴ for `std::unique_ptr` and `std::shared_ptr` (in C++11 and newer).

The reason why the name `boost_or_std::auto_ptr` has been used — as opposed to `boost_or_std::unique_ptr` — is because it is the common denominator. The idea is to make it obvious that this kind of smart pointer does **not** guarantee the safer semantics of `std::unique_ptr`. It could behave like the deprecated `std::auto_ptr` (in case of C++03), or it just *might* behave like the safer alternative (in case of modern C++).

The consequences are that COAST must not make use of the copy assignment semantics of `std::auto_ptr`, which it currently doesn't anyway. In addition to that, it also means that COAST has to keep using `boost_or_std::auto_ptr` until it eventually drops support for C++03, at which point all occurrences of `boost_or_std::auto_ptr` can safely be replaced with `std::unique_ptr`.

⁴http://en.cppreference.com/w/cpp/language/type_alias

8.4 Removing Obsolete Information

8.4.1 *PC-Lint* Magic Comments

There were quite a few of left over magic comments which disabled warnings from the static code analyzer *PC-Lint*. These were obsolete, because the IFS doesn't use that software anymore. More importantly, it was also the cause for at least one of the porting issues, as discussed before.

We removed all of them at once using the following *sed* one-liner:

```
sed -i 's,[\t ]*//lint.*,,' $(git grep -l //lint)
```

8.4.2 The `register` Keyword

There were a few occurrences of the `register` keyword. It's been deprecated for quite a while and will actually lose even its historical meaning with C++17. Prof. Sommerlad advised us to remove it completely, along the lines of [24, Keywords: The Lesser Ones]:

“Never write `register` . It's exactly as meaningful as whitespace.”

This was not as easy as a *sed* one-liner, but easy enough:

```
vi $(git grep -l '\<register\>')
```

This at least opened all files containing the word *register* at once. Using simple *Vim* commands, it was a matter of seconds.

Chapter 9

Optional Goal: Improving `Anything` Internals

We discussed this goal and the consensus was the same, as Prof. Sommerlad's decision. The internals are `long int` members and the interface is `AsLong()`. Most parts of the application do not care if it's a 32- or 64-bit integer, there is no discrepancy. Code that depends on the size is less common and often low-level functionality not using `Anything`. Another point is, that such a change would have a huge impact on all modules, since `Anything` is a commonly used data structure in COAST. So for the sake of simplicity, we chose not to optimize for 32-bit integers.

Chapter 10

Conclusion

The COAST framework has successfully been ported to 64-bit while maintaining compatibility with 32-bit systems. Of all adapted test suites, the difference in performance characteristics have been analyzed and explained. The newly added infrastructure to measure performance is completely independent of the COAST code itself, which means low coupling and maintenance.

The optional goal Performance History has been completed as well. Using its functionality, arbitrary combinations of test suites, revisions and performance measurement methods can be ran to produce a result in the versatile CSV format.

All new scripts have been developed with portability in mind. This means that they'll run on any Unix system, provided the required performance measurement tools (and Git) are available.

Another optional goal, the porting of additional test suites, has been achieved. All optional tests, have been ported.

All non-functional requirements have been met: COAST runs on Linux, can still be compiled on C++03, and the example application *CoastRecipes* still works. The test quality did not decrease, as they've merely been fixed.

Furthermore, the code base has been analyzed and prepared for C++11/14.

Bibliography

- [1] *[A portable, fast, and free implementation of the MD5 Message-Digest Algorithm (RFC 1321)]*. URL: <http://openwall.info/wiki/people/solar/software/public-domain-source-code/md5>.
- [2] Harsha S. Adiga. *Porting Linux applications to 64-bit systems*. Oct. 2007. URL: <http://www.ibm.com/developerworks/library/l-port64/index.html>.
- [3] *boost::shared_ptr.hpp*. URL: http://www.boost.org/doc/libs/1_60_0/boost/smart_ptr/shared_ptr.hpp.
- [4] C committee. *ISO C standard*. Apr. 2011. URL: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>.
- [5] C++ committee. *ISO C++ standard*. Mar. 2014. URL: <https://github.com/cplusplus/draft/blob/b7b8ed08ba4c111ad03e13e8524a1b746cb74ec6/papers/N3936.pdf>.
- [6] Dahna McConnachie (Computerworld). *Bruce Almighty: Schneier preaches security to Linux faithful*. Dec. 2007. URL: https://www.computerworld.com.au/article/46254/bruce_almighty_schneier_preaches_security_linux_faithful/?pp=3.
- [7] *CppCon 2015: Chandler Carruth "Tuning C++: Benchmarks, and CPUs, and Compilers! Oh My!"* URL: <https://youtu.be/nXaxk27zwlk>.
- [8] cppreference.com. *Chrono: date and time utilities*. URL: <http://en.cppreference.com/w/cpp/chrono>.
- [9] cppreference.com. *Fixed width integer types*. URL: <http://en.cppreference.com/w/cpp/types/integer>.
- [10] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*. Oct. 2013. URL: http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf.
- [11] *gethostbyname_r(3) - Linux man page*. URL: http://linux.die.net/man/3/gethostbyname_r.
- [12] *git-read-tree(1) Manual Page*. URL: <https://www.kernel.org/pub/software/scm/git/docs/git-read-tree.html>.
- [13] Brendan Gregg. *Linux perf examples*. URL: <http://www.brendangregg.com/perf.html>.
- [14] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Oct. 1999. URL: https://books.google.ch/books?id=5wBQEp6ruIAC&pg=PA76&redir_esc=y#v=onepage&q&f=false.
- [15] Andrey Karpov. *Viva64*. Sept. 2009. URL: <http://www.viva64.com/en/a/0050/>.
- [16] Andrey Karpov and Evgeniy Ryzhkov. *20 issues of porting C++ code on the 64-bit platform*. URL: http://www.gamedev.net/page/resources/_/technical/general-programming/20-issues-of-porting-c-code-on-the-64-bit-platform-r2419.
- [17] kernelnewbies.org. *Linux y2038*. URL: <http://kernelnewbies.org/y2038>.
- [18] *malloc(3): Linux man page*. URL: <http://linux.die.net/man/3/malloc>.
- [19] Stephen B. Morris. *Migrating C/C++ from 32-Bit to 64-Bit*. Mar. 2015. URL: <http://www.informit.com/articles/article.aspx?p=2339636>.
- [20] *Perf Wiki*. URL: https://perf.wiki.kernel.org/index.php/Main_Page.
- [21] *PerfTools Compendium*. URL: <https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/PerfTools>.
- [22] *proc(5): Linux man page*. URL: <http://linux.die.net/man/5/proc>.

- [23] Theo de Raadt. *Going long long on time_t to cope with 2,147,483,647+1*. URL: http://www.openbsd.org/papers/eurobsdcon_2013_time_t/.
- [24] Herb Sutter. *Keywords That Aren't (or, Comments by Another Name)*. Mar. 2003. URL: <http://www.drdobbs.com/keywords-that-arent-or-comments-by-anoth/184403859>.
- [25] *The GNU C Library — Date and Time*. URL: ftp://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_21.html.

Part III

Appendix

Appendix A

Self Reflection

As discussed in one of the meetings, we tend to create Redmine issues which impose too much work. We could have benefitted from smaller work packages, as it would avoid the situation of it looking like we're stuck somewhere or, even worse, not doing anything. Smaller packages means a higher frequency of solving issues on Redmine, and thus more to show off during the meetings.

We completed the three *Construction* milestones with minor delays. This resulted partly from underestimating effort needed, and partly from simply forgetting to close a left-over parent issue on Redmine (after solving all sub issues).

Regarding estimated efforts, we definitely could have spent a little more time on planning during the *Elaboration* phase. This could have lead to a more accurate estimations of the time needed to complete the milestones. Since it's not a usual software implementation project but instead porting work, which we're less used to, deciding on effort and time needed turned out hard.

Getting used to tracking time on Redmine took us a while. Initially, we used to postpone logging the time we spent on the project. But later on, we picked up the habit of doing it on a daily to weekly basis.

A.1 Thank You

At this point, we'd like to thank Marcel Huber and Prof. Peter Sommerlad. We greatly appreciate all the support we got regarding technical, as well as administrative issues.

Appendix B

Formalities

B.1 Declaration of Originality

We hereby confirm that we are the sole authors of this document and the described changes to the COAST framework.

B.2 Permissions



Vereinbarung

Gegenstand der Vereinbarung

Mit dieser Vereinbarung werden die Rechte über die Verwendung und die Weiterentwicklung der Ergebnisse der Projektarbeit "COAST 64bit Migration" von Philipp Schönenberg und Patrik Wenger unter der Betreuung von Peter Sommerlad und Marcel Huber geregelt.

Urheberrecht

Die Urheberrechte stehen den Studenten zu.

Verwendung

Die Ergebnisse der Arbeit dürfen von der HSR sowie von Peter Sommerlad nach Abschluss der Arbeit verwendet und weiter entwickelt werden. Den Studenten stehen keinerlei Verwertungsrechte am COAST Framework oder Teilen davon zu.

Rapperswil, den 31.3.16 P. Schönenberg, P. Wenger
Die Studenten

Rapperswil, den 31.3.16 P. Sommerlad, M. Huber
Betreuer der Projektarbeit

Rapperswil, den 1.4.16 178/0
Der Studiengangleiter

Appendix C

Project Plan

C.1 Organization

This term project is supervised by Marcel Huber of the IFS Institute For Software at HSR. As the head of the IFS lab and current lead developer of the COAST framework, he'll be the go-to person for inquiries regarding the framework throughout the course of this term project. Naturally, he'll be taking part at most project meetings.

The assigned expert is Peter Sommerlad, original developer of the COAST framework. As the director of the IFS, he'll attend the project meetings less frequently. Known for his immense experience in software engineering and the C++ language, he'll be available for especially tricky problems and important design decisions.

We, Philipp Schönenberg and Patrik Wenger, BSc students and authors of this document, will be using engineering methods acquired and practiced at HSR to achieve the term project goals, as well as properly document our decisions and progress. In addition, we'll also prepare for and lead all the project meetings.

Most communication between the parties will be verbal or via email. Project management, time tracking, meeting agendas and minutes, as well as some of the feedback from the supervisor and the expert will take place on the Redmine platform.

Appendix D

Infrastructural Problems

Requiring quite an extensive environment, getting up and running with COAST and Redmine on a provided VM wasn't exactly straight-forward. Here we'll explain some of the obstacles we deal with prior to the actual porting. These were huge time wasters.

D.1 Redmine: MySQL driver

One would imagine that setting Redmine up using the system's package manager (`apt`) would be easy. But it's not. We wasted hours trying to find the reason why the Redmine application was unable to establish a connection to the pre-installed MySQL server instance. It turns out that one has to specify the driver `mysql2` instead of just `mysql` in the application's database configuration file.

D.2 Software Versions on VM: Upgrade

The VM provided by HSR's IT department initially ran Ubuntu 14.04 LTS, a 2 years old release of the *Long Term Support* variant of Ubuntu releases. Due to several issues, including

- a mess resulting from attempting to get the *Boost* library to cooperate with *multilib* (support for 32-bit **and** 64-bit libraries on a single system),
- outdated root certificates used for SSL, and
- Python's `pip` command refusing to ignore invalid SSL certificates to be able to install COAST's Python dependency (namely *SConsider*¹)

an upgrade to Ubuntu 15.10 was required, which took us a few hours to plan and perform.

D.3 Redmine Bugs and Another Upgrade

D.3.1 Buggy Pre-Installed Version

Due to several bugs in the pre-installed version of Redmine, including one which made setting a task's parent impossible, we had to upgrade Redmine. We installed it manually (without `apt-get`) under `/usr/local/share/redmine`.

¹<https://coast-project.org/projects/sconsider>

Marcel Huber’s hint about Redmine’s official Docker image² unfortunately came too late. It might have eased the installation pain a bit.

D.3.2 Gantt Charts

The situation around Gantt charts on Redmine is catastrophic. First of all, the built-in Gantt chart functionality of Redmine is useless. Issues that depend on each other (such as *precedes*, *blocks*, ...) are not intelligently drawn that way. Even though the necessary information can be recorded on the issue tickets — and was in our case — it is not used while the Gantt chart is generated, and thus even dependent issues are placed at the same point on the timeline as the issue they depend on.

One would actually have to set **each and every** issue’s start date manually before the Gantt chart even starts making sense and possibly become a helpful insight about the project. And all issues would have to be updated manually, one by one, in case a milestone has to be postponed for say, a week. Of course this was out of the question.

One might think this could be easily fixed by installing one of the several Gantt chart plugins available for Redmine. Not the case. One of these plugins³ is ancient, meaning it doesn’t work on even remotely recent versions of Redmine.

Another one⁴ felt like spamware (sign up required) and didn’t work at all, so its plugin migration had to be manually undone. Waste of time.

D.4 SSH Access to VM

Three weeks into the term project, all of a sudden SSH access from outside the HSR network was blocked. A week later, in an attempt to adopt a suggestion in which we’d just tell `sshd` to also listen on a port in the 40000 – 40010 range, `sshd` refused to start completely and logged the reason as being:

```
Mar 16 18:05:44 sinv-56044 systemd[1]: ssh.service: Start request repeated too ↵  
↳ quickly.
```

Fortunately, the IT department was able to fix the problem on March 17th. On Marcel Huber’s request, the IT department then sent out an email explaining the recent change in the firewall, justifying its decision with some recent SSH-based attacks. It would have been nice (and professional) for them to send that email proactively instead of reactively, though.

D.5 Mails from Redmine

Unfortunately, for quite a while, Redmine was unable to send any emails⁵. The outgoing ports 587 and 645 are blocked. The PDF describing all details about the VM does not mention anything about this.

²https://hub.docker.com/_/redmine/

³http://www.redmine.org/plugins/redmine_better_gantt_chart

⁴<https://www.easyredmine.com/redmine-gantt-plugin>

⁵<http://sinv-56044.edu.hsr.ch/redmine/issues/75>

Luckily though, Marcel Huber mentioned that HSR has its own SMTP relay which can at least be used to send email to HSR accounts. With that information, it didn't take us long to adapt Redmine's configuration and fix this issue.

It's sad that something as important and central as this is not listed in the PDF, especially since the VM comes with Redmine pre-installed.

D.6 Cevalop

Due to indexing problems in Cevalop, all identifiers in COAST's source used to be marked red for weeks. Only with help from Thomas Corbat, we got it to work on April 1st. To do so, we had to:

- disable mockator plugin
- close all projects
- run `find /path/to/coast -name .cproject -delete`
- reopen project(s)
- rebuild index
- (readd SCons targets)

D.7 Jenkins

Jenkins was a huge pain to set up. On top of that, it's pretty foolishly designed, as in: If a build takes longer than the interval between periodic SCM checks, it just queues another build.

Around April 12th, we had a suddenly failing build. It turned out to be Jenkins itself. This definitely did not help the project progression.

Appendix E

perf stat diff

Diffs done with:

```
find . -name 'Coast*32*.perf' -execdir bash -c 'git diff --no-index "$1" "${1//32/64}'  
" _ {} ;
```

E.1 CoastEBCDICTest

```
diff --git a/./CoastEBCDICTest-32_optimized.perf b/./CoastEBCDICTest-64_optimized.perf  
--- a/./CoastEBCDICTest-32_optimized.perf  
+++ b/./CoastEBCDICTest-64_optimized.perf  
@@ -1,22 +1,22 @@  
-      5.280923      task-clock (msec)          #    0.865 CPUs utilized  
+      5.729020      task-clock (msec)          #    0.698 CPUs utilized  
-          26      context-switches            #    0.004 M/sec  
+          28      context-switches            #    0.004 M/sec  
-          3      cpu-migrations                #    0.498 K/sec  
+          3      cpu-migrations                #    0.421 K/sec  
-          232     page-faults                 #    0.039 M/sec  
+          268     page-faults                 #    0.038 M/sec  
-      8149282     cycles                      #    1.353 GHz  
+      7898533     cycles                      #    1.108 GHz  
-      17119007     instructions                #    2.98   insns per cycle  
+      15239481     instructions                #    2.78   insns per cycle  
-      3530628     branches                    #  586.313 M/sec  
+      3133209     branches                    #  439.478 M/sec  
-          77444     branch-misses              #    2.20% of all branches  
+          72810     branch-misses              #    2.35% of all branches  
- <not counted>    L1-dcache-loads              (0.68%)  
+ <not counted>    L1-dcache-loads              (1.81%)  
- <not counted>    L1-dcache-load-misses        (0.00%)  
- <not counted>    LLC-loads                    (0.00%)  
- <not counted>    LLC-load-misses              (0.00%)  
  
-      0.006105133 seconds time elapsed        ( +-  4.45% )  
+      0.008210330 seconds time elapsed        ( +-  8.61% )
```

Listing E.1: Diff: perf stat of CoastEBCDICTest

```

diff --git a/./CoastEBCDICTest-32_optimized.perf b/./CoastEBCDICTest-64_optimized.perf
--- a/./CoastEBCDICTest-32_optimized.perf
+++ b/./CoastEBCDICTest-64_optimized.perf
@@ -1,22 +1,22 @@
- 4.965388 task-clock (msec) # 0.977 CPUs utilized
+ 4.296779 task-clock (msec) # 0.935 CPUs utilized
- 29 context-switches # 0.006 M/sec
+ 25 context-switches # 0.006 M/sec
- 3 cpu-migrations # 0.593 K/sec
+ 3 cpu-migrations # 0.660 K/sec
- 229 page-faults # 0.045 M/sec
+ 271 page-faults # 0.060 M/sec
- 4995603 cycles # 0.987 GHz
+ 11445507 cycles # 2.518 GHz
<not supported> stalled-cycles-frontend
<not supported> stalled-cycles-backend
- 17123259 instructions # 3.75 insns per cycle
+ 15250695 instructions # 3.17 insns per cycle
- 3531819 branches # 697.613 M/sec
+ 3135411 branches # 689.903 M/sec
- 76899 branch-misses # 2.18% of all branches
+ 67291 branch-misses # 2.15% of all branches
<not counted> L1-dcache-loads (0.00%)
<not counted> L1-dcache-load-misses (0.00%)
<not counted> LLC-loads (0.00%)
<not counted> LLC-load-misses (0.00%)

- 0.005082447 seconds time elapsed ( +- 0.69% )
+ 0.004596529 seconds time elapsed ( +- 1.34% )

```

Listing E.2: Diff: perf stat of CoastEBCDICTest (#2)

E.2 CoastFoundationAnythingOptionalTest

```

diff --git a/./CoastFoundationAnythingOptionalTest-32_optimized.perf b/./CoastFoundationAnythingOptionalTest-64_optimized.perf
--- a/./CoastFoundationAnythingOptionalTest-32_optimized.perf
+++ b/./CoastFoundationAnythingOptionalTest-64_optimized.perf
@@ -1,22 +1,22 @@
- 15.067565 task-clock (msec) # 1.018 CPUs utilized
+ 10.794900 task-clock (msec) # 0.983 CPUs utilized
- 3 context-switches # 0.208 K/sec
+ 0 context-switches # 0.000 K/sec
- 0 cpu-migrations # 0.000 K/sec
+ 0 page-faults # 0.012 M/sec
- 170 page-faults # 0.020 M/sec
+ 216 page-faults # 0.235 GHz
- 3389355 cycles # 0.742 GHz
+ 7938349 cycles # 5.11 insns per cycle
- 23386741 instructions # 6.00 insns per cycle
+ 31660150 instructions # 670.483 M/sec
- 9667867 branches # 1076.740 M/sec
+ 11522220 branches # 1.54% of all branches
- 148462 branch-misses # 1.05% of all branches
+ 109614 branch-misses # 1279.061 M/sec (32.00%)
- 18443109 L1-dcache-loads (11.05%)
+ <not counted> L1-dcache-loads (0.00%)
- 164936 L1-dcache-load-misses # 0.75% of all L1-dcache hits (6.69%)
+ <not counted> L1-dcache-load-misses (0.00%)
<not counted> LLC-loads (0.00%)
<not counted> LLC-load-misses (0.00%)

- 0.014805273 seconds time elapsed ( +- 0.94% )
+ 0.010977443 seconds time elapsed ( +- 2.06% )

```

Listing E.3: Diff: perf stat of CoastFoundationAnythingOptionalTest

E.3 CoastFoundationBaseTest

```
diff --git a/./CoastFoundationBaseTest-32_optimized.perf b/./CoastFoundationBaseTest-64←
└ _optimized.perf
--- a/./CoastFoundationBaseTest-32_optimized.perf
+++ b/./CoastFoundationBaseTest-64_optimized.perf
@@ -1,22 +1,22 @@
-      801.351790 task-clock (msec)      #    0.563 CPUs utilized
+      560.921989 task-clock (msec)      #    0.465 CPUs utilized
-      503 context-switches             #    0.643 K/sec
+      282 context-switches             #    0.485 K/sec
-      74 cpu-migrations                 #    0.095 K/sec
+      82 cpu-migrations                 #    0.141 K/sec
-     10843 page-faults                  #    0.014 M/sec
+     13671 page-faults                  #    0.024 M/sec
-    2137690557 cycles                   #    2.732 GHz          (50.00%)
+   1517809325 cycles                   #    2.611 GHz          (49.76%)
-   3633629009 instructions              #    1.77 insns per cycle (62.45%)
+   3198772821 instructions              #    2.05 insns per cycle (62.20%)
-   770217125 branches                  #  984.519 M/sec        (62.61%)
+   634127162 branches                  # 1090.953 M/sec        (62.24%)
-   3481615 branch-misses                #    0.45% of all branches (62.65%)
+   2372624 branch-misses                #    0.37% of all branches (63.01%)
-   1211055612 L1-dcache-loads           # 1548.015 M/sec        (61.47%)
+   733776472 L1-dcache-loads           # 1262.390 M/sec        (61.27%)
-   7480169 L1-dcache-load-misses        #    0.62% of all L1-dcache hits (25.05%)
+   9135454 L1-dcache-load-misses        #    1.21% of all L1-dcache hits (25.07%)
-   3270711 LLC-loads                    #    4.181 M/sec        (25.28%)
+   4055687 LLC-loads                    #    6.977 M/sec        (25.13%)
-   2241473 LLC-load-misses              #   83.37% of all LL-cache hits (37.59%)
+   2450330 LLC-load-misses              #   58.10% of all LL-cache hits (37.36%)

-   1.423357572 seconds time elapsed    ( +-  1.86% )
+   1.207541715 seconds time elapsed    ( +-  1.53% )
```

Listing E.4: Diff: perf stat of CoastFoundationBaseTest

E.4 CoastFoundationIOTest

```
diff --git a/./CoastFoundationIOTest-32_optimized.perf b/./CoastFoundationIOTest-64←
└ _optimized.perf
--- a/./CoastFoundationIOTest-32_optimized.perf
+++ b/./CoastFoundationIOTest-64_optimized.perf
@@ -1,22 +1,22 @@
-      51.715175 task-clock (msec)      #    0.069 CPUs utilized
+      39.390288 task-clock (msec)      #    0.053 CPUs utilized
-      123 context-switches             #    0.003 M/sec
+      78 context-switches             #    0.002 M/sec
-      30 cpu-migrations                 #    0.634 K/sec
+      25 cpu-migrations                 #    0.619 K/sec
-     1177 page-faults                  #    0.025 M/sec
+     1265 page-faults                  #    0.031 M/sec
-     82184677 cycles                   #    1.735 GHz          (48.43%)
+    54540763 cycles                   #    1.351 GHz          (50.57%)
-   127110291 instructions              #    1.53 insns per cycle (65.38%)
+   95220942 instructions              #    1.44 insns per cycle (68.84%)
-   26192250 branches                  #  553.098 M/sec        (68.33%)
+   20165281 branches                  #  499.594 M/sec        (70.94%)
-   505625 branch-misses                #    2.05% of all branches (72.37%)
+   406588 branch-misses                #    2.16% of all branches (71.68%)
-   40203645 L1-dcache-loads           #  848.975 M/sec        (38.60%)
+   32746371 L1-dcache-loads           #  811.290 M/sec        (34.49%)
-   508840 L1-dcache-load-misses        #    1.18% of all L1-dcache hits (27.54%)
+   499924 L1-dcache-load-misses        #    1.59% of all L1-dcache hits (23.83%)
-   93253 LLC-loads                    #    1.969 M/sec        (27.24%)
+   97086 LLC-loads                    #    2.405 M/sec        (29.57%)
-   22438 LLC-load-misses              #   16.53% of all LL-cache hits (37.35%)
+   15978 LLC-load-misses              #    8.65% of all LL-cache hits (0.00%)

-   0.747026867 seconds time elapsed    ( +-  0.48% )
+   0.736938180 seconds time elapsed    ( +-  0.36% )
```

Listing E.5: Diff: perf stat of CoastFoundationIOTest

E.5 CoastFoundationMiscellaneousTest

```
diff --git a/./CoastFoundationMiscellaneousTest-32_optimized.perf b/./←
    CoastFoundationMiscellaneousTest-64_optimized.perf
--- a/./CoastFoundationMiscellaneousTest-32_optimized.perf
+++ b/./CoastFoundationMiscellaneousTest-64_optimized.perf
@@ -1,22 +1,22 @@
-      1.486727 task-clock (msec)          #    0.795 CPUs utilized
+      1.421020 task-clock (msec)          #    0.756 CPUs utilized
+              0 context-switches         #    0.000 K/sec
+              0 cpu-migrations            #    0.000 K/sec
-             120 page-faults              #    0.073 M/sec
+             143 page-faults              #    0.086 M/sec
-      2415664 cycles                      #    1.472 GHz
+      3676541 cycles                      #    2.202 GHz
-      5849795 instructions                 #    1.82 insns per cycle
+      5493075 instructions                 #    1.74 insns per cycle
-      1100742 branches                    #   670.891 M/sec
+      1039058 branches                    #   622.400 M/sec
-      28799 branch-misses                  #    2.62% of all branches
+      26595 branch-misses                  #    2.55% of all branches
<not counted> L1-dcache-loads              (0.00%)
<not counted> L1-dcache-load-misses         (0.00%)
<not counted> LLC-loads                     (0.00%)
<not counted> LLC-load-misses               (0.00%)

-      0.001870784 seconds time elapsed    ( +-  4.76% )
+      0.001879033 seconds time elapsed    ( +-  5.15% )
```

Listing E.6: Diff: perf stat of CoastFoundationMiscellaneousTest

```
--- a/./CoastFoundationMiscellaneousTest-32_optimized.perf
+++ b/./CoastFoundationMiscellaneousTest-64_optimized.perf
@@ -1,22 +1,22 @@
-      1.473729 task-clock (msec)          #    0.826 CPUs utilized
+      1.365291 task-clock (msec)          #    0.817 CPUs utilized
+              0 context-switches         #    0.000 K/sec
+              0 cpu-migrations            #    0.000 K/sec
-             118 page-faults              #    0.073 M/sec
+             141 page-faults              #    0.095 M/sec
-      3612495 cycles                      #    2.250 GHz
+      2244385 cycles                      #    1.505 GHz
<not supported> stalled-cycles-frontend
<not supported> stalled-cycles-backend
-      5845165 instructions                 #    1.81 insns per cycle
+      5494081 instructions                 #    1.69 insns per cycle
-      1099772 branches                    #   684.863 M/sec
+      1039934 branches                    #   697.166 M/sec
-      29382 branch-misses                  #    2.67% of all branches
+      26869 branch-misses                  #    2.59% of all branches
<not counted> L1-dcache-loads              (0.00%)
<not counted> L1-dcache-load-misses         (0.00%)
<not counted> LLC-loads                     (0.00%)
<not counted> LLC-load-misses               (0.00%)

-      0.001783771 seconds time elapsed    ( +-  3.27% )
+      0.001671264 seconds time elapsed    ( +-  3.15% )
```

Listing E.7: Diff: perf stat of CoastFoundationMiscellaneousTest (#2)

E.6 CoastFoundationPerfTest

```
diff --git a/./CoastFoundationPerfTest-32_optimized.perf b/./CoastFoundationPerfTest-64←
└ _optimized.perf
--- a/./CoastFoundationPerfTest-32_optimized.perf
+++ b/./CoastFoundationPerfTest-64_optimized.perf
@@ -1,22 +1,22 @@
- 4376.864150 task-clock (msec) # 0.997 CPUs utilized
+ 2353.611768 task-clock (msec) # 1.018 CPUs utilized
- 157 context-switches # 0.037 K/sec
+ 268 context-switches # 0.120 K/sec
- 3 cpu-migrations # 0.001 K/sec
+ 8 cpu-migrations # 0.004 K/sec
- 572064 page-faults # 0.134 M/sec
+ 128991 page-faults # 0.058 M/sec
- 10338462403 cycles # 2.420 GHz (49.97%)
+ 6096891093 cycles # 2.739 GHz (49.99%)
- 15056696125 instructions # 1.47 insns per cycle (62.50%)
+ 10764417330 instructions # 1.79 insns per cycle (62.51%)
- 3303084188 branches # 773.045 M/sec (62.53%)
+ 2511942286 branches # 1128.511 M/sec (62.52%)
- 10241137 branch-misses # 0.31% of all branches (62.57%)
+ 6941908 branch-misses # 0.28% of all branches (62.63%)
- 5505914644 L1-dcache-loads # 1288.589 M/sec (62.36%)
+ 3265283310 L1-dcache-loads # 1466.955 M/sec (62.23%)
- 100663906 L1-dcache-load-misses # 1.83% of all L1-dcache hits (25.00%)
+ 153880607 L1-dcache-load-misses # 4.66% of all L1-dcache hits (24.97%)
- 33676739 LLC-loads # 7.882 M/sec (24.99%)
+ 33468200 LLC-loads # 15.036 M/sec (24.98%)
- 9048701 LLC-load-misses # 26.55% of all LL-cache hits (37.50%)
+ 5063149 LLC-load-misses # 16.92% of all LL-cache hits (37.46%)

- 4.390040827 seconds time elapsed ( +- 2.37% )
+ 2.311206670 seconds time elapsed ( +- 2.31% )
```

Listing E.8: Diff: perf stat of CoastFoundationPerfTest

```
diff --git a/./CoastFoundationPerfTest-32_optimized.perf b/./CoastFoundationPerfTest-64←
└ _optimized.perf
--- a/./CoastFoundationPerfTest-32_optimized.perf
+++ b/./CoastFoundationPerfTest-64_optimized.perf
@@ -1,22 +1,22 @@
- 2844.325392 task-clock (msec) # 0.949 CPUs utilized
+ 1824.969129 task-clock (msec) # 0.968 CPUs utilized
- 43 context-switches # 0.015 K/sec
+ 36 context-switches # 0.020 K/sec
- 16 cpu-migrations # 0.005 K/sec
+ 10 cpu-migrations # 0.006 K/sec
- 569743 page-faults # 0.195 M/sec
+ 124109 page-faults # 0.069 M/sec
- 8469778140 cycles # 2.900 GHz (49.93%)
+ 5307459072 cycles # 2.930 GHz (49.96%)
<not supported> stalled-cycles-frontend
<not supported> stalled-cycles-backend
- 14088820832 instructions # 1.68 insns per cycle (62.49%)
+ 10082927402 instructions # 1.94 insns per cycle (62.54%)
- 3085224807 branches # 1056.234 M/sec (62.52%)
+ 2390206280 branches # 1319.516 M/sec (62.56%)
- 8932930 branch-misses # 0.29% of all branches (62.62%)
+ 6682683 branch-misses # 0.28% of all branches (62.59%)
- 5141274856 L1-dcache-loads # 1760.128 M/sec (62.34%)
+ 3150127669 L1-dcache-loads # 1739.031 M/sec (62.14%)
- 102120684 L1-dcache-load-misses # 1.99% of all L1-dcache hits (24.98%)
+ 128111859 L1-dcache-load-misses # 4.11% of all L1-dcache hits (25.00%)
- 33566283 LLC-loads # 11.491 M/sec (24.98%)
+ 31034498 LLC-loads # 17.133 M/sec (25.01%)
- 6478049 LLC-load-misses # 20.18% of all LL-cache hits (37.43%)
+ 3762116 LLC-load-misses # 13.73% of all LL-cache hits (37.50%)

- 2.996582628 seconds time elapsed ( +- 0.37% )
+ 1.884923152 seconds time elapsed ( +- 0.32% )
```

Listing E.9: Diff: perf stat of CoastFoundationPerfTest (#2)

E.7 CoastFoundationTest

```
diff --git a/./CoastFoundationTest-32_optimized.perf b/./CoastFoundationTest-64_optimized.perf
└─ perf
--- a/./CoastFoundationTest-32_optimized.perf
+++ b/./CoastFoundationTest-64_optimized.perf
@@ -1,22 +1,22 @@
-      19.735887 task-clock (msec)      #    0.010 CPUs utilized
+      15.951401 task-clock (msec)      #    0.008 CPUs utilized
-      3 context-switches              #    0.140 K/sec
+      4 context-switches              #    0.258 K/sec
-      1 cpu-migrations                #    0.047 K/sec
+      0 cpu-migrations                #    0.000 K/sec
-      193 page-faults                 #    0.009 M/sec
+      229 page-faults                 #    0.015 M/sec
-      12943445 cycles                  #    0.604 GHz              (36.61%)
+      10931446 cycles                  #    0.706 GHz              (49.28%)
-      36316915 instructions            #    0.75 insns per cycle   (51.39%)
+      40788557 instructions            #    2.33 insns per cycle   (69.50%)
-      12580222 branches                # 586.884 M/sec             (56.79%)
+      13596250 branches                # 877.649 M/sec             (84.04%)
-      154013 branch-misses             #    1.04% of all branches  (74.19%)
+      159681 branch-misses             #    1.39% of all branches
-      37529611 L1-dcache-loads          # 1750.806 M/sec            (40.46%)
+      16531229 L1-dcache-loads          # 1067.105 M/sec            (31.15%)
+      325050 L1-dcache-load-misses      #    1.02% of all L1-dcache hits (30.94%)
-      <not counted> L1-dcache-load-misses (12.94%)
-      <not counted> LLC-loads             (17.41%)
+      <not counted> LLC-loads             (0.00%)
-      <not counted> LLC-load-misses      (0.00%)

-      1.924435833 seconds time elapsed  ( +- 0.07% )
+      1.917767435 seconds time elapsed  ( +- 0.02% )
```

Listing E.10: Diff: perf stat of CoastFoundationTest

E.8 CoastFoundationTimeTest

```
diff --git a/./CoastFoundationTimeTest-32_optimized.perf b/./CoastFoundationTimeTest-64_optimized.perf
└─ perf
--- a/./CoastFoundationTimeTest-32_optimized.perf
+++ b/./CoastFoundationTimeTest-64_optimized.perf
@@ -1,22 +1,22 @@
-      21.379210 task-clock (msec)      #    0.011 CPUs utilized
+      15.673228 task-clock (msec)      #    0.008 CPUs utilized
-      15 context-switches              #    0.697 K/sec
+      3 context-switches              #    0.173 K/sec
-      0 cpu-migrations                #    0.000 K/sec
-      192 page-faults                 #    0.009 M/sec
+      229 page-faults                 #    0.013 M/sec
-      15493827 cycles                  #    0.720 GHz              (37.17%)
+      44781696 cycles                  #    2.581 GHz              (43.20%)
-      43867467 instructions            #    0.99 insns per cycle   (53.42%)
+      49214838 instructions            #    2.16 insns per cycle   (62.33%)
-      14607041 branches                # 678.525 M/sec             (62.25%)
+      8720687 branches                # 502.651 M/sec             (75.53%)
-      152180 branch-misses             #    0.96% of all branches  (83.93%)
+      99710 branch-misses             #    0.84% of all branches  (92.28%)
-      23269053 L1-dcache-loads          # 1080.892 M/sec            (42.31%)
+      24884427 L1-dcache-loads          # 1434.312 M/sec            (33.66%)
-      288248 L1-dcache-load-misses      #    0.94% of all L1-dcache hits (27.93%)
+      305900 L1-dcache-load-misses      #    1.39% of all L1-dcache hits (18.09%)
-      <not counted> LLC-loads             (14.56%)
+      150152 LLC-loads                 #    8.655 M/sec            (0.00%)
-      <not counted> LLC-load-misses      (0.00%)
+      48619 LLC-load-misses            # 102.21% of all LL-cache hits (0.00%)

-      1.925149354 seconds time elapsed  ( +- 0.05% )
+      1.919801165 seconds time elapsed  ( +- 0.07% )
```

Listing E.11: Diff: perf stat of CoastFoundationTimeTest

```

diff --git a/./CoastFoundationTimeTest-32_optimized.perf b/./CoastFoundationTimeTest-64←
└─ _optimized.perf
--- a/./CoastFoundationTimeTest-32_optimized.perf
+++ b/./CoastFoundationTimeTest-64_optimized.perf
@@ -1,22 +1,22 @@
- 42.248368 task-clock (msec) # 0.022 CPUs utilized
+ 40.950845 task-clock (msec) # 0.021 CPUs utilized
- 4 context-switches # 0.085 K/sec
+ 3 context-switches # 0.085 K/sec
- 0 cpu-migrations # 0.000 K/sec
- 189 page-faults # 0.004 M/sec
+ 227 page-faults # 0.006 M/sec
- 41320172 cycles # 0.883 GHz (48.38%)
+ 29347377 cycles # 0.830 GHz (53.25%)
<not supported> stalled-cycles-frontend
<not supported> stalled-cycles-backend
- 80491813 instructions # 1.91 insns per cycle (62.86%)
+ 61214071 instructions # 1.84 insns per cycle (66.15%)
- 19732038 branches # 421.548 M/sec (66.97%)
+ 15659184 branches # 442.684 M/sec (68.09%)
- 178664 branch-misses # 0.98% of all branches (72.83%)
+ 170270 branch-misses # 1.10% of all branches (69.25%)
- 34766381 L1-dcache-loads # 742.736 M/sec (51.92%)
+ 21751073 L1-dcache-loads # 614.901 M/sec (42.11%)
- 152028 L1-dcache-load-misses # 0.46% of all L1-dcache hits (23.11%)
+ 214481 L1-dcache-load-misses # 0.99% of all L1-dcache hits (22.62%)
- 24189 LLC-loads # 0.517 M/sec (20.59%)
+ 25666 LLC-loads # 0.726 M/sec (0.00%)
- 128 LLC-load-misses # 0.31% of all LL-cache hits (34.14%)
+ 191 LLC-load-misses # 0.58% of all LL-cache hits (0.00%)

- 1.949338859 seconds time elapsed ( +- 0.07% )
+ 1.937886946 seconds time elapsed ( +- 0.09% )

```

Listing E.12: Diff: perf stat of CoastFoundationTimeTest (#2)

E.9 CoastMTFoundationTest

```

diff --git a/./CoastMTFoundationTest-32_optimized.perf b/./CoastMTFoundationTest-64←
└─ _optimized.perf
--- a/./CoastMTFoundationTest-32_optimized.perf
+++ b/./CoastMTFoundationTest-64_optimized.perf
@@ -1,22 +1,22 @@
- 166.080839 task-clock (msec) # 0.025 CPUs utilized
+ 144.504034 task-clock (msec) # 0.022 CPUs utilized
- 8063 context-switches # 0.050 M/sec
+ 8272 context-switches # 0.055 M/sec
- 919 cpu-migrations # 0.006 M/sec
+ 638 cpu-migrations # 0.004 M/sec
- 376 page-faults # 0.002 M/sec
+ 1160 page-faults # 0.008 M/sec
- 291993399 cycles # 1.812 GHz (61.53%)
+ 217651172 cycles # 1.447 GHz (58.62%)
- 205407448 instructions # 0.95 insns per cycle (73.81%)
+ 243197511 instructions # 1.26 insns per cycle (69.51%)
- 50440560 branches # 313.065 M/sec (70.22%)
+ 50570070 branches # 336.240 M/sec (68.73%)
- 680013 branch-misses # 1.34% of all branches (72.97%)
+ 579754 branch-misses # 1.26% of all branches (69.13%)
- 59692822 L1-dcache-loads # 370.491 M/sec (44.82%)
+ 57216269 L1-dcache-loads # 380.430 M/sec (46.25%)
- 2845482 L1-dcache-load-misses # 3.84% of all L1-dcache hits (33.72%)
+ 3461980 L1-dcache-load-misses # 6.15% of all L1-dcache hits (34.87%)
- 1166465 LLC-loads # 7.240 M/sec (34.70%)
+ 902299 LLC-loads # 5.999 M/sec (33.65%)
- 141050 LLC-load-misses # 13.76% of all LL-cache hits (49.01%)
+ 70657 LLC-load-misses # 7.22% of all LL-cache hits (47.89%)

- 6.602245364 seconds time elapsed ( +- 0.29% )
+ 6.618256784 seconds time elapsed ( +- 0.20% )

```

Listing E.13: Diff: perf stat of CoastMTFoundationTest

E.10 CoastRegexTest

```
diff --git a/./CoastRegexTest-32_optimized.perf b/./CoastRegexTest-64_optimized.perf
--- a/./CoastRegexTest-32_optimized.perf
+++ b/./CoastRegexTest-64_optimized.perf
@@ -1,22 +1,22 @@
-      118.817049 task-clock (msec)      #    0.431 CPUs utilized
+      87.468881 task-clock (msec)      #    0.362 CPUs utilized
-      169 context-switches             #    0.001 M/sec
+      159 context-switches             #    0.002 M/sec
-      12 cpu-migrations                 #    0.096 K/sec
+      5 cpu-migrations                 #    0.057 K/sec
-      1482 page-faults                 #    0.012 M/sec
+      1301 page-faults                 #    0.015 M/sec
-      321647088 cycles                  #    2.564 GHz              (49.94%)
+      250847241 cycles                  #    2.869 GHz              (49.45%)
-      630683480 instructions             #    1.86 insns per cycle   (62.58%)
+      549978529 instructions             #    2.35 insns per cycle   (62.03%)
-      155815519 branches                # 1242.132 M/sec           (62.93%)
+      131763501 branches                # 1507.113 M/sec           (62.80%)
-      491549 branch-misses              #    0.32% of all branches (64.90%)
+      315208 branch-misses              #    0.25% of all branches (64.99%)
-      251352167 L1-dcache-loads          # 2003.732 M/sec           (57.17%)
+      171284821 L1-dcache-loads          # 1959.158 M/sec           (55.15%)
-      747126 L1-dcache-load-misses      #    0.30% of all L1-dcache hits (24.77%)
+      1357131 L1-dcache-load-misses      #    0.83% of all L1-dcache hits (25.14%)
-      169554 LLC-loads                  #    1.352 M/sec           (24.66%)
+      509399 LLC-loads                  #    5.827 M/sec           (24.44%)
-      51608 LLC-load-misses              #   32.63% of all LL-cache hits (36.16%)
+      52964 LLC-load-misses              #   21.48% of all LL-cache hits (35.67%)

-      0.275821866 seconds time elapsed  ( +-  1.49% )
+      0.241413638 seconds time elapsed  ( +-  3.25% )
```

Listing E.14: Diff: perf stat of CoastRegexTest

```
diff --git a/./CoastRegexTest-32_optimized.perf b/./CoastRegexTest-64_optimized.perf
--- a/./CoastRegexTest-32_optimized.perf
+++ b/./CoastRegexTest-64_optimized.perf
@@ -1,22 +1,22 @@
-      120.045145 task-clock (msec)      #    0.459 CPUs utilized
+      81.437651 task-clock (msec)      #    0.350 CPUs utilized
-      135 context-switches             #    0.001 M/sec
+      135 context-switches             #    0.002 M/sec
-      1 cpu-migrations                 #    0.009 K/sec
+      1 cpu-migrations                 #    0.012 K/sec
-      1486 page-faults                 #    0.013 M/sec
+      1303 page-faults                 #    0.015 M/sec
-      311287132 cycles                  #    2.754 GHz              (51.00%)
+      191305663 cycles                  #    2.258 GHz              (50.01%)
<not supported> stalled-cycles-frontend
<not supported> stalled-cycles-backend
-      654903046 instructions             #    2.21 insns per cycle   (63.79%)
+      457793461 instructions             #    2.07 insns per cycle   (62.97%)
-      157405445 branches                # 1392.456 M/sec           (63.62%)
+      123582091 branches                # 1458.496 M/sec           (62.91%)
-      494398 branch-misses              #    0.32% of all branches (64.62%)
+      344673 branch-misses              #    0.28% of all branches (65.30%)
-      252720030 L1-dcache-loads          # 2235.638 M/sec           (57.83%)
+      177554196 L1-dcache-loads          # 2095.465 M/sec           (56.92%)
-      1752659 L1-dcache-load-misses     #    0.70% of all L1-dcache hits (24.99%)
+      1444585 L1-dcache-load-misses     #    0.90% of all L1-dcache hits (25.88%)
-      79855 LLC-loads                   #    0.706 M/sec           (25.29%)
+      286722 LLC-loads                   #    3.384 M/sec           (24.72%)
-      7242 LLC-load-misses               #    7.10% of all LL-cache hits (37.23%)
+      5160 LLC-load-misses               #    3.36% of all LL-cache hits (36.07%)

-      0.261589596 seconds time elapsed  ( +-  1.48% )
+      0.232891130 seconds time elapsed  ( +-  1.87% )
```

Listing E.15: Diff: perf stat of CoastRegexTest (#2)

E.11 CoastStorageTest

```
diff --git a/./CoastStorageTest-32_optimized.perf b/./CoastStorageTest-64_optimized.perf
--- a/./CoastStorageTest-32_optimized.perf
+++ b/./CoastStorageTest-64_optimized.perf
@@ -1,22 +1,22 @@
-    935.406719    task-clock (msec)      #    0.988 CPUs utilized
+    664.506267    task-clock (msec)      #    1.012 CPUs utilized
-      14         context-switches      #    0.015 K/sec
+      9          context-switches      #    0.014 K/sec
-      3          cpu-migrations        #    0.003 K/sec
+      3          cpu-migrations        #    0.005 K/sec
-    10646        page-faults          #    0.011 M/sec
+    17454        page-faults          #    0.027 M/sec
-    2346787694   cycles                #    2.511 GHz          (49.97%)
+    1714948299   cycles                #    2.659 GHz          (49.65%)
-    4417236736   instructions          #    1.84   insns per cycle (62.56%)
+    3333571997   instructions          #    1.94   insns per cycle (62.28%)
-    1037345630   branches              # 1109.851 M/sec        (62.64%)
+    795558932    branches              # 1233.665 M/sec        (62.41%)
-    5263167      branch-misses         #    0.51% of all branches (62.79%)
+    3227658      branch-misses         #    0.41% of all branches (62.73%)
-    1685309524   L1-dcache-loads       # 1803.104 M/sec        (61.87%)
+    1003506186   L1-dcache-loads       # 1556.126 M/sec        (61.54%)
-    5212057      L1-dcache-load-misses #    0.31% of all L1-dcache hits (24.99%)
+    7883968      L1-dcache-load-misses #    0.77% of all L1-dcache hits (25.19%)
-    895701       LLC-loads             #    0.958 M/sec        (24.96%)
+    1303771      LLC-loads             #    2.022 M/sec        (25.01%)
-    571941       LLC-load-misses       #   55.26% of all LL-cache hits (37.45%)
+    728242       LLC-load-misses       #   55.65% of all LL-cache hits (37.29%)

-    0.946956762 seconds time elapsed   ( +-  0.98% )
+    0.656905051 seconds time elapsed   ( +-  1.05% )
```

Listing E.16: Diff: perf stat of CoastStorageTest

E.12 CoastSystemFunctionsTest

```
diff --git a/./CoastSystemFunctionsTest-32_optimized.perf b/./CoastSystemFunctionsTest-64_
    _optimized.perf
--- a/./CoastSystemFunctionsTest-32_optimized.perf
+++ b/./CoastSystemFunctionsTest-64_optimized.perf
@@ -1,22 +1,22 @@
-    1.217745     task-clock (msec)      #    0.686 CPUs utilized
+    1.159816     task-clock (msec)      #    0.677 CPUs utilized
-      0          context-switches      #    0.000 K/sec
+      0          context-switches      #    0.000 K/sec
-      0          cpu-migrations        #    0.000 K/sec
+    115         page-faults          #    0.078 M/sec
+    137         page-faults          #    0.094 M/sec
-    3271492      cycles                #    2.222 GHz
+    3113712      cycles                #    2.128 GHz
-    4739259      instructions          #    1.38   insns per cycle
+    4542215      instructions          #    1.47   insns per cycle
-    867446       branches              #   589.233 M/sec
+    841703       branches              #   575.239 M/sec
-    23911        branch-misses         #    2.76% of all branches
+    22439        branch-misses         #    2.67% of all branches
-    <not counted> L1-dcache-loads       (0.00%)
+    <not counted> L1-dcache-load-misses (0.00%)
-    <not counted> L1-dcache-load-misses (0.00%)
+    <not counted> LLC-loads             (0.00%)
-    <not counted> LLC-load-misses       (0.00%)

-    0.001776178 seconds time elapsed   ( +-  7.40% )
+    0.001711945 seconds time elapsed   ( +-  8.54% )
```

Listing E.17: Diff: perf stat of CoastSystemFunctionsTest

Appendix F

COAST Setup Cookbook

Applicability: This howto has been tested on a headless Ubuntu Server 15.4.

F.1 Getting Started

This part of the howto explains how to get COAST up and running. You'll need to be in your home directory, so make sure of that, e.g. by running `cd` in your shell.

F.1.1 About *Boost*

We'll be using a system-wide installed distribution of *Boost* in this howto as opposed to one from the *3rdparty* directory. This should work fine in most common Linux distributions. The only exception is Ubuntu where if and only if you want to be able to compile COAST for both 32 AND 64-bit, the multilib architecture results in a disaster. For that case, the solution is to use *Boost* from the *3rdparty* directory.

F.1.2 Installing Dependencies

F.1.2.1 Ubuntu Packages

```
apt-get --no-install-recommends install g++-multilib libboost-dev:i386 \
libssl-dev:i386 libboost-regex-dev:i386 \
libboost-system-dev:i386 doxygen graphviz python-pip
```

PIP Packages

Get the appropriate PIP config like this:

```
mkdir -p ~/.pip && wget -O ~/.pip/pip.conf https://raw.githubusercontent.com/\
marcelhuberfoo/docker-coast-recipes/master/pip.conf
```

F.1.2.2 Python's *virtualenv*

This is how *virtualenv* is cloned and installed:

```
git clone --single-branch --branch master --depth 1 \
    https://github.com/pypa/virtualenv.git
python2 $HOME/virtualenv/virtualenv.py $VENVDIR
```

F.1.2.3 Shell Startup (optional)

Add this to your `~/.bashrc` or `~/.zshrc` or similar to load the virtual environment automatically on shell startup, **only if** you want it to be available all the time. We'll provide a simple startup script to start Cevalop which does this already. See below.

```
export VENVDIR=$HOME/.venv27scons
[ -d "$VENVDIR" ] && . $VENVDIR/bin/activate
```

If you prefer to do it manually, just add the following line to the respective startup file:

```
export VENVDIR=$HOME/.venv27scons
```

And then, if needed, run the following command:

```
. $VENVDIR/bin/activate@.
```

F.1.3 Cloning COAST

Here we'll have to disable SSL verification (encryption only) to work around missing signature of *Let's Encrypt* certificate.

```
GIT_SSL_NO_VERIFY=true git clone \
    https://gerrit.coast-project.org/p/coast
cd coast
```

F.1.3.1 Sub Repositories

Here's how to clone the required sub repositories:

```
git clone --single-branch --branch master --depth 1 \
    https://gerrit.coast-project.org/p/wdscripts.git
git clone --single-branch --branch master --depth 1 \
    https://gerrit.coast-project.org/p/recipes.git
git clone --single-branch --branch master --depth 1 \
    https://gerrit.coast-project.org/p/zlib.git 3rdparty/zlib
```

For 3rd party *Boost* and *OpenSSL*, execute these commands as well:

```
git clone --single-branch --branch master --depth 1 \
    https://gerrit.coast-project.org/p/boost.git 3rdparty/boost
git clone --single-branch --branch master --depth 1 \
```


F.1.4 Further Dependencies

The dependencies include:

- sconsider
- python-ldap
- pyopenssl

These dependencies are listed in a file within the COAST repository. That's the reason why we had to check out COAST first. Install them like this:

```
pip install -U -r ~/coast/requirements.txt
```

F.1.5 Example Webapp: *CoastRecipes*

F.1.5.1 Generating the API Documentation

For 3rd party *boost*, add `--with-src-boost=3rdparty/boost`:

```
scons -u --jobs=2 --ignore-missing --doxygen-only
```

For 3rd party *boost*, add `--with-src-boost=3rdparty/boost`:

```
scons -u --jobs=2 CoastRecipes && \  
cd apps/CoastRecipes && ln -s ../../doc/Coast/html COASTDoc
```

F.1.5.2 Starting It

For 3rd party *boost*, add `--with-src-boost=3rdparty/boost`.

```
scons -u --jobs=2 CoastRecipes --run
```

F.1.6 Trouble Shooting

F.1.6.1 Class Index is Empty

When starting the example app and you see an empty class index, you might need to get rid of untracked files:

```
git clean -xf --dry-run # (1)  
git clean -xf           # (2)
```

It's most likely the *Doxyfile* that has to be deleted.

Important: To avoid deleting non-recoverable files, check the output of command (1) first. Then proceed with command (2).

F.2 Development

This part of the howto is about making changes to COAST.

F.2.1 Headless

If you're running a headless Ubuntu server installation and would like to make changes to COAST using Cevalop, but don't plan to run X on your server, you can display Cevalop on a remote X server. Assuming you have your X server running on your local machine (like XQuartz on OS X), just access the Ubuntu server like this:

```
ssh -X IPADDR
```

Replace IPADDR with the correct IP address. The option `-X` tells SSH to perform the required measures (forwarding the X traffic and announcing the display through the `DISPLAY` environment variable).

F.2.2 Getting and Running Cevalop

Get and unpack Cevalop using the following command:

```
wget https://www.cevelop.com/cevelop/downloads/\
cevelop-1.4.0-201512021228-linux.gtk.x86_64.tar.gz
tar xf cevelop
```

Make sure you get the latest version. They tend to run better.

As a convenience, you might want to use the following start script.

```
#!/bin/sh

# COAST
export VENVDIR=$HOME/.venv27scons
[ -d "$VENVDIR" ] && . $VENVDIR/bin/activate

export SCONSFLAGS="--ignore-missing --with-src-zlib=3rdparty/zlib \
    --with-bin-openssl=3rdparty/openssl --build-cfg=debug \
    --warnlevel=medium --enable-Trace --config=force \
    --archbits=64"
# for 3rdparty boost, add --with-src-boost=3rdparty/boost

~/cevelop-*/cevelop.sh &
```

Listing F.1: Convenience script to start Cevalop for COAST development

Save it under `~/start_cevelop.sh` and make it executable. It'll activate the virtual environment, set options appropriate for development on 64-bit COAST, and finally start Cevalop.

If needed, adapt the options to your needs.

Start the script like this:

```
~/start_cvelop.sh
```

When run the first time, you'll have to import COAST as a C++ application.

F.2.3 Running Test Suites

To run a test suite, e.g. `CoastFoundationBaseTest`, inside a running shell, make sure you've activated the virtual environment (see above), and then run the following command:

```
scons --ignore-missing --with-src-boost=3rdparty/boost \  
      --with-src-zlib=3rdparty/zlib \  
      --with-bin-openssl=3rdparty/openssl \  
      --build-cfg=debug --use-lang-features=c++03 \  
      --archbits=32 --run-force CoastFoundationBaseTest
```

Adapt the command to your needs. For example, change/add the following options to produce an optimized 64-bit executable using C++14:

```
--build-cfg=optimized --use-lang-features=c++14 --archbits=64
```

F.2.4 Trouble Shooting

F.2.4.1 Cvelop index wrong

In this case Cvelop's index isn't built properly and all class names are shown with a red squiggly line. Procedure to fix it:

1. close projects
2. run `find /path/to/coast/workspace -name .cproject -delete`
3. open projects
4. click on SCons button

Appendix G

Usage of perf/perf-history

The following usage can be shown by running `./perf-history --help`:

SYNOPSIS

- (1) `./perf-history [--method=METHOD] [--max-count=N] <commit>...`
- (2) `./perf-history [--method=METHOD] --stdin`

The purpose of this script is to generate a history of performance measurement results across a given commit range.

BUILD TIME

By default, this utility works in bisect fashion and limits the number of commits to be built to save build time.

As an example, instead of building (*) all commits of a given range,

some are skipped (.) as shown here:

..........*.....*.....*.....*.....*.....*.....*

The others are skipped with the idea that the results can be interpolated.

USAGE

- (1) The `<commit>` argument as it is understood by `git-rev-list`. Useful for commit ranges.
- (2) Useful when a specific list of commits (not a range) should be measured or other functionality of `git-rev-list` is needed.
Example:

```
git-rev-list --reverse master...feature1 -- only/here/ | ./perf-history --  
  ↵ stdin
```

This would otherwise be impossible without emulating more of `git-rev-list`'s functionality.

ALGORITHM

- 1) Parse options and determine set of commits of interest.
- 2) Print summary of what it's about to do.
- 3) Get confirmation from the user.
- 4) Create a new directory `./history-YYYYMMDD-XXXXXX`
- 4) For each commit:

```

    a) Create result directory ("./history-YYYYMMDD-XXXXXX/<commit>/")
    b) Checkout commit
    a) Run the selected METHOD tool
        This builds, runs, measures, and saves results.
    b) Run it again with --export
        This extracts interesting values.
5) Accumulate the result to a CSV file

YYYYMMDD is the current date. The X's in "XXXXXX" are randomly chosen once per
run.

FILES AND DIRECTORIES
-----
* history-YYYYMMDD-XXXXXX/commits.txt
    a list of commits that were built and measured

* history-YYYYMMDD-XXXXXX/<commit>/
    a directory for each commit, containing the results of METHOD

* history-YYYYMMDD-XXXXXX/<commit>/method.log
    the output of the measurement script (for diagnostics)

* history-YYYYMMDD-XXXXXX/<commit>/result.csv
    the performance result of one commit

* history-YYYYMMDD-XXXXXX/result.csv
    the accumulated results of all commits as CSV

* history-YYYYMMDD-XXXXXX/test_names.txt
    list of tests to be built and measured for each commit

OPTIONS
-----
-h --help            show this help and exit
-m --method=METHOD (default: time) time, perf_stat, or valgrind
-n --max-count=N     (default: 16) max. # of commits to measure,
                     or 0 to build all commits
    --stdin          take commit list from STDIN, disable max. # commits

If --stdin is not given, it makes sense to pass roughly a power
of two to --max-count if you want the number of skipped, consecutive commits
not to differ too much.

For details about the performance measurement ("method") scripts, read the
usage of the lower-level script(s):

    ./with_* --help

About <commit>...: Consult the manpage if unsure how to specify commit ranges:

    man gitrevisions

PREREQUISITES
-----
* virtual ENV for COAST has to be activated already

ENVIRONMENT VARIABLES
-----
Honored env vars are:

* ALL_ARCHBITS
    Defaults to the platform's hardware (either "32" or "64").
    But could be e.g. "64 32" to compare both 64-bit and 32-bit over time.
* TEST_NAMES
    File containing test names to build, run, and measure.
    Defaults to a list of predefined core tests.

```

Listing G.1: Usage of perf-history script

Appendix H

Usage of perf/with_*

All with_* scripts share the same usage, which can be shown using the --help option:

SYNOPSIS

- (1) ./with_perf
- (2) ./with_perf TEST...
- (3) ./with_perf --all-tests
- (4) ./with_perf --diff=TEST
- (5) ./with_perf --export [--all-tests | TEST...]

The purpose of this script is to build and run test suites and measure performance. The results are saved into files in under ./perf_results to be able to compare the differences between 32-bit and 64-bit builds.

USAGE

- (1) Measures the predefined set of core tests (SA's mandatory goal).
- (2) Measures the given tests.
- (3) Measures all available tests (determined dynamically using scons)
- (4) Prints a command which would show the performance differences of 32/64-bit builds for the given test.
- (5) Extracts common values from results and prints them as CSV.

MEASUREMENT METHODS

Depending on which tool you're running, the measurement method is:

with_time	=>	time
with_perf	=>	perf (linux-tools)
with_perf_stat	=>	perf stat (linux-tools)
with_valgrind	=>	Valgrind
with_massif	=>	Valgrind massif tool

PREREQUISITES

* virtual ENV for COAST has to be activated already

OPTIONS

-a --all-tests	run all tests available (otherwise core only)
-d --diff=TEST	show command to get performance difference of TEST
-e --export	export common values results as CSV
-h --help	show this help and exit

ENVIRONMENT VARIABLES

The following env variables are honored (with default value):

* PERF_DIR	"/perf_results"
* ALL_ARCHBITS	"64 32"
* TEST_NAMES	temp file, content depending on options/args
* TIMES	"20"

TIMES is currently only used by with_time. It defines how many times to run a test to get a more accurate timing measurement.

Listing H.1: Usage of with_* performance measurement scripts